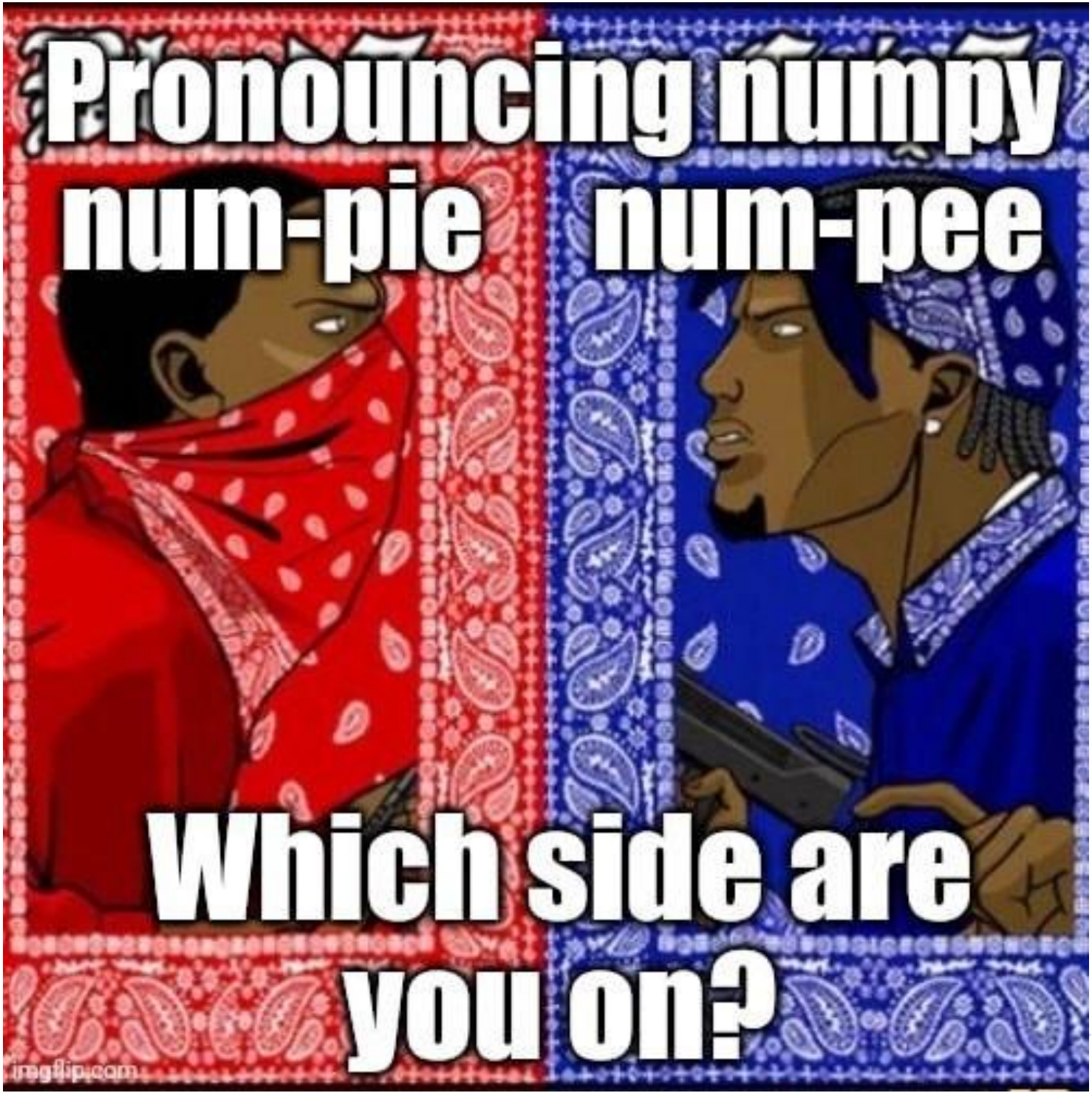
CS6.302 - Software System Development
Python Session - 4
NumPy, Matplotlib, Pandas

Chandrasekar S, Kunal Bhosikar

October 31, 2025

numpy


Pronouncing numpy
num-pie
num-pee
Which side are you on?
imgflip.com

Introduction to Numpy

- Fundamental package for scientific computing with Python.
- Features an N-dimensional array object.
- Provides tools for linear algebra, Fourier transforms, random number capabilities.
- Serves as a building block for other packages (e.g., SciPy, Pandas).
- Works well with plotting libraries (`matplotlib, seaborn, plotly etc.`).
- Open source.

