**NumPy Material**

1. **INTRODUCTION**

**NumPy**, which stands for **Numerical Python, is an open-source** [**Python**](https://www.tutorialspoint.com/python/index.htm)**library consisting of multidimensional and single-dimensional array elements.**

* NumPy is a fundamental package for numerical computation in Python. It provides mathematical functions to compute data as well as functions to operate multi-dimensional arrays and matrices efficiently.

The NumPy API (Application Programming Interface) in Python is used primarily for numerical computing. It provides support for a wide range of mathematical functions to operate on data efficiently.

Applications of NumPy :

1. **Data Analysis**: NumPy offers rapid and effective array operations, rendering it well-suited for tasks like data cleansing, filtering, and transformation. It is predominantly used in the analysis and scientific handling of data, particularly when working with extensive, large datasets.
2. **Data Visualisation**: NumPy doesn't provide data visualisation but supports Matplotlib and Seaborn libraries to generate plots and visualisations from numerical data.
3. **Machine Learning and Artificial Intelligence:** Different machine learning and deep learning frameworks in Python, such as TensorFlow, and PyTorch, rely on NumPy arrays for handling input data, model parameters, and outputs.

**Example 1 :**

# Importing NumPy Array

import numpy as np

# Creating an array using np.array() method

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

# Printing

print(arr) # Prints [10 20 30 40 50]

**Operations using NumPy**

1. Mathematical and logical operations on arrays.
2. Fourier transforms and routines for shape manipulation.
3. Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

**Comparison B/W List & NumPy**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **NumPy** | **List** |
| Memory Storage | NumPy uses a contiguous block of memory, which improves cache efficiency and access speed. | Python lists consist of pointers to objects, leading to more memory fragmentation and slower access. |
| Data Types | NumPy supports homogeneous data types (all elements are of the same type), leading to more efficient memory use. | Python lists can contain heterogeneous data types (elements can be of different types), resulting in higher memory overhead. |
| Operations | NumPy uses vector operations that leverage SIMD (Single Instruction, Multiple Data) for parallel processing. | Python lists rely on loop-based operations, which are slower due to the overhead of Python's interpreted nature. |
| Efficiency | NumPy is written in C and optimized for performance, reducing the execution time of numerical operations. | Python lists are executed as Python byte-code, which is generally slower compared to compiled C code. |
| Memory Usage | NumPy requires less memory due to fixed data types and contiguous storage. | Python lists use more memory because each element is a separate Python object with additional overhead. |
| Broadcasting | NumPy supports broadcasting, allowing operations on arrays of different shapes without creating additional copies. | Python lists do not support broadcasting, making element-wise operations less efficient. |
| Performance | Better cache utilization due to contiguous memory storage, leading to faster access and processing. | Poor cache utilization because of scattered memory allocation, slowing down access. |
| Functionality | NumPy provides a rich set of mathematical functions and tools optimized for array operations. | Python lists are Limited to basic operations and lack advanced mathematical capabilities. |

**Installing NumPy Using pip**

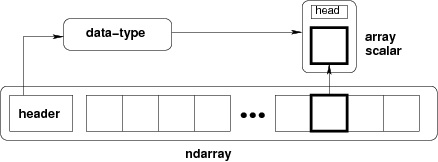
* pip install numpy

**Verify Installation :**

* import numpy as np
* print(np.\_\_version\_\_)

**NumPy ndarray**

* The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes a collection of items of the same type, which can be accessed using a zero-based index.
* Each item in an ndarray takes the same size of block in the memory and is represented by a data-type object called **dtype**.



**Creating an ndarray**

* numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

|  |  |
| --- | --- |
| 1 | object  Any object exposing the array interface method returns an array, or any (nested) sequence. |
| 2 | **dtype**  Desired data type of array, optional |
| 3 | **copy**  Optional. By default (true), the object is copied |
| 4 | **order**  C (row major) or F (column major) or A (any) (default) |
| 5 | **subok**  By default, returned array forced to be a base class array. If true, sub-classes passed through |
| 6 | **ndmin**  Specifies minimum dimensions of resultant array |

Example: Create a One-dimensional Array

import numpy as np

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

print(a)

Example: Create a Multi-dimensional Array

import numpy as np

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

print(a)

Example: Specify Minimum Dimensions

import numpy as np

a = np.array([1, 2, 3, 4, 5], ndmin=2)

print(a)

Example: Specify Data Type

import numpy as np

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

print(a)

Indexing Scheme

* **Shape −** The shape of an array is a tuple of integers representing the size of the array along each dimension. For example, the shape of a 2x3 array is (2, 3).
* **Strides −** Strides are the number of bytes to step in each dimension when traversing the array. The strides determine how to move from one element to the next in each dimension.

Example:

import numpy as np

# Creating a 2x3 array in row-major order

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

print("Array:")

print(a)

print("Shape:", a.shape)

print("Strides:", a.strides)

We can create a NumPy array using various function provided by the Python NumPy library.

