Contents

[NUMPY 2](#_Toc187141140)

[WHAT IS NUMPY 2](#_Toc187141141)

[NUMPY ARRAYS 2](#_Toc187141142)

[NUMPY FUNCTIONS 2](#_Toc187141143)

[arange 2](#_Toc187141144)

[EXAMPLE 3](#_Toc187141145)

[np.ones() and np.zeros() 3](#_Toc187141146)

[EXAMPLE 3](#_Toc187141147)

[np.linspace() 3](#_Toc187141148)

[EXAMPLE 4](#_Toc187141149)

# NUMPY

## WHAT IS NUMPY

NumPy is a Python library that stands for "Numerical Python". It is a fundamental package for scientific computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

## NUMPY ARRAYS

* A NumPy array, also known as ndarray, is a central data structure in NumPy library. It is a multi-dimensional grid of elements of the same type. NumPy arrays can have any number of dimensions, but most commonly they are 1D (one-dimensional), 2D (two-dimensional), or 3D (three-dimensional).
* NumPy arrays are widely used in scientific computing, data analysis, machine learning, and numerical computations due to their efficiency and versatility.

|  |  |
| --- | --- |
| **import numpy as np**    **# Creating a 1D array**  **arr1 = np.array([1, 2, 3, 4, 5])**  **print(arr1) # Output: [1 2 3 4 5]**    **# Creating a 2D array**  **arr2 = np.array([[1, 2, 3], [4, 5, 6]])**  **print(arr2)**  **# Output:**  **# [[1 2 3]**  **# [4 5 6]]**    **# Creating a 3D array**  **arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])**  **print(arr3)**  **# Output:**  **# [[[1 2]**  **# [3 4]]**    **# [[5 6]**  **# [7 8]]]** | Key features and characteristics of NumPy arrays:   1. **Homogeneous Data**: NumPy arrays contain elements of the **same data type** 2. **Fixed Size**: Once a NumPy array is created, its size is fixed and cannot be changed. To modify the size, a new array needs to be created. 3. **Fast and Efficient**: NumPy arrays are implemented in C, making them faster and more efficient compared to Python lists. They allow for vectorized operations, which can perform computations on entire arrays without the need for loops. 4. **Powerful Indexing and Slicing**: NumPy arrays support advanced indexing and slicing operations, allowing easy access to elements or sub-arrays based on specific conditions or criteria. 5. **Mathematical Operations**: NumPy arrays provide a wide range of mathematical functions and operations to perform calculations efficiently on arrays**, such as element-wise operations, linear algebra operations, statistical functions.** |

## NUMPY FUNCTIONS

## arange

* The `**np.arange()`** function in NumPy is used to create an array with regularly spaced values within a specified range.
* **It is similar to the Python built-in `range()` function but returns an array instead of a list**.
* **SYNTAX**

**np.arange(start, stop, step, dtype=None)**

|  |  |
| --- | --- |
| `start` (optional): | The starting value of the sequence. If not provided, the default value is 0. |
| stop | The end value of the sequence. It is exclusive, so the generated sequence will stop before reaching this value. |
| step | The step size between consecutive values in the sequence. If not provided, the default value is 1 |
| dtype | The data type of the elements in the resulting array. If not specified, NumPy will determine it based on the other input arguments |

### EXAMPLE

|  |
| --- |
| **import numpy as np**    **# Example 1: Generate a sequence of numbers from 0 to 9**  **arr1 = np.arange(10)**  **print(arr1) # Output: [0 1 2 3 4 5 6 7 8 9]** |
| **# Example 2: Generate a sequence of even numbers from 2 to 10**  **arr2 = np.arange(2, 11, 2)**  **print(arr2) # Output: [2 4 6 8 10]** |
| **# Example 3: Generate a sequence of floating-point numbers from 0 to 1 with a step of 0.1**  **arr3 = np.arange(0, 1, 0.1)**  **print(arr3) # Output: [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]** |
| **print(np.arange(0, 10, 2, dtype=float)) #OUTPUT [0. 2. 4. 6. 8.]** |