Basics

```
1  import numpy as np
2
3  A = np.array([[1, 2, 3],
4                [4, 5, 6]])
5
6  print(A)
7  # Output:
8  # [[1 2 3]
9  #  [4 5 6]]
10
11 Af = np.array([1, 2, 3], dtype=float)
```

Array Creation and Initialization

```
1  np.arange(0, 1, 0.2)
2  # array([0. , 0.2, 0.4, 0.6, 0.8])
3
4  np.linspace(0, 2*np.pi, 4)
5  # array([0. , 2.09, 4.18, 6.28])
6
7  A = np.zeros((2, 3))
8  # array([[0., 0., 0.],
9  #        [0., 0., 0.]])
10 # np.ones, np.diag
11
12 A.shape
13 # (2, 3)
```

Numpy arrays are mutable

```
1  A = np.zeros((2, 2))
2  # array([[ 0.,  0.],
3  #        [ 0.,  0.]])
4
5  C = A
6  C[0, 0] = 1
7
8  print(A)
9  # [[ 1.  0.]
10 #  [ 0.  0.]]
```

You can prevent this behavior by using the `np.copy()` function to create an independent copy of the array.

Array Attributes

```
1 a = np.arange(10).reshape((2, 5))
2 print(a.ndim)     # 2 (dimensions)
3 print(a.shape)    # (2, 5)
4 print(a.size)     # 10 (number of elements)
5 print(a.T)        # Transpose
6 a.dtype           # Data type
```

Basic Operations

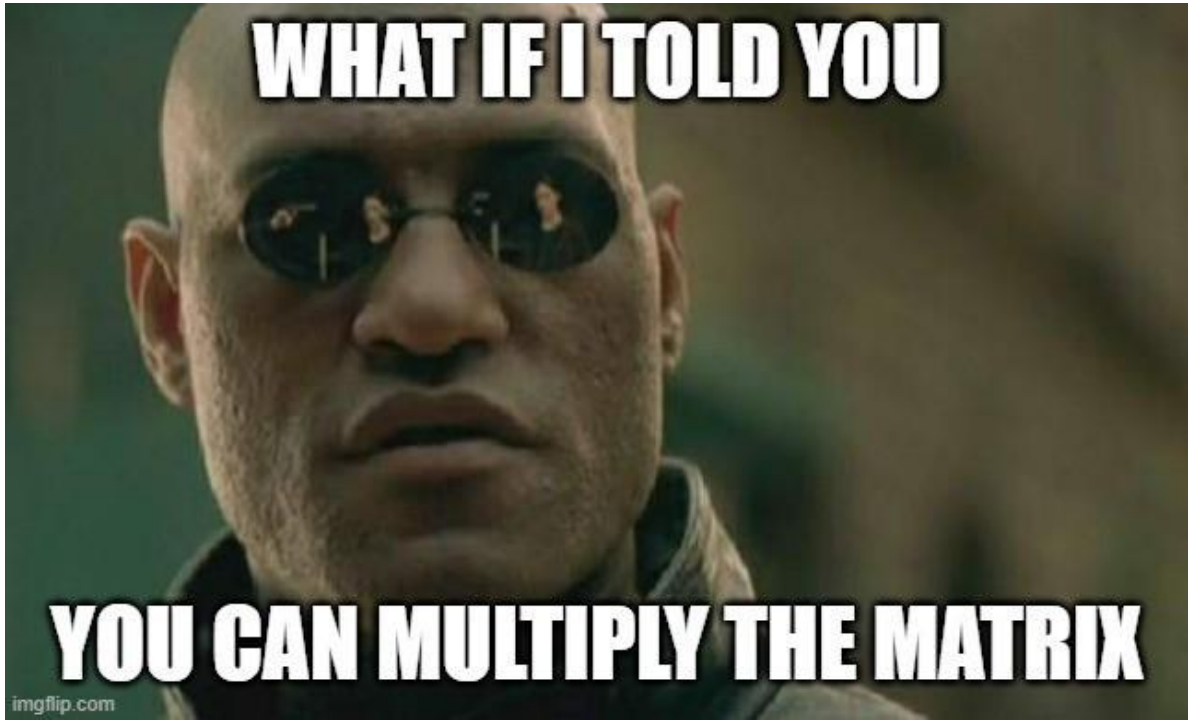
```
1  a = np.arange(4)   # array([0, 1, 2, 3])
2
3  b = np.array([2, 3, 2, 4])
4  print(a * b)       # array([0, 3, 4, 12])
5
6  print(b - a)       # array([2, 2, 0, 1])
7  c = [2, 3, 4, 5]
8  print(a * c)       # array([0, 3, 8, 15])
```

