Numpy Notebook

August 12, 2025

1 NumPy — Numerical Python

NumPy is the core library for numerical computing in Python, providing: - Fast and memory-efficient multi-dimensional arrays - Mathematical functions for linear algebra, statistics, and more - Tools for integrating with other scientific libraries

1.1 Brief History

- In **1995** *Numeric* library released by Jim Hugunin (early numerical computing in Python).
- In **2001** *Numarray* created for handling large datasets more efficiently.
- In **2005** Travis Oliphant merged Numeric and Numarray into **NumPy** (Numerical Python).
- Today, it is maintained by the open-source community and NumPy Developers, forming the **foundation for nearly all Python data science libraries**.

1.2 Why NumPy is Important

- **Performance** –> Uses optimized C code under the hood, much faster than plain Python lists.
- **Memory Efficiency** –> Stores elements in contiguous memory blocks.
- **Powerful Functions** –> Mathematical, statistical, and linear algebra operations.
- **Multidimensional Arrays** –> Supports 1D, 2D, and nD arrays with broadcasting.
- **Interoperability** –> Works with Pandas, Matplotlib, Scikit-learn, TensorFlow, and more.

1.3 Key Facts

- **Core object**: `ndarray` (N-dimensional array).
- **Vectorized operations**: No need for explicit Python loops.
- **Broadcasting**: Allows operations on arrays of different shapes.
- **Indexing types**: Basic slicing, advanced (fancy) indexing, boolean masking.
- **Random module**: For reproducible simulations and experiments.

It is the foundation of most of the data sciences and ML libraries like, Pandas, Scikit-Learn, Pytorch, Tensorflow, etc.

[1]:
```
# installation process
# pip install numpy

import numpy as np
```

1.4 Creating Numpy arrays

There are **Two** methods of creating Numpy arrays 1. Through lists 2. Through tuples

[ ]:
```
array = np.array([1, 2, 3, 4, 5])

print(f"Array: {array}")
print(f"Type of array: {type(array)}")
```

With DataType

[ ]:
```
arr = np.array([1, 2, 3, 4.555, '5'], dtype = 'i')
print(arr, type(arr))

arr2 = np.array([2, 3, 4, 6, 8], dtype = 'f')
print(arr2, type(arr2))
```

1.5 NumPy vs List

[ ]:
```
List = [1, 2, 3, 4, 5]
print(List * 2)

Array = np.array([1, 2, 3, 4, 5])
print(Array * 2)
```

1.5.1 Execution time taken

It is the comparasion between the time taken by a list to execute VS the numpy array

[ ]:
```
import numpy as np
import time
t = time.time()
# Execution time for List
List = [2 * i for i in range(20000)]

print(f"Execution time: {time.time()-t}")

u = time.time()
# Execution time for numpy arrays
npArr = np.arange(20000) * 2

print(f"Execution time: {time.time()-u}")
```

1.5.2 MultiDimensional arrays

Arrays having more than one dimensions are called multidimensional numpy arrays

[ ]:
```
arr1 = np.array([[1, 2, 3],
                 [4, 5, 6], [7, 8, 9]])  # 2D array
print(arr1, arr1.ndim)
```

```
arr2 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr2, arr2.ndim)  # 3D array
```

1.5.3 Creating arrays from scratch

[ ]:
```
# Special values

zero = np.zeros((2, 4), dtype='i')  # zero or null matrix
print(f"Zeros array: {zero}")

ones = np.ones((3, 3), dtype='i')
print(f"Ones array: {ones}")  # Matrix with all entities 1

full = np.full((3, 5), 6, dtype='i')   # full(dimensions in form of tuple,␣
 ↪offset)
print(f"Full array with 6s: {full}")

random = np.random.rand(5, 3)    #creates an array with random values from [0,␣
 ↪1)
print(f"Random float array: {random}")

# to generate random numbers:
randint = np.random.randint(1, 20)
print(f"Your single integer: {randint}")

# to generate ana array of random integers:
rand = np.random.randint(1, 20, 5)  # 1D array
randintArray = np.random.randint(1, 20, (3, 4))  # 2D array of random values
print(f"Your 2D random array: {randintArray}")

# Sequence array:
seq1 = np.arange(1, 20)   # prints values sequence wise from 1 to 20
seq = np.arange(0, 16, 3)  # prints from 0-16 with interval of 3
print(f"Our sequenced array is: {seq}")


# Diagonal 2D array

diagonal = np.eye(4, dtype='i')
print(f"Diagonal array: \n {diagonal}")
```