## np.ones() and np.zeros()

* In NumPy, the functions `np.ones()` and `np.zeros()` are used to create arrays filled with ones and zeros, respectively. These functions allow we to easily create arrays of desired shapes and sizes filled with the specified values.
* **SYNTAX** - This function creates an array of ones with the specified shape

**np.ones(shape, dtype=None, order='C')**

|  |  |
| --- | --- |
| shape | The shape of the array, specified as a tuple of integers. For example, `(3, 4)` creates a 2D array with 3 rows and 4 columns |
| dtype(optional): | The data type of the elements in the array. If not specified, the default data type is `float64`. |
| order(optional): | The order in which the array is stored in memory. It can be `'C'` for row-major (C-style) or `'F'` for column-major (Fortran-style). The default is `'C' |

* **SYNTAX** - This function creates an array of zeros with the specified shape

**np.zeros(shape, dtype=None, order='C')**

### EXAMPLE

|  |  |
| --- | --- |
| **# Example 1: Create a 1D array of ones with 5 elements**  **arr1 = np.ones(5)**  **print(arr1)**  **# Output: [1. 1. 1. 1. 1.] 🡪 “.”(dot) represents the float** | **# Example 3: Create a 3D array of ones with dimensions 2x3x2**  **arr3 = np.ones((2, 3, 2))**  **print(arr3)**  **# Output:**  **# [[[1. 1.]**  **# [1. 1.]**  **# [1. 1.]]**    **# [[1. 1.]**  **# [1. 1.]**  **# [1. 1.]]]** |
| **# Example 2: Create a 2D array of zeros with 3 rows and 4 columns**  **arr2 = np.zeros((3, 4)) 🡨 THE ROWS AND COLUMNS NEED TO BE PASSED AS TUPLE**  **print(arr2)**  **# Output:**  **# [[0. 0. 0. 0.]**  **# [0. 0. 0. 0.]**  **# [0. 0. 0. 0.]]** |

## np.linspace()

* the `np.linspace()` function is used to create an array with evenly spaced values over a specified interval. It is particularly useful when we want to generate a sequence of numbers with a specific number of elements.
* `np.linspace()` is commonly used in various scientific and numerical computing applications, such as plotting graphs, creating test datasets, and generating time series data.

**np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)**

|  |  |
| --- | --- |
| **start** | The starting value of the sequence |
| **stop** | The end value of the sequence |
| **num(optional):** | The number of equally spaced values to generate between `start` and `stop`. The default value is 50 i.e How many numbers we want between start and stop (including stop) |
| **endpoint(optional)** | Whether or not to include the `stop` value in the sequence. **If `True`, the sequence will include `stop`. The default value is `True** |
| **retstep(optional)** | Whether or not to return the spacing between consecutive values. If `True`, the function will return a tuple containing the array and the step value. The default value is `False`. |
| **dtype(optional)** | The data type of the elements in the resulting array. If not specified, NumPy will determine it based on the other input arguments. |

### EXAMPLE

|  |
| --- |
| **# Example 1: Generate a sequence of 10 numbers from 0 to 1 (including 1)**  **arr1 = np.linspace(0, 1, num=10)**  **print(arr1)**  **# Output: [0. 0.11111111 0.22222222 0.33333333 0.44444444 0.55555556**  **# 0.66666667 0.77777778 0.88888889 1. ]** |
| **# Example 2: Generate a sequence of 5 numbers from -2 to 2 (including -2 and 2)**  **arr2 = np.linspace(-2, 2, num=5)**  **print(arr2)**  **# Output: [-2. -1. 0. 1. 2.]** |
| **# Example 3: Generate a sequence of 6 numbers from 1 to 10 (excluding 10) and return the step value**  **arr3, step = np.linspace(1, 10, num=6, retstep=True)**  **print(arr3) # [ 1. 3.8 6.6 9.4 12.2 15. ]**  **print(step) 🡺 # 3.2 🡸 THIS IS THE SPACING BETWEEN THE ELEMENTS** |