Vector Operations

```
1  # Example vectors
2  u = [1, 2, 3]
3  v = [1, 1, 1]
4
5  # Inner Product
6  np.inner(u, v) # Output: 6
7
8  # Outer Product
9  np.outer(u, v) # Output:
10 #                 array([[1, 1, 1],
11 #                        [2, 2, 2],
12 #                        [3, 3, 3]])
13
14 # Dot Product (Matrix Multiplication)
15 np.dot(u, v) # Output: 6
```

Matrix Operations

- `np.add(A, B)`: Adds matrices A and B element-wise.
- `np.subtract(A, B)`: Subtracts matrix B from matrix A element-wise.
- `np.multiply(A, B)`: Multiplies matrices A and B element-wise.
- `np.divide(A, B)`: Divides matrix A by matrix B element-wise.
- `np.dot(A, B)`: Computes the dot product of matrices A and B.


WHAT IF I TOLD YOU
YOU CAN MULTIPLY THE MATRIX
imgflip.com

Matrix Operations

- `np.transpose(A)`: Transposes matrix A (swaps rows and columns).
- `np.matmul(A, B)`: Performs matrix multiplication (supports 2D arrays). Performs the same functions as `np.dot`
- `np.identity(n)`: Creates an $n \times n$ identity matrix.
- `np.trace(A)`: Computes the trace of matrix A (sum of diagonal elements).

Matrix Operations

```
1  # Define matrices
2  A = np.ones((3, 2))
3  # array([[ 1., 1.],
4  #        [ 1., 1.],
5  #        [ 1., 1.]])
6  A.T
7  # array([[ 1., 1., 1.],
8  #        [ 1., 1., 1.]])
9
10 B = np.ones((2, 3))
11 # array([[ 1., 1., 1.],
12 #        [ 1., 1., 1.]])
```

Matrix Operations

```
1  # Matrix operations
2
3  np.dot(B.T, A.T)
4  # array([[ 2.,  2.,  2.],
5  #        [ 2.,  2.,  2.],
6  #        [ 2.,  2.,  2.]])
7
8  np.dot(A, B.T)
9  # Error: ValueError: shapes (3,2) and (3,2) not
       aligned: ...
10 # ... 2 (dim 1) != 3 (dim 0)
```

1-D Array Slicing Example

```
1  # Generate a 1-D array
2  a = np.array([0.25, 0.56, 0.98, 0.13, 0.72])
3
4  ## Select elements from index 2 to the end
5  a[2:] # array([ 0.98,  0.13,  0.72])
6
7  ## Select elements from index 1 to 4 (exclusive)
8  a[1:4] # array([ 0.56,  0.98,  0.13])
9
10 # Select every second element
11 a[::2] # array([ 0.25,  0.98,  0.72])
```

1-D Array Slicing Example

```
1  # Select elements in reverse order
2  a[::-1] # array([ 0.72,  0.13,  0.98,  0.56,  0.25])
3
4  # Select elements with a negative step from index 4 to
       0 (exclusive)
5  a[4:0:-1] # array([0.72,    0.13,  0.98,  0.56])
6
7  # Select the last element (negative indexing)
8  a[-1] # 0.72
9
10 # Select the second to last element (negative indexing
       )
11 a[-2]
12 # Output: 0.13
```

2-D Array Slicing

```
1  a = np.array([[ 0.25,  0.56,  0.98,  0.13,  0.72],
2                [ 0.43,  0.15,  0.67,  0.89,  0.24],
3                [ 0.91,  0.78,  0.64,  0.38,  0.55],
4                [ 0.19,  0.82,  0.13,  0.29,  0.71]])
5
6  # Select the third row, all columns
7  a[2, :]
8  # Output: array([ 0.91,  0.78,  0.64,  0.38,  0.55])
9
10 # Select the 2nd and 3rd rows, all colu mns
11 a[1:3]
12 # Output: array([[ 0.43,  0.15,  0.67,  0.89,  0.24],
13 #                [ 0.91,  0.78,  0.64,  0.38,  0.55]])
```

2-D Array Slicing

```
1  # Select rows 1 through 3 (exclusive of 3), columns 1
       through 4 (exclusive of 4)
2  a[1:3, 1:4]
3  # Output: array([[ 0.15,  0.67,  0.89],
4  #                [ 0.78,  0.64,  0.38]])
5  # Select rows in reverse order, every other column
6  a[::-1, ::2]
7  # Output: array([[ 0.19,  0.13,  0.71],
8  #                [ 0.91,  0.64,  0.55],
9  #                [ 0.43,  0.67,  0.24],
10 #                [ 0.25,  0.98,  0.72]])
11
12 # Select the last element from the last row (negative
       indexing)
13 a[-1, -1]
14 # Output: 0.71
```