1.6 Vectors, Matrices and Tensors

- **Vectors:** These are one dimentional arrays
- **Matrices:** These are two dimensional arrays
- **Tensors:** These are multidimensional arrays

[ ]:
```
# Creating Vector

array = np.array((1, 2, 3, 4, 5))
print(f"Vector: {array}")

# Creating Matrix

Matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"Matrix: \n{Matrix}")

# Creating Tensors (Multidimensional arrays)

tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(f"Tensor: \n{tensor}")
```

1.7 Array Properties

We can check the shape, size, dimensions and data type of an array

1.7.1 Data Types

Data type of the array elements

[ ]:
```
array = np.array([1, 2, 3, 4, 5])
array1 = np.array([1., 2., 3., 4., 5.])
array2 = np.array(['abc', 'def', 'ghi'])
array3 = np.array([True, False])
array4 = np.array(['Hy', 2, 3, 4, 'Hello'])
array5 = np.array(['hy', 2., 3.5, 4.4, 'By'])
array6 = np.array([True, 'Hy', 'Hello', False])
# Data type
print(f"Data type: {array.dtype}")  # int32 in case of integer array
print(f"Data type: {array1.dtype}")  # float64 in case of floating point array,␣
 ↪as well as an array having both integers and floats
print(f"Data type: {array2.dtype}")  # <U3 in case of string literals
print(f"Data type: {array3.dtype}")  # bool in case of boolean
print(f"Data type: {array4.dtype}")  # <U11 in case of an array having strings␣
 ↪and integers
print(f"Data type: {array5.dtype}")  # <U32 in case of an array having both␣
 ↪floats and strings
print(f"Data type: {array6.dtype}")  # <U5 in case of an array having both␣
 ↪boolean and string vals
```

1.7.2 Dimensions

Number of dimensions (axes) of the array

[ ]:
```python
array = np.array([1, 2, 3, 4, 5])
print(f"Dimensions: {array.ndim}")  # 1D

Matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"Dimensions: {Matrix.ndim}")  #2D

tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(f"Dimensions: {tensor.ndim}")   # 3D
```

1.7.3 Shape

Tuple showing the size along each dimension

[ ]:
```python
array = np.array([1, 2, 3, 4, 5])
print(f"Shape: {array.shape}")  # (5,)

Matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"Shape: {Matrix.shape}")  # (3, 3)

tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(f"Shape: {tensor.shape}")   # (2, 2, 3)
```

1.7.4 Size

Total number of elements in the array

[ ]:
```python
# Prints the num of elements

array = np.array([1, 2, 3, 4, 5])
print(f"Size: {array.size}")  # 5

Matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"Size: {Matrix.size}")  # 9

tensor = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(f"Size: {tensor.size}")   # 12
```

1.8 Basic Mathematical Operations

Sum

[ ]:
```python
# Sum of 2 numpy arrays
# size must be same for both arrays
x = np.arange(1, 11)
y = np.arange(6, 16)

sum = x + y
print(f"Sum is {sum}")
```