* Using numpy.array() Function
* Using numpy.zeros() Function
* Using numpy.ones() Function
* Using numpy.arange() Function
* Using numpy.linspace() Function
* Using numpy.random.rand() Function
* Using numpy.empty() Function
* Using numpy.full() Function

Example: Creating a 2D NumPy Array

import numpy as np

# Creating a 2D array from a list of lists

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

print("2D Array:\n", arr)

Using numpy.zeros() Function

* numpy.zeros(shape, dtype=float, order='C')

import numpy as np

# Creating an array of zeros

arr = np.zeros(5)

print(arr)

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

import numpy as np

# Creating an array of ones

arr = np.ones(3)

print(arr)

Example: Creating 2D array of ones

import numpy as np

# Creating 2D array of ones

array\_2d = np.ones((4, 3))

print(array\_2d)

Example: Creating a Fortran-ordered array of ones

import numpy as np

# Creating Fortran-ordered array of ones

array\_F = np.ones((4, 3), order='F')

**Using numpy.arange() Function**

* The **numpy.arange()** function creates an array by generating a sequence of numbers based on specified start, stop, and step values. It is similar to Python's built-in range() function.

import numpy as np

# Providing just the stop value

array1 = np.arange(10)

print("array1:", array1)

# Providing start, stop and step value

array2 = np.arange(1, 10, 2)

print("array2:",array2)

**Using numpy.linspace() Function**

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

Example:

import numpy as np

# Creating an array of 10 evenly spaced values from 0 to 5

array1 = np.linspace(0, 5, num=10)

print("array1:",array1)

# Creating an array with 5 values from 1 to 2, excluding the endpoint

array2 = np.linspace(1, 2, num=5, endpoint=False)

print("array2:",array2)

# Creating an array and returning the step value

array3, step = np.linspace(0, 10, num=5, retstep=True)

print("array3:",array3)

print("Step size:", step)

**Using random.rand() Function**

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

Example:

import numpy as np

# Generating a single random float

random\_float = np.random.rand()

print("random\_float:",random\_float)

# Generating a 1D array of random floats

array\_1d = np.random.rand(5)

print("array\_1d:",array\_1d)

# Generating a 2D array of random floats

array\_2d = np.random.rand(2, 3)

print("array\_2d:",array\_2d)

# Generating a 3D array of random floats

array\_3d = np.random.rand(2, 3, 4)

print("array\_3d:",array\_3d)

Using numpy.full() Function

import numpy as np

array1 = np.full((2, 3), 5)

print(array1)

Changing Shape

numpy.reshape(arr, newshape, order='C')

import numpy as np

# Create a 1D array with 8 elements

a = np.arange(8)

print('The original array:')

print(a)

print('\n')

# Reshape the array to a 2D array with shape (4, 2)

b = a.reshape(4, 2)

print('The modified array:')

print(b)

numpy.ndarray.flat

import numpy as np

# Creating a 2D numpy array

array\_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Using flat to iterate over elements

for item in array\_2d.flat:

print(item)

numpy.transpose(a, axes=None)

import numpy as np

# Original 2D array

array\_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Transposing the 2D array

transposed\_2d = np.transpose(array\_2d)

print("Original array:\n", array\_2d)

print("Transposed array:\n", transposed\_2d)

ndarray.T

import numpy as np

# Creating a 2D array (matrix)

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

# Transposing the matrix

transposed\_matrix = matrix.T

print("Original Matrix:")

print(matrix)

print("Transposed Matrix:")

print(transposed\_matrix)

numpy.split(ary, indices\_or\_sections, axis=0)

import numpy as np

# Create an array

arr = np.arange(9)

print("Original array:")

print(arr)

# Split the array into 3 equal parts

result = np.split(arr, 3)

print("\nSplit array into 3 equal parts:")

for i, sub\_array in enumerate(result):

print(f"Sub-array {i+1}:")

print(sub\_array)

Stacking Arrays Using stack() Function

import numpy as np

# arrays

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

arr2 = np.array([4, 5, 6])

arr3 = np.array([7, 8, 9])

# Stack arrays along a new axis

stacked\_arr = np.stack((arr1, arr2, arr3), axis=0)

print("Stacked Array along a new axis (Axis 0):")

print(stacked\_arr)

**NumPy Indexing & Slicing**

* slice object is constructed by giving **start, stop**, and **step** parameters to the built-in **slice** function.

Example 1 :

import numpy as np

a = np.arange(10)

s = slice(2,7,2)

print a[s]

Example 2 :

import numpy as np

a = np.arange(10)

b = a[2:7:2]

print b

Example 3 :

# slice single item

import numpy as np

a = np.arange(10)

b = a[5]

print b

Example 4 :

# slice items starting from index

import numpy as np

a = np.arange(10)

print a[2:]

Example 5 :

# slice items between indexes

import numpy as np

a = np.arange(10)

print a[2:5]

Advanced Indexing :

import numpy as np

x = np.array([[1, 2], [3, 4], [5, 6]])

y = x[[0,1,2], [0,1,0]]

print y

NumPy Shape Attribute :

import numpy as np

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

print (a.shape)

NumPy ReShape Attribute :

import numpy as np

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

b = a.reshape(3,2)

print (b)