Reshaping Arrays

```
1  # Create a 1D array of size 12
2  a = np.arange(12)
3  print("Original array:")
4  print(a)
5  # Original array:
6  # [ 0  1  2  3  4  5  6  7  8  9 10 11]
7
8  # Reshape the array to 3x4
9  reshaped = a.reshape(3, 4)
10 print("\nReshaped array (3x4):")
11 print(reshaped)
12
13 # Reshaped array (3x4):
14 # [[ 0  1  2  3]
15 #  [ 4  5  6  7]
16 #  [ 8  9 10 11]]
```

Rules of Reshaping

The total number of elements in the original array must equal the total number of elements in the reshaped array:

$$\text{Original Size} = \prod(\text{original dimensions}) = \prod(\text{new dimensions})$$

- Example 1: A 1D array with 12 elements can be reshaped into a 3×4 array because $3 \times 4 = 12$.
- Example 2: Attempting to reshape a 1D array with 12 elements into a 3×5 array raises an error because $3 \times 5 \neq 12$.

```
1  # Reshape Example
2  a = np.arange(12)
3
4  # Valid reshaping
5  reshaped = a.reshape(3, 4)
6
7  # Invalid reshaping (raises an error)
8  invalid = a.reshape(3, 5) # ValueError : cannot
   reshape array
```

Random Sampling with NumPy

- For random sampling we use `np.random` module.

Random Sampling Functions - 1

- `rand(d0, d1, ..., dn)`: Generates random values from a uniform distribution in the range [0, 1) with a specified shape. For example, `rand(2, 3)` will generate a 2x3 array of random values.
- `randn(d0, d1, ..., dn)`: Generates random values from a standard normal distribution (mean=0, variance=1) with the given shape. For example, `randn(2, 3)` will produce a 2x3 array with values from the standard normal distribution.

Random Sampling Functions - 2

- `randint(lo, hi, size)`: Generates random integers from the range [*lo*, *hi*), where `lo` is the lower bound (inclusive) and `hi` is the upper bound (exclusive). The `size` argument specifies the shape of the output array. For example, `randint(0, 10, (2, 3))` will produce a 2x3 array with integers between 0 and 9.
- `choice(a, size, repl, p)`: Randomly samples elements from array `a`. The `size` specifies the number of elements to sample. If `repl=True`, the same element can be selected multiple times (sampling with replacement), otherwise, each element can only be selected once (sampling without replacement). The `p` parameter specifies the probability distribution for the sampling.

Understanding Seeds in Random Sampling

- In random sampling, a **seed** is an initial value used by a pseudorandom number generator (PRNG) to produce a sequence of random numbers.
- The seed ensures that random sampling can be reproduced. If you use the same seed value, you will get the same random numbers each time, which is useful for debugging and reproducibility in experiments.

Setting a Seed in NumPy

`np.random.seed(seed)`: Sets the seed for the random number generator. Once the seed is set, all subsequent random number generation will be deterministic and based on that seed.

```
1  # Set the seed for reproducibility
2  np.random.seed(42)
3  print(np.random.rand(2,3))
4  # array([[0.37454012, 0.95071431, 0.73199394],
5  #        [0.59865848, 0.15601864, 0.15599452]])
6
7  # seed again
8  np.random.seed(42)
9  print(np.random.rand(2,3))
10 # array([[0.37454012, 0.95071431, 0.73199394],
11 #        [0.59865848, 0.15601864, 0.15599452]])
12
13 # Generates the same numbers every time after setting
       the seed
```

Introduction to Masking in NumPy

- Masking allows for filtering elements of an array based on specified conditions.
- The result is an array where only the elements satisfying the condition(s) are included.
- Masking uses Boolean expressions to create a mask (True/False values) that is applied to the array.
- Common logical operators used in masking:
  - & (AND) – for multiple conditions that must be true
  - | (OR) – for conditions where at least one must be true

Example 1: Using & (AND) and — (OR)

- Masking with AND condition: values greater than 2 AND less than 5

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

# Masking with AND condition
mask = (arr > 2) & (arr < 5)
masked_array = arr[mask]
print(masked_array)   # Output: [3 4]
```

- Masking with OR condition: values less than 2 OR greater than 5