Subtraction

[ ]:
```
# Subtraction
sub = x - y
print(f"Subtraction is {sub}")
```

Multipication

[ ]:
```
# Multiplication
mul = x * y
print(f"Multiplication is {mul}")
```

Division

[ ]:
```
# Division
div = x / y
print(f"Division is {div}")
```

Modulus

[ ]:
```
# Modulus
mod = x % y
print(f"Modulus is {mod}")
```

1.9 Comparasional operations

Greater than

[ ]:
```
# Greater than
gt = x > y
print(f"x > y: {gt}")
```

Squaring

[ ]:
```
x = np.arange(1, 6)
y = np.arange(1, 6)

Square = x ** y
print(f"Square is: {Square}")
```

Square root

[ ]:
```
x = np.arange(1, 8)

sq_rt = np.sqrt(x)  # Return values in floating point numbers
print(f"Square root is: {sq_rt}")
```

Equal to, not equal to

[ ]:
```
# Equal to
eq = x == y
print(f"x == y: {eq}")
# Not equal to
neq = x != y
print(f"x != y: {neq}")
```

Exponential values

[ ]:
```
y = np.exp(x)   # e -> 2.718
print(f"Exponent times number is: {y}")
```

1.10 Aggregate Operations

1.10.1 Some aggregate functions

[ ]:
```
# Sum of all elements
total_sum = arr.sum()
print(f"Sum of all elements: {total_sum}")

# Minimum element
min_val = arr.min()
print(f"Minimum element: {min_val}")

# Maximum element
max_val = arr.max()
print(f"Maximum element: {max_val}")

# Mean of all elements
mean_val = arr.mean()
print(f"Mean of all elements: {mean_val}")

# Standard deviation
std_val = arr.std()
print(f"Standard deviation: {std_val}")

# Sum along columns (axis=0)
col_sum = arr.sum(axis=0)
print(f"Sum along columns: {col_sum}")

# Sum along rows (axis=1)
row_sum = arr.sum(axis=1)
print(f"Sum along rows: {row_sum}")
```

1.11 Indexing and Slicing

[ ]:
```
import numpy as np

arr = np.array([5, 7, 20, 25, 19, 75])
print(arr[1])
# print(arr[4])  # Index error
print(arr[-4])
# print(arr[-5])   # Index error
print(arr[0:4])  # print values from index 0 to 3
print(arr[2:])  # same as arr[2:7]
```

```
print(arr[:4])  # same as arr[0:4]
print(arr[-4:])   # not same as arr[-4:-1]

# Broadcasting --> altering several values in an array at once by a specific␣
 ↪sequence
arr[0:3] = 3
print(arr)

# 2D arrays

arr2D = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(arr2D)

arr2D[0:3, 1:3] = 5
print(arr2D)
```

1.12 Accessing 2D array elements

[ ]:
```
arr2D = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])

print(arr2D[0])  # prints row 1
print(arr2D[0][0])
print(arr2D[3][2])

# access array column

print(arr2D[:, 1])  # column 2 (at index 1)
```

1.13 Elements Selection (On the basis of some condition)

[ ]:
```
# Create a 5x5 matrix with random integers between 1 and 14
matrix = np.random.randint(1, 15, (5, 5))
print(matrix)

# Define a 4x3 2D array
arr2D = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9],
                  [10, 11, 12]])

# Select elements from 'matrix' greater than 10 (boolean indexing)
mat = matrix[matrix > 10]
print(mat)

# Select odd elements from 'arr2D' (elements % 2 == 1)
mat2 = arr2D[arr2D % 2 == 1]
print(mat2)
```

2 Advance Topics

Views, copies, Fancy indexing,...

2.1 Matrix Inversion (using numpy.linalg)

linagl –> linear algebra, a module for operations like inversion, determinant, eigenvalues, etc.

