Numpy_Tutorial

June 7, 2025

0.1 Numpy_Basics by Isha

0.2 Importing NumPy

[11]:
```
import numpy as np  # Standard import
```

0.3 Creating NumPy Arrays

[20]:
```
# From list
arr1 = np.array([1, 2, 3])
print("Array from list:", arr1)

# 2D array
arr2 = np.array([[1, 2], [3, 4]])
print("2D Array:", arr2)

# Zeros, Ones, Full, Arange, Linspace
print("Zeros:", np.zeros((2, 3)))
print("Ones:", np.ones((2, 3)))
print("Full:", np.full((2, 3), 7))
print("Arange:", np.arange(0, 10, 2))
print("Linspace:", np.linspace(0, 1, 5))
```

```
Array from list: [1 2 3]
2D Array: [[1 2]
 [3 4]]
Zeros: [[0. 0. 0.]
 [0. 0. 0.]]
Ones: [[1. 1. 1.]
 [1. 1. 1.]]
Full: [[7 7 7]
 [7 7 7]]
Arange: [0 2 4 6 8]
Linspace: [0.   0.25 0.5  0.75 1.  ]
```

0.4 Array Attributes

[29]:
```python
# Create a 2D NumPy array (a matrix with 2 rows and 3 columns)
a = np.array([[1, 2, 3], [4, 5, 6]])

# Print the shape of the array
# Output: (2, 3) → 2 rows and 3 columns
print("Shape:", a.shape)

# Print the number of dimensions
# Output: 2 → because this is a 2D array (like a table)
print("Dimensions:", a.ndim)

# Print the data type of the elements in the array
# Output: dtype('int64') or similar → means each element is an integer
print("Data type:", a.dtype)

# Print the size (in bytes) of each element in the array
# For example, if dtype is int64, itemsize will be 8 (bytes)
print("Item size:", a.itemsize)
```

```
Shape: (2, 3)
Dimensions: 2
Data type: int32
Item size: 4
```

0.5 Indexing and Slicing

[30]:
```python
# Create a 2D NumPy array with 2 rows and 3 columns
a = np.array([[10, 20, 30],
              [40, 50, 60]])

# Access and print the element at row 0, column 1
# Row 0 → [10, 20, 30], so a[0, 1] = 20
print("Element at (0,1):", a[0, 1])

# Print the entire first row of the array
# a[0] gives [10, 20, 30]
print("First row:", a[0])

# Print the first column (all rows, column index 0)
# a[:, 0] → [10, 40]
print("First column:", a[:, 0])
```

```
Element at (0,1): 20
First row: [10 20 30]
First column: [10 40]
```

0.6 Mathematical Operations

[31]:
```
# Create two 1D NumPy arrays of equal length
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Element-wise addition: [1+4, 2+5, 3+6] → [5, 7, 9]
print("Addition:", a + b)

# Element-wise multiplication: [1*4, 2*5, 3*6] → [4, 10, 18]
print("Multiplication:", a * b)

# Apply sine function element-wise
# sin(1), sin(2), sin(3) (in radians)
print("Sine:", np.sin(a))

# Apply natural logarithm (base e) to each element
# log(1) = 0, log(2), log(3)
print("Log:", np.log(a))
```

```
Addition: [5 7 9]
Multiplication: [ 4 10 18]
Sine: [0.84147098 0.90929743 0.14112001]
Log: [0.         0.69314718 1.09861229]
```

0.7 Broadcasting

[32]:
```
# Create a 2D array with shape (3,1) - 3 rows and 1 column
a = np.array([[1],
              [2],
              [3]])

# Create a 1D array with 3 elements (shape: (3,))
b = np.array([10, 20, 30])

# NumPy automatically "broadcasts" b across each row of a
# Resulting in a 3x3 array where each row in 'a' is added to 'b':
# Row 1: [1+10, 1+20, 1+30] → [11, 21, 31]
# Row 2: [2+10, 2+20, 2+30] → [12, 22, 32]
# Row 3: [3+10, 3+20, 3+30] → [13, 23, 33]

print("Broadcasted Addition:\n", a + b)
```

```
Broadcasted Addition:
 [[11 21 31]
 [12 22 32]
 [13 23 33]]
```

0.8 Aggregate Functions

[33]:
```
# Create a 2D NumPy array (2 rows × 2 columns)
a = np.array([[1, 2],
              [3, 4]])

# Sum of all elements: 1 + 2 + 3 + 4 = 10
print("Sum:", np.sum(a))

# Maximum value in the entire array: max(1, 2, 3, 4) = 4
print("Max:", np.max(a))

# Mean (average) of all elements: (1 + 2 + 3 + 4) / 4 = 2.5
print("Mean:", np.mean(a))

# Standard deviation (how spread out the numbers are)
# Formula: sqrt(mean((x - mean)^2)) → for [1,2,3,4], std ≈ 1.118
print("Standard Deviation:", np.std(a))
```

```
Sum: 10
Max: 4
Mean: 2.5
Standard Deviation: 1.118033988749895
```