```
mask = (arr < 2) | (arr > 5)
masked_array = arr[mask]
print(masked_array)   # Output: [1 6]
```

Example 2: Using & (AND) for Multiple Conditions

- Masking with AND condition: values greater than 2 AND even numbers

```
1 arr = np.array([1, 2, 3, 4, 5, 6])
2
3 # Masking with AND condition: greater than 2 AND
      even numbers
4 mask = (arr > 2) & (arr % 2 == 0)
5 masked_array = arr[mask]
6 print(masked_array)  # Output: [4 6]
```

Important NumPy Functions

- `np.concatenate()` : Joins two or more arrays along an existing axis.
- `np.mean()` : Computes the mean (average) of array elements.
- `np.median()` : Computes the median of array elements.
- `np.std()` : Computes the standard deviation of array elements.
- `np.unique()` : Finds the unique elements of an array.
- `np.split()` : Splits an array into multiple sub-arrays.
- `np.argmax()` : Returns the indices of the maximum values along an axis.
- `np.argmin()` : Returns the indices of the minimum values along an axis.


THAT LIST NEVER
ENDS

Important NumPy Functions

- `np.argsort()` : Returns the indices that would sort an array.
- `np.hstack()` : Stacks arrays in sequence horizontally (column-wise).
- `np.vstack()` : Stacks arrays in sequence vertically (row-wise).
- `np.repeat()` : Repeats elements of an array.
- `np.isnan()` : Tests element-wise for NaNs (Not a Number).
- `np.isin()` : Tests whether elements of an array are in another array.
- `np.newaxis` : Adds a new axis to an array, used for reshaping or increasing dimensions.


THE END IS NEAR
THIS WILL NEVER END
YOUR OPTIMISM DISGUSTS ME.
©HILARY B. PRICE 11-13-2016
rhymeswithorange.com
Dist. by King Features

It doesn"t end here...

NumPy"s submodules:

- `linalg`: Linear algebra functions, such as matrix decompositions, solving linear systems, eigenvalues/eigenvectors, and more.
- `random`: Provides a suite of functions for generating random numbers, including probability distributions and random sampling.
- `fft`: Fast Fourier Transform functions for signal processing, spectral analysis, and efficient computation of discrete Fourier transforms.
- and more...

https://numpy.org/doc/stable/reference/index.html

SIMD, Vectorization, and Broadcasting

Optimizing Performance in NumPy:

- SIMD (Single Instruction, Multiple Data): A technique for processing multiple data points with a single instruction, enabling parallel processing for better performance.
- Vectorization: Leveraging NumPy"s ability to perform operations on entire arrays without explicit loops, allowing for faster and more efficient computations.
- Broadcasting: A powerful feature that enables NumPy to perform operations on arrays of different shapes without needing to reshape them, promoting memory efficiency.

Explore these concepts to get and understanding of how NumPy works!

Summary

- Numpy provides a powerful toolkit for numerical computation.
- Offers support for arrays, linear algebra, Fourier transforms, and random sampling.
- Highly optimized and widely used in scientific computing.
- **Next Steps:** Try `SciPy` to dive deeper into scientific computing tools.

Questions?

matplotlib

Matplotlib is a powerful Python library used for creating static, animated, and interactive visualizations. It is widely used for data visualization in scientific computing.

Installation

To install Matplotlib, use the following command:

```
pip install matplotlib
```

Basic Plot

A simple line plot can be created as follows:

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

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

plt.plot(x, y, marker="o", linestyle="-",
color="b") plt.xlabel("X Axis")
plt.ylabel("Y Axis")
plt.title("Basic Line Plot")
plt.show()
```

Scatter Plot

A scatter plot can be created using:

```
x = np.random.rand(50)
y = np.random.rand(50)
colors = np.random.rand(50)

plt.scatter(x, y, c=colors, alpha=0.5, cmap="viridis")
plt.colorbar()
plt.title("Scatter Plot")
plt.show()
```

Scatter Plot and Line Plot on the Same Axes

To combine a scatter plot and a line plot in the same figure, you can use both `plot()` and `scatter()` functions:

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
x_scatter = np.random.rand(30) * 10
y_scatter = np.sin(x_scatter) + np.random.randn(30) *
    0.1

plt.plot(x, y, label="Line Plot", color="b")  # Line
    plot
plt.scatter(x_scatter, y_scatter, label="Scatter Plot"
    , color="r")  # Scatter plot

plt.xlabel("X Axis
") plt.ylabel("Y
Axis")
plt.title("Scatter Plot and Line Plot")
plt.legend()
plt.show()
```

Bar Plot

Creating a bar chart:

```
categories = ["A","B","C","D"]
values = [3, 7, 1, 8]

plt.bar(categories, values, color=["red","blue",
    "green","purple"])
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Bar Chart")
plt.show()
```

You can also use `plt.barh()` to create a horizontal chart.

Side by Side Bar Plot

```
categories = ["A", "B","C"]
values1 = [10, 20, 30]
values2 = [15, 25, 35]

barWidth = 0.25

r1 = np.arange(len(categories))   # Positions for the
    first set of bars
r2 = [x + barWidth for x in r1]   # Positions for the
    second set of bars

plt.bar(r1, values1, color="b", width=barWidth, label=
    "Series 1")
plt.bar(r2, values2, color="r", width=barWidth, label=
    "Series 2")
```

Side by Side Bar Plot (Contd.)

```
# Add labels and title
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Side by Side Bar Plot")
# Add custom x-axis tick labels
plt.xticks([r + barWidth / 2 for r in r1], categories)

# Add legend
plt.legend()

# Show the plot
plt.show()
```

Histogram

A histogram is useful for visualizing data distributions:

```
data = np.random.randn (1000)
plt. hist( data , bins =30 , color="blue", edgecolor="black",
alpha =0.7)
plt. xlabel("Value ")
plt. ylabel("Frequency ")
plt. title ("Histogram ")
plt. show ()
```

More plots...

Additionally, `matplotlib` supports other types of plots like:

- Pie Chart
- Heatmap
- Box Plot

Subplots and Figures

Matplotlib provides the `subplot` and `subplots` functions to create multiple plots within a single figure.

Using Subplot

The `subplot` function allows creating multiple plots by specifying the grid layout as `subplot(rows, cols, index)`.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

plt.subplot(2, 1, 1)
plt.plot([1, 2, 3], [4, 5, 6], "r")
plt.title("First Subplot")

plt.subplot(2, 1, 2)
plt.plot([1, 2, 3], [10, 20, 30], "b")
plt.title("Second Subplot")

plt.tight_layout()
plt.show()
```

Now let"s see some advanced topics and libraries you can dive into to enhance your data visualization skills.

`subplots()` in Matplotlib

Creating multiple plots within a single figure can be achieved with `subplot()` or `subplots()` functions. These allow you to display several visualizations side by side for better comparison.

- `subplot()` allows you to manually define grid layout (e.g., 2 rows and 2 columns) and select the plot"s position.
- `subplots()` provides a more flexible approach, returning a figure and an array of axes, ideal for complex layouts.

3D Plotting with Matplotlib

Matplotlib supports 3D plotting via the `mpl_toolkits.mplot3d` module, which is helpful for visualizing relationships in multivariate data.

- Create 3D scatter plots, surface plots, wireframes, etc.
- Ideal for datasets with more than two variables.

Seaborn Integration

Seaborn, built on top of Matplotlib, simplifies the creation of complex visualizations and offers a high-level interface with built-in themes and color palettes.

- Seaborn handles statistical plots like categorical plots and pair plots with minimal code.
- It integrates well with pandas dataframes, automatically handling data aggregation and statistical analysis.

Interactive Plotting with Plotly

Plotly is a library for creating interactive plots, making it ideal for dynamic data exploration in web applications or Jupyter notebooks.

- Supports various charts, including 3D plots, heatmaps, and geographic maps.
- Features like zooming, hovering, and clicking on data points for detailed information enhance data interactivity.

Questions?

pandas

What is Pandas?

Pandas is a powerful Python library for data manipulation and analysis.

- Provides data structures like Series and DataFrame.
- Built on top of NumPy.
- Ideal for data cleaning, transformation, and analysis.

Installing Pandas

To install Pandas, use the following command:

```
pip install pandas
```

Importing Pandas

To use Pandas in Python, import it as follows:

```
import pandas as pd
```

Pandas Series

A Series is a one-dimensional labeled array.

```
s = pd.Series([10, 20, 30, 40])
print(s)
# Output:
# 0    10
# 1    20
# 2    30
# 3    40
# dtype: int64
```

Pandas DataFrame

A DataFrame is a two-dimensional labeled data structure.

```
data = {
    "Name": ["Alice", "Bob",
    "Charlie"], "Age": [25, 30, 35],
    "City": ["New York", "Paris", "London"]
}
df = pd.DataFrame(data)
print(df)
# Output:
#       Name   Age         City
# 0    Alice    25    New York
# 1      Bob    30       Paris
# 2  Charlie     35     London
```