[ ]:
```
import numpy as np
import numpy.linalg as la

matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Calculate and print the inverse of the matrix
# Note: This matrix is singular and does NOT have an inverse, so this will␣
 ↪raise an error
print(la.inv(matrix))

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = la.eig(matrix)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Alternatively, you could print both together like this:
# print(la.eig(matrix))
```

2.2 Views & Copies

View: When slicing, we get a view by default (shared data) (actual change of array)

Copy: Does not change actual array (use .copy)

[ ]:
```
# View

Arr = np.arange(26)
# print(Arr)

b = Arr[2:7]
print(b)

b[0] = -1200
print(b)

print(Arr)  # altering b also changed Arr
```

[ ]:
```
# Copy

Arr = np.arange(26)
# print(Arr)

b = Arr[2:7].copy()
print(b)

b[0] = -1200
print(b)

print(Arr)  # altering b also changed Arr
```

2.3 Slicing Tricks

[ ]:
```
array = np.arange(101)
print(array)

# array[:n]  --> n step slicing
print(array[::3])

# array[:-n]  --> backward every n step
print(array[::-3])

# array[::-1]  --> Reverses an array
print(array[::-1])
```

2.4 Finding and Modifying elements

np.argwhere(condition) returns indices where the condition is true

[ ]:
```
array = np.arange(101)

index = np.argwhere(array % 5 == 0)
array[index] = -10
print(array)
```

2.5 Accessing Rows And Columns of an matrix

[ ]:
```
Mat = np.round(10 * np.random.rand(5, 4)).astype(int)  # --> similar as randint
print(Mat)
print(Mat[1, 2])  # same as print(Mat[1][2])
print(Mat[1, :])  # prints full row
print(Mat[:, 1])  # prints full column
print(Mat[1:3, 2:4])  # print specific sub-matrix
```

3 Transpose arrays

–> Rows converted in columns, and vice versa

[ ]:
```
import numpy as np

Mat = np.round(10 * np.random.rand(5, 4)).astype(int)
arr = np.arange(7)
print(arr.T)
#.T for transpose
print(Mat)  # Original Matrix
print(Mat.T)  # Transpose
```

4 Sorting Arrays

- axis = 0 => Sorts columns
- axis = 1 => Sorts rows

[ ]:
```
# np.sort for sorting

arr1d = np.array([5, 2, 9, 1, 7])
sortedArr = np.sort(arr1d) # Ascending order
print(sortedArr)  # [1, 2, 5, 7, 9]  , indexes --> [3, 1, 0, 4, 2]

sortedDesc = np.sort(arr1d)[::-1]  # Descending order
print(sortedDesc)  # [9, 7, 5, 2, 1]


# 2D arrays

arr2D = np.array([[8, 4],
                  [3, 9],
                  [2, 7],
                  [5, 1]])


# Sort each col (axis = 0)

sortedCol = np.sort(arr2D, axis = 0)
print(sortedCol)

# Sort each row (axis = 1)
sortedRow = np.sort(arr2D, axis = 1)
print(sortedRow)

# indices that will sort array
index = np.argsort(arr1d)
print(index)
```

```
print(arr1d[index])
```

5 Fancy Indexing

Give values according to the index

[ ]:
```
import numpy as np

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

# Pick specific elements
indices = [0, 2, 4]
print(arr[indices])  # [10 30 50]

# even repeat indices
print(arr[[1, 1, 3]])  # [20 20 40]

arr2d = np.array([[11, 12, 13],
                  [21, 22, 23],
                  [31, 32, 33]])

# Select specific rows
print(arr2d[[0, 2]])
# [[11 12 13]
#  [31 32 33]]

# Select specific columns
print(arr2d[:, [0, 2]])

# Select specific (row, col) pairs
print(arr2d[[0, 2], [1, 0]])
# Picks (0,1) → 12 and (2,0) → 31 → [12 31]
```

6 Boolean Indexing

Uses Relational operations

[ ]:
```
# Example 1D array (make sure arr1d is defined first)
arr1d = np.array([1, 3, 5, 7, 9])
print(arr1d)  # Print the original 1D array

# Create an array with values from 0 to 29
arr = np.arange(30)
print(arr)

# Boolean indexing: select elements from arr1d that are <= 5
b = arr1d[arr1d <= 5]
```

```
print(b)  # Only values 1, 3, 5 will remain

# Boolean indexing: select elements from arr that are < 15
c = arr[arr < 15]
print(c)  # Output will be [ 0  1  2 ... 14]
```