0.9 Reshaping and Flattening

[34]:
```
# Create a 1D NumPy array with 12 elements: [0, 1, 2, ..., 11]
a = np.arange(12)

# Reshape the 1D array into a 2D array with 3 rows and 4 columns
# New shape: (3, 4)
b = a.reshape((3, 4))
print("Reshaped Array:\n", b)

# Flatten the 2D array back into a 1D array
# It reads row-wise (C-style): [0, 1, 2, ..., 11]
print("Flattened:", b.flatten())
```

```
Reshaped Array:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Flattened: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

0.10 Stacking Arrays

[35]:
```
# Create a 2D NumPy array with shape (2, 2)
a = np.array([[1, 2],
              [3, 4]])

# Create a second 2D array with shape (1, 2)
b = np.array([[5, 6]])

# Vertically stack arrays a and b
# Stacking rows: a (2 rows) + b (1 row) → result will be (3, 2)
print("Vertical Stack:\n", np.vstack([a, b]))

# Horizontally stack array a with itself
# Stacking columns side by side → result will be (2, 4)
# [1, 2] + [1, 2] → [1, 2, 1, 2]
# [3, 4] + [3, 4] → [3, 4, 3, 4]
print("Horizontal Stack:\n", np.hstack([a, a]))
```

```
Vertical Stack:
 [[1 2]
 [3 4]
 [5 6]]
Horizontal Stack:
 [[1 2 1 2]
 [3 4 3 4]]
```

0.11 Data Type Conversion

[36]:
```
# Create a 1D NumPy array of floating-point numbers
a = np.array([1.1, 2.2, 3.3])

# Print the original data type of the array elements (float64)
print("Original dtype:", a.dtype)

# Convert the float array to integer using astype()
# This will truncate the decimal part (not round)
b = a.astype(int)

# Print the converted integer array: [1, 2, 3]
print("Converted to int:", b)
```

```
Original dtype: float64
Converted to int: [1 2 3]
```

0.12 Random Number Generation

[37]:
```
import numpy as np

# Generate a 2x3 array of random integers between 0 (inclusive) and 10␣
 ↪(exclusive)
print("Random Integer:", np.random.randint(0, 10, size=(2, 3)))

# Generate a 2x3 array of random floats between 0.0 and 1.0 (uniform␣
 ↪distribution)
print("Random Float:", np.random.rand(2, 3))

# Generate a 2x3 array of random numbers from a standard normal distribution␣
 ↪(mean=0, std=1)
print("Normal Distribution:", np.random.randn(2, 3))
```

```
Random Integer: [[8 5 9]
 [7 1 4]]
Random Float: [[0.32615647 0.69294871 0.87444168]
 [0.9649856  0.96579638 0.05145104]]
Normal Distribution: [[-0.33888374  0.08412282 -0.15218141]
 [-1.07526514  0.81816641 -1.13225337]]
```

0.13 Linear Algebra

[38]:
```
import numpy as np

# Define two 2x2 matrices
a = np.array([[1, 2],
              [3, 4]])

b = np.array([[2, 0],
              [1, 2]])

# Matrix multiplication (dot product)
# Multiply rows of 'a' with columns of 'b'
# Result is also a 2x2 matrix
print("Matrix Product:\n", np.dot(a, b))

# Transpose of matrix 'a'
# Rows become columns: [[1, 3], [2, 4]]
print("Transpose:\n", a.T)

# Inverse of matrix 'a'
# np.linalg.inv() computes the inverse if it's not singular
print("Inverse:\n", np.linalg.inv(a))
```

```
Matrix Product:
```

```
 [[ 4  4]
 [10  8]]
Transpose:
 [[1 3]
 [2 4]]
Inverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

0.14 File I/O

[39]:
```python
import numpy as np

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

# Save the array 'a' to a CSV file named 'array.csv'
# Each element will be written separated by a comma
np.savetxt('array.csv', a, delimiter=',')

# Load the array back from the saved CSV file
loaded = np.loadtxt('array.csv', delimiter=',')

# Display the loaded array to confirm it's the same as original
print("Loaded from CSV:\n", loaded)
```

```
Loaded from CSV:
 [[1. 2.]
 [3. 4.]]
```