Viewing Data

```
print(df.head())   # First 5 rows
print(df.tail())   # Last 5 rows

print(df.head(2)) # First 2 rows
print(df.head(2)) # Last 2 rows
```

Selecting Columns

```
print(df["Name"])   # Selecting a column

# Output:
# 0     Alice
# 1       Bob
# 2   Charlie
# Name: Name, dtype: object
```

Filtering Data

```
print(df[df["Age"] > 25])  # Filtering rows

# Output:
#       Name   Age     City
# 1      Bob    30    Paris
# 2  Charlie     35   London
```

This is similar to masking in NumPy

Adding a Column

```
df["Salary"] = [50000, 60000, 70000]
print(df)

# Output:
#       Name   Age         City   Salary
# 0   Alice     25    New York   50000
# 1      Bob     30       Paris   60000
# 2 Charlie      35     London   70000
```

GroupBy and Aggregation

```
print(df.groupby("City")[["Age", "Salary"]].mean())

# Output:
#                Age     Salary
# City
# London        35.0    70000.0
# New York      25.0    50000.0
# Paris         30.0    60000.0
```

- `df.groupby("City")` groups the data by unique values in the `City` column.
- `[["Age", "Salary"]]` selects the `Age` and `Salary` columns for aggregation.
- `.mean()` computes the average of `Age` and `Salary` for each city.
- Output shows the mean `Age` and `Salary` for each city:

Merging DataFrames

```
df1 = pd.DataFrame({"ID": [1, 2, 3], "Name": ["Alice
    ", "Bob", "Charlie"]})
df2 = pd.DataFrame({"ID": [1, 2, 3], "Salary": [50000,
    60000, 70000]})

merged_df = pd.merge(df1, df2, on="ID")
print(merged_df)

# Output:
#    ID      Name   Salary
# 0   1     Alice    50000
# 1   2       Bob    60000
# 2   3   Charlie    70000
```

Viewing DataFrame Structure

Understanding the structure of a DataFrame is crucial.

```
data = {"Name": ["Alice", "Bob", "Charlie
        "], "Age": [25, 30, 35],
        "City": ["New York", "Paris", "London
        "], "Salary": [50000, 60000, 70000]}
df = pd.DataFrame(data)
df.info()  # Summary of the DataFrame
# Output:
"""
<class "pandas.core.frame.DataFrame">
RangeIndex: 3 entries, 0 to 2
Data columns (total 4 columns):
 #    Column   Non-Null Count   Dtype
---   ------   --------------   -----
 0    Name     3 non-null       object
 1    Age      3 non-null       int64
 2    City     3 non-null       object
 3    Salary   3 non-null       int64
dtypes: int64(2), object(2)
memory usage: 224.0+ bytes"""
```

Viewing DataFrame Structure

```
df.describe()   # Statistical summary

# Output:
"""
            Age     Salary
count       3.0        3.0
mean       30.0    60000.0
std         5.0    10000.0
min        25.0    50000.0
25%        27.5    55000.0
50%        30.0    60000.0
75%        32.5    65000.0
max        35.0    70000.0
"""
```

Handling Missing Values

Missing data is common in datasets. Pandas provides functions to handle them.

```
# Creating a sample DataFrame with missing values
data = {"A": [1, 2, None, 4], "B": [None, 2, 3, 4],
"C
    ": [1, None, None, 4]}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

#       A     B     C
# 0   1.0   NaN   1.0
# 1   2.0   2.0   NaN
# 2   NaN   3.0   NaN
# 3   4.0   4.0   4.0
```

Handling Missing Values

```
# Using dropna() to remove rows with missing values
df_dropped = df.dropna()
print("\nDataFrame after dropna():")
print(df_dropped)
#  DataFrame after dropna():
#      A    B    C
# 3  4.0  4.0  4.0

# Using fillna() to replace missing values with 0
df_filled = df.fillna(0)
print("\nDataFrame after fillna(0):")
print(df_filled)

#  DataFrame after fillna(0):
#      A    B    C
# 0  1.0  0.0  1.0
# 1  2.0  2.0  0.0
# 2  0.0  3.0  0.0
# 3  4.0  4.0  4.0
```

Sorting Data

Sorting is useful for organizing data.

```
data = {"Name": ["Bob", "Charlie", "Alice
        "], "Age": [30, 35, 25],
        "City": ["Paris", "London", "New York
        "], "Salary": [60000, 70000, 50000]}
df = pd.DataFrame(data)

df.sort_values(by="Age")  # Sort DataFrame by Age
    column
# Output:
#       Name  Age      City  Salary
# 2    Alice   25  New York   50000
# 0      Bob   30     Paris   60000
# 1  Charlie   35    London   70000
```

