NumPy: The Fundamental Package for Scientific Computing in Python

Dr. Ahmed Awais

May 5, 2025

Introduction to NumPy

0.1 What is NumPy?

NumPy (**Numerical Python**) is the core library for numerical computing in Python. It provides:

- Efficient **multi-dimensional array (ndarray)** objects
- Mathematical functions for array operations (linear algebra, statistics, Fourier transforms)
- Tools for integrating with C/C++/Fortran code

Why We Should Use NumPy?: Motivation

- **Performance**: Operations are implemented in optimized C (approx. 100x faster than Python loops)
- **Memory Efficiency**: Arrays store data in contiguous memory blocks
- **Vectorization**: Eliminates need for explicit loops
- **Ecosystem**: Foundation for Pandas, SciPy, Scikit-learn, etc.

NumPy Arrays vs Python Lists

Table 1: Comparison of NumPy Arrays and Python Lists

Feature	**Python List**	**NumPy Array**
Storage	Heterogeneous	Homogeneous (fixed dtype)
Speed	Slow (interpreted)	Fast (C-optimized)
Memory	High (stores pointers)	Low (contiguous)
Operations	Basic	Advanced math (FFT, SVD) FAST Fourier Transform,Singular Value Decomposition
Vectorization	No	Yes

Practice Questions

- **Create two NumPy arrays representing matrices (e.g., 3x3 and 3x2). Perform matrix multiplication using np.dot() or the @ operator. Print the resulting matrix and its shape. Explain the conditions required for matrix multiplication (inner dimensions must match).**
- **Create a NumPy array of numerical data. Calculate the mean, median, standard deviation, and variance of the data using NumPy functions (np.mean(), np.median(), np.std(), np.var()). Print the results.**
- **Generate a NumPy array of random numbers using np.random.rand(). Create a histogram of the data using plt.hist(). Customize the histogram by setting the number of bins, color, and title.**
- **Create a 1D NumPy array. Reshape it into a 2D array with a specified number of rows and columns using np.reshape(). Print the original and reshaped arrays. Handle potential errors if the reshaping is not possible (e.g., the number of elements doesn't match).**
- **Create a NumPy array of numerical data. Use boolean indexing to create a new array containing only the elements that satisfy a certain condition (e.g., greater than 5, even numbers). Print the original and filtered arrays.**
- **Broadcasting: Create a NumPy array. Add a scalar value to each element of the array using broadcasting. Print the original and modified arrays. Explain how broadcasting works in this case.**
- **Create a NumPy array. Save it to a file using np.save() or np.savetxt(). Load the array from the file using np.load() or np.loadtxt(). Print the original and loaded arrays to verify that they are the same.**

```
import numpy as np

# Create two matrices
matrix_a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
matrix_b = np.array([[10, 11], [12, 13], [14, 15]])

# Perform matrix multiplication
result_matrix = np.dot(matrix_a, matrix_b)  # or matrix_a @ matrix_b

# Print the result
print("Matrix A:\n", matrix_a)
print("\nMatrix B:\n", matrix_b)
print("\nResultant Matrix:\n", result_matrix)
print("\nShape of Resultant Matrix:", result_matrix.shape)
```

```
import numpy as np

# Create a NumPy array
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Calculate statistics
mean = np.mean(data)
median = np.median(data)
std_dev = np.std(data)
variance = np.var(data)

# Print the results
print("Data:\n", data)
print("\nMean:", mean)
print("\nMedian:", median)
print("\nStandard Deviation:", std_dev)
print("\nVariance:", variance)
```

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Generate random data
data = np.random.rand(1000)  # 1000 random numbers between 0 and 1