7 Random Permutations & Shuffling

[ ]:
```
# Create an array with values from 0 to 29
arr = np.arange(30)
print(arr)

# Randomly permute (shuffle) the elements, returning a COPY
rp = np.random.permutation(arr)   # Original array remains unchanged
print(rp)  # random num from 0-29
print(arr)
# Output: Original order still [0 1 2 ... 29]

# Shuffle IN-PLACE (modifies the original array)
np.random.shuffle(arr)
print(arr)
# Output: Original array is now randomly arranged
```

8 Sorting Strings

[ ]:
```
arr = np.array(['abc', 'Howareyou', 'u786', '13er'])   # According to ASCII␣
 ↪table
arr.sort() # Sorts the strings
print(arr)  # output --> ['13er', 'HowareYou', 'abc', 'u786']
```

9 Stacking & Splitting

- Stacking means combining two arrays
- Splitting means breaking an array into two or more arrays

[ ]:
```
import numpy as np

a = np.array([[1, 2],
              [3, 4]])
b = np.array([[5, 6],
              [7, 8]])

# Stacking arrays
print(f"Vertical Stack: \n{np.vstack((a, b))}")     # Stack arrays vertically␣
 ↪(row-wise)
```

```
print(f"Horizontal Stack: \n{np.hstack((a, b))}")   # Stack arrays horizontally␣
 ↪(column-wise)
print(f"Depth Stack: \n{np.dstack((a, b))}")        # Stack along depth (adds␣
 ↪new 3rd axis)

# Splitting arrays
array = np.arange(30)
arr = np.split(array, 5)      # Split 1D array into 5 equal parts
print(arr)
print()

# Example 2D array for splitting
arr2D = np.arange(16).reshape(4, 4)
print(arr2D)

hSplit = np.hsplit(arr2D, 2)  # Split into 2 parts horizontally (by columns)
print(hSplit)

vSplit = np.vsplit(arr2D, 2)  # Split into 2 parts vertically (by rows)
print(vSplit)
```

10 Structured Arrays

[ ]:
```
# Define a structured data type: name (string up to 10 chars), age (int32),␣
 ↪height (float32)
dt = np.dtype([('name', 'U10'), ('age', 'i4'), ('height', 'f4')])

# Create a NumPy structured array with that data type
people = np.array([
    ('Talal', 18, 5.9),
    ('Ahmad', 19, 5.8),
    ('Faiqa', 17, 5.5)
], dtype=dt)

print(people)          # Full structured array
print(people['name'])  # Only the 'name' column
print(people['age'])   # Only the 'age' column
print(people['height'])# Only the 'height' column
```

11 Conclusion

In this notebook, we explored: - Creating and manipulating arrays - Indexing, slicing, and fancy indexing - Sorting, stacking, and splitting arrays - Random number generation & shuffling - Structured arrays

NumPy is the foundation for data science libraries like Pandas, Scikit-Learn and others, so mas-

tering it sets the stage for deeper analysis.

11.1 Next Steps

Now that NumPy fundamentals are complete, the journey continues with: - **Pandas** → DataFrames, data cleaning, and real-world datasets - **Matplotlib / Seaborn** → Visualizing patterns and trends - **Mini Projects** → Applying NumPy + Pandas to solve problems (CSV analysis, sports stats, expenses tracking) - Gradually moving toward **Machine Learning** with Scikit-learn

NumPy is the backbone — everything else will build on these concepts.

11.1.1 Author: Hafiz Muhammad Talal

Completed on: 11-Aug-2025 Email: muhammadtala20201@gmail.com
LinkedIn: linkedin.com/in/talalhafizmuhammad
GitHub: github.com/talalhafizmuhammad