Selecting Specific Rows and Columns

There are multiple ways to access specific data points.

```
data = {"Name": ["Alice", "Bob", "Charlie", "David"],
        "Age": [25, 30, 35, 40],
        "City": ["New York", "Los Angeles", "Chicago",
            "Houston"]}
df = pd.DataFrame(data)
print(df)
"""      Name   Age           City
0      Alice    25       New York
1        Bob    30    Los Angeles
2    Charlie    35        Chicago
3      David    40        Houston"""
# Using .loc[]
print(df.loc[0])          # Select first row by label
# Output:
# Name          Alice
# Age              25
# City       New York
# Name: 0, dtype: object
```

Selecting Specific Rows and Columns

```
print(df.loc[1, "City"])   # Select Bob"s city
# Output: Los Angeles

# Using .iloc[]
print(df.iloc[0, 1])    # First row, second column (Age
    of Alice)
# Output: 25

print(df.iloc[2, :])    # Select third row (Charlie)
# Output:
"""
Name      Charlie
Age            35
City      Chicago
Name: 2, dtype: object
"""
```

Applying Functions in Pandas

Transforming DataFrame columns using `apply()` allows for flexible data manipulation.

```
data = {"Name": ["Alice", "Bob", "Charlie", "David
        "], "Age": [25, 30, 35, 40],
        "Salary": [50000, 60000, 55000, 70000]}
df = pd.DataFrame(data)

# Example 1: Incrementing all ages by 5
df["Age"] = df["Age"].apply(lambda x: x + 5)

print("After incrementing ages by 5:")
print(df, end="\n\n\n")
"""
After incrementing ages by 5:
      Name   Age   Salary
0    Alice    30    50000
1      Bob    35    60000
2  Charlie    40    55000
3    David    45    70000
"""
```

Applying Functions in Pandas

```
# Example 2: Categorizing salary levels
df["Salary_Level"] = df["Salary"].apply(lambda x:
"


print("\nAfter categorizing salary levels:")
print(df, end="\n\n\n")
"""
After categorizing salary levels:
      Name   Age   Salary  Salary_Level
0    Alice    30    50000           Low
1      Bob    35    60000           Low
2  Charlie    40    55000           Low
3    David    45    70000          High
"""
```

Applying Functions in Pandas

```
# Example 3: Applying a complex function to format
    names
def format_name(name):
    return name.upper() if len(name) > 4 else name.
        lower()

df["Formatted_Name"] = df["Name"].apply(format_name)

print("\nAfter formatting names:")
print(df, end="\n\n\n")
"""
After formatting names:
      Name   Age   Salary  Salary_Level  Formatted_Name
0    Alice    30    50000           Low           ALICE
1      Bob    35    60000           Low             bob
2  Charlie    40    55000           Low         CHARLIE
3    David    45    70000          High           DAVID


"""
```

Reading and Writing Files in Pandas

Pandas makes it easy to read and write data from various file formats.

Reading Files:

```
# Reading a CSV file
df_csv = pd.read_csv("data.csv")

# Reading an Excel file
df_excel = pd.read_excel("data.xlsx", sheet_name
    ="Sheet1")

# Reading a JSON file
df_json = pd.read_json("data.json")
```

Reading and Writing Files in Pandas

Writing Files:

```
# Writing to a CSV file
df_csv.to_csv("output.csv", index=False)

# Writing to an Excel file
df_excel.to_excel("output.xlsx", sheet_name="Results",
    index=False)

# Writing to a JSON file
df_json.to_json("output.json", orient="records")
```

Use Cases:

- Importing large datasets for analysis.
- Saving processed data for future use.
- Converting data between different formats.

Resetting Index

Resetting the index is useful after filtering, sorting, or merging data.

```
df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie"
    ], "Age": [25, 30, 35]})
filtered_df = df[df["Age"] > 25]
print(filtered_df)   # Index remains [1, 2]

filtered_df.reset_index(drop=True, inplace=True)
print(filtered_df)   # Index is now reset
```

Why use it?

- Removes gaps in index after filtering.
- Useful after sorting or merging data.
- Ensures a clean, sequential index.

Next Steps

To further improve your Pandas skills, explore:

- For Practice Notebooks click here.
- Advanced indexing, Summary Functions and Conditional Selection.
- Working with large datasets and performance optimization.

Questions?


MACHINE LEARNING
Why is it, when something happens, it is always you three?
Numpy
Pandas
Matplotlib