0.15 Fancy Indexing and Boolean Masking

[3]:
```python
import numpy as np

# Create an array of integers from 0 to 9
a = np.arange(10)

# Fancy indexing: select elements at specific indices
selected = a[[2, 4, 6]]
print("Selected elements at indices 2, 4, 6:", selected)

# Boolean masking: select elements greater than 5
mask = a > 5
filtered = a[mask]
print("Elements greater than 5:", filtered)
```

```
Selected elements at indices 2, 4, 6: [2 4 6]
Elements greater than 5: [6 7 8 9]
```

0.16 Structured Arrays

[4]:
```
import numpy as np

# Define a structured data type with fields 'name' (string), 'age' (int), and␣
 ↪'weight' (float)
dtype = [('name', 'U10'), ('age', 'i4'), ('weight', 'f4')]

# Create an array of structured data
data = np.array([('Alice', 25, 55.0), ('Bob', 30, 85.5)], dtype=dtype)

# Access the 'name' field
names = data['name']
print("Names:", names)

# Access the 'age' field
ages = data['age']
print("Ages:", ages)
```

```
Names: ['Alice' 'Bob']
Ages: [25 30]
```

0.17 Masked Arrays

[5]:
```
import numpy as np

# Create an array with some invalid values
data = np.array([1.0, np.nan, 3.0, np.inf])

# Mask invalid (NaN or Inf) values
masked_data = np.ma.masked_invalid(data)

print("Masked array:", masked_data)
```

```
Masked array: [1.0 -- 3.0 --]
```

0.18 Broadcasting with np.newaxis

[6]:
```
import numpy as np

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

# Reshape 'a' to a column vector
a_column = a[:, np.newaxis]

# Create another 1D array
b = np.array([4, 5, 6])
```

```
# Add 'a_column' and 'b' using broadcasting
result = a_column + b

print("Broadcasted addition result:\n", result)
```

```
Broadcasted addition result:
 [[5 6 7]
 [6 7 8]
 [7 8 9]]
```

0.19 Advanced Linear Algebra Operations

[7]:
```
import numpy as np

# Create a 2x2 matrix
A = np.array([[1, 2], [3, 4]])

# Compute the determinant
det_A = np.linalg.det(A)
print("Determinant of A:", det_A)

# Compute the inverse
inv_A = np.linalg.inv(A)
print("Inverse of A:\n", inv_A)

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

```
Determinant of A: -2.0000000000000004
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
Eigenvalues: [-0.37228132  5.37228132]
Eigenvectors:
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

0.20 Random Number Generation

[8]:
```
import numpy as np

# Set a seed for reproducibility
np.random.seed(0)

# Generate a 2x3 array of random floats in [0.0, 1.0)
```

```
random_floats = np.random.rand(2, 3)
print("Random floats:\n", random_floats)

# Generate a 2x3 array of random integers between 0 and 10
random_ints = np.random.randint(0, 10, size=(2, 3))
print("Random integers:\n", random_ints)

# Generate a 2x3 array of samples from a standard normal distribution
random_normals = np.random.randn(2, 3)
print("Random normals:\n", random_normals)
```

```
Random floats:
 [[0.5488135  0.71518937 0.60276338]
 [0.54488318 0.4236548  0.64589411]]
Random integers:
 [[4 7 6]
 [8 8 1]]
Random normals:
 [[0.14404357 1.45427351 0.76103773]
 [0.12167502 0.44386323 0.33367433]]
```

0.21 Performance Optimization with Vectorization

[9]:
```
import numpy as np
import time

# Create a large array
a = np.arange(1e6)

# Vectorized operation: add 10 to each element
start_time = time.time()
a_plus_10 = a + 10
end_time = time.time()
print("Vectorized addition took:", end_time - start_time, "seconds")

# Equivalent operation using a Python loop (slower)
a_list = list(range(int(1e6)))
start_time = time.time()
a_plus_10_list = [x + 10 for x in a_list]
end_time = time.time()
print("Loop addition took:", end_time - start_time, "seconds")
```

```
Vectorized addition took: 0.002999544143676758 seconds
Loop addition took: 0.057985544204711914 seconds
```

0.22 Memory Mapping Large Files

[10]:
```
import numpy as np

# Create a large array and save it to a binary file
large_array = np.arange(1e7)
large_array.tofile('large_array.dat')

# Memory-map the binary file for reading
mapped_array = np.memmap('large_array.dat', dtype='float64', mode='r',␣
 ↪shape=(int(1e7),))

# Access elements without loading the entire file into memory
print("First 5 elements:", mapped_array[:5])
```

```
First 5 elements: [0. 1. 2. 3. 4.]
```