# Create a histogram
plt.hist(data, bins=30, color='skyblue', edgecolor='black')
plt.title("Histogram of Random Data")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

```
import numpy as np

# Create a 1D array
data = np.arange(12)  # Array from 0 to 11

# Reshape the array
try:
    reshaped_data = data.reshape(3, 4)  # Reshape to 3x4
    print("Original Array:\n", data)
    print("\nReshaped Array:\n", reshaped_data)
except ValueError as e:
    print("Reshaping not possible:", e)
```

```
import numpy as np

# Create a NumPy array
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Filter the array (elements greater than 5)
filtered_data = data[data > 5]

# Print the results
print("Original Array:\n", data)
print("\nFiltered Array (elements > 5):\n", filtered_data)
```

```
import numpy as np

# Create a NumPy array
data = np.array([1, 2, 3, 4, 5])

# Add a scalar to the array (broadcasting)
modified_data = data + 10

# Print the results
print("Original Array:\n", data)
print("\nModified Array (after broadcasting):\n", modified_data)
```

```
import numpy as np

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

# Save the array to a file
np.savetxt("my_array.txt", data)  # or np.save("my_array.npy", data)

# Load the array from the file
loaded_data = np.loadtxt("my_array.txt")  # or np.load("my_array.npy")

# Print the results
print("Original Array:\n", data)
print("\nLoaded Array:\n", loaded_data)

# Verify that the arrays are the same
print("\nArrays are equal:", np.array_equal(data, loaded_data))
```

Questions to attempt

- **Recall last lecture and Create the NumPy array with the specified shape and values (5,3,2), can you create an image (the straight answer is NO, image needs 3 channels)**

```
import numpy as np

data = np.array(...)

# Print the array and its shape
print("NumPy Array:\n", data)
print("\nShape:", data.shape)
```

The data array is a 3-dimensional NumPy array with the following structure:

Dimension 0 (Blocks): It contains 5 "blocks" or "slices".

Dimension 1 (Rows): Each block is a 2D array with 3 rows.

Dimension 2 (Columns): Each row has 2 columns. Therefore, you can say:

"This array contains 5 blocks, where each block is a 3x2 array."

- **Repeat the previous one with (5,3,3), can you create an image (the straight answer is Yes, because it has now 3 channels), but how to access, let's start understanding**

NumPy Data Types (dtypes)

Let's discuss dtypes

Table 2: NumPy Data Types and Their Properties

Data Type (dtype)	Description	Value Range	Common Usage
`int8`	Signed 8-bit integer	-128 to 127	Small integer values, memory-constrained applications
`int16`	Signed 16-bit integer	-32,768 to 32,767	Moderate-sized integer values
`int32`	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647	General-purpose integer values
`int64`	Signed 64-bit integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Large integer values
`uint8`	Unsigned 8-bit integer	0 to 255	Image representation (pixel values), small positive integers, **We are going to use this**
`uint16`	Unsigned 16-bit integer	0 to 65,535	Moderate-sized positive integers
`uint32`	Unsigned 32-bit integer	0 to 4,294,967,295	General-purpose positive integers
`uint64`	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615	Large positive integers
`float16`	Half-precision float	-65,500 to 65,500 (approx.)	Reduced memory usage, less precision required
`float32`	Single-precision float	-3.4e38 to 3.4e38 (approx.)	General-purpose floating-point values
`float64`	Double-precision float	-1.7e308 to 1.7e308 (approx.)	High-precision floating-point values, scientific computing
`bool`	Boolean (True or False)	0 or 1	Logical operations, masking
`complex64`	Complex number (32-bit floats)	N/A	Scientific and engineering computations
`complex128`	Complex number (64-bit floats)	N/A	High-precision complex number computations

Basics of Image Processing

```
import numpy as np
import matplotlib.pyplot as plt

# Create a small 3x3 RGB image
img = np.zeros((3, 3, 3), dtype=np.uint8)  # Initialize with all zeros (black)

# --- Pixel Locations ---
# (row, column)
top_left     = (0, 0)
top_middle   = (0, 1)
```

```
top_right    = (0, 2)
middle_left  = (1, 0)
middle_middle= (1, 1)
middle_right = (1, 2)
bottom_left  = (2, 0)
bottom_middle= (2, 1)
bottom_right = (2, 2)

# --- Color Definitions ---
red   = [255, 0, 0]
green = [0, 255, 0]
blue  = [0, 0, 255]
white = [255, 255, 255]
black = [0, 0, 0]
magenta = [255, 0, 255]
cyan = [0, 255, 255]
yellow = [255, 255, 0]

# --- Manipulating Pixels ---

# First Row
img[top_left]     = red       # Top-left pixel to red
img[top_middle]   = green     # Top-middle pixel to green
img[top_right]    = blue      # Top-right pixel to blue

# Second Row
img[middle_left]  = yellow    # Middle-left pixel to yellow
img[middle_middle]= magenta   # Center pixel to magenta
img[middle_right] = cyan      # Middle-right pixel to cyan

# Third Row
img[bottom_left]  = white     # Bottom-left pixel to white
img[bottom_middle]= [128, 128, 128] # Gray
img[bottom_right] = black     # Bottom-right pixel to black

# --- Display the Image ---
plt.imshow(img)
plt.title("Manipulated 3x3 RGB Image")
plt.axis('off')  # Hide axes
plt.show()

# --- Explanation (just for understanding, no need to write it)---
print("Explanation:")
print("- `img[row, column] = [R, G, B]` sets the color of the pixel at the specified row and column.")
print("- `R`, `G`, and `B` are the red, green, and blue color components, ranging from 0 to 255.")
print("- The pixel locations (top_left, middle_middle, etc.) are defined as tuples (row, column) for clarity.")
print("- You can change the colors and pixel locations to experiment with different patterns.")
```

Question to attempt

- **Create an array with 4x4x3 shape and rearrange pixel locations**
- Can image have more than 3 channel? Search over internet and let's experience RGBA

```
import numpy as np
import matplotlib.pyplot as plt

# Create a small RGBA image (3x3 pixels)
rgba_image = np.zeros((3, 3, 4), dtype=np.uint8)

# Make the top-left pixel red and fully opaque
rgba_image[0, 0] = [255, 0, 0, 255]

# Make the center pixel green and 50% transparent
rgba_image[1, 1] = [0, 255, 0, 128]

# Display the image
plt.imshow(rgba_image)
plt.title("RGBA Image")
plt.axis('off')
plt.show()
```

Loading Real Images

dataset source: https://www.kaggle.com/datasets/k122215awaisahmed/kittydata

```
import numpy as np
import matplotlib.pyplot as plt
source_path = "/kaggle/input/kittydata/kitty.png"

kitty_img = plt.imread(source_path)

print(kitty_img.shape, kitty_img.size)

# Extract the red channel
red_channel = kitty_img[:, :, 0]

# Extract the green channel
green_channel = kitty_img[:, :, 1]

# Extract the blue channel
blue_channel = kitty_img[:, :, 2]
plt.figure(figsize=(12, 4))

plt.subplot(1, 4, 1)  # 1 row, 4 columns, first subplot
plt.imshow(kitty_img)  # Use 'gray' colormap to visualize single channel
plt.title('Input Image')
# plt.axis('off')

plt.subplot(1, 4, 2)  # 1 row, 4 columns, second subplot
plt.imshow(red_channel)  # Use 'gray' colormap to visualize single channel
plt.title('Red Channel')
# plt.axis('off')

plt.subplot(1, 4, 3)  # 1 row, 4 columns, third subplot
plt.imshow(green_channel)
plt.title('Green Channel')
# plt.axis('off')

plt.subplot(1, 4, 4)  # 1 row, 4 columns, fourth subplot
plt.imshow(blue_channel)
plt.title('Blue Channel')
# plt.axis('off')

plt.tight_layout()  # Adjusting layout to prevent overlapping titles
plt.show()
```