# Introduction to Data Science

# Summary

- NumPy
- Pandas
- Matplotlib
- CSV Handling

# Why Python for Data Science?

- Simple and readable syntax
- Huge ecosystem of libraries (e.g., Pandas, NumPy, Scikit-learn, TensorFlow)
- Used widely in the industry and academia
- Ease of learning

levi nine

# NumPy

# What is NumPy?

- Stands for Numerical Python
- Core library for numerical computing in Python
- Efficient handling of large arrays and matrices
- Supports vectorized operations (faster than loops)
- Foundation for libraries like Pandas, SciPy, Scikit-learn

# NumPy Basics

- Array creation

```
import numpy as np

# Creating arrays
a = np.array([1, 2, 3])       # [1, 2, 3]
b = np.arange(0, 10, 2)       # [0, 2, 4, 6, 8]
c = np.zeros((2, 3))          # [[0. 0. 0.] [0. 0. 0.]]
d = np.ones((2, 3))           # [[1. 1. 1.] [1. 1. 1.]]
```

# NumPy Basics

- 1 dimension array

```
import numpy as np

# Create a 1D array
arr = np.array([1, 2, 3, 4])
print(arr * 2)          # Vectorized operation: [2 4 6 8]
print(arr + 5)              # [6 7 8 9]
```

# NumPy Basics

- 2 dimensions array

```
import numpy as np

# Create a 2D array
arr = np.array([[1, 2], [3, 4]])

arr.shape       # (2, 2)
np.sum(arr)     # 10
arr.reshape(4) # [1 2 3 4]
```

# NumPy Basics

- Basic math functions

```
import numpy as np

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

print("Mean:", np.mean(arr))                  # 3.0
print("Sum:", np.sum(arr))                    # 15
print("Standard Deviation:", np.std(arr))     # ~1.41
print("Max:", np.max(arr))                    # 5
print("Min:", np.min(arr))                    # 1
print("Cumulative Sum:", np.cumsum(arr))      # [ 1  3  6 10 15]
print("Square Root:", np.sqrt(arr))           # [1. 1.41 1.73 2. 2.24]
```

# NumPy Arrays vs Lists

- With Python lists, loops are often needed  to perform operations. NumPy arrays allow vectorized math, meaning operations are applied to the whole array at once, faster and more concise.

```
# Python list
lst = [1, 2, 3, 4]
doubled_lst = [x * 2 for x in lst]  # [2, 4, 6, 8]


# NumPy array
import numpy as np
arr = np.array([1, 2, 3, 4])
doubled_arr = arr * 2               # [2 4 6 8]
```

# NumPy Arrays

- All elements in the array must be of the same type
- NumPy is optimized for numerical computation
- With strings, the entire array will be of type str
- For mixed types NumPy will upcast all values to a common type, often to object, which removes performance benefits

# Pandas

# What is Pandas?

- Pandas it is a High-level data manipulation library built on NumPy
- Ideal for handling structured data
- Makes data cleaning, filtering, and transformation easy
- Integrates well with CSV, Excel, SQL, and more

# Series

- Series are One-dimensional labeled array (1D)
- Can hold any data type (integers, strings, floats, etc.)
- Similar to a column in Excel or a NumPy array with labels
- Index gives each value a unique label
- Supports vectorized operations (like NumPy)

# Series

```
import pandas as pd

data = [10, 20, 30, 40]
labels = ['a', 'b', 'c', 'd']

series = pd.Series(data, index=labels)
print(series)
```

| | 0 |
|---|---|
| a | 10 |
| b | 20 |
| c | 30 |

dtype: int64

# Series Basic Operations

- Indexing and slicing

```
series['b']     # Access by label – 20

series[1:3]     # Slicing by position
```

| | 0 |
|---|---|
| b | 20 |
| c | 30 |

dtype: int64

# Series Basic Operations

- Arithmetic operations

```
import pandas as pd

s1 = pd.Series([1, 2, 3])
s2 = pd.Series([4, 5, 6])

print(s1 + s2)
```

| | 0 |
|---|---|
| 0 | 5 |
| 1 | 7 |
| 2 | 9 |

dtype: int64

# Series Basic Operations

- String operations

```
import pandas as pd

s = pd.Series(["hello world", "pandas library", "data science", "AI"])

s.str.count("a")  s.str.get(0)  s.str.replace("a", "@")  s.str.title()  s.str.len()
```

| | 0 |
|---|---|
| 0 | 0 |
| 1 | 3 |
| 2 | 2 |
| 3 | 0 |

**dtype:** int64

| | 0 |
|---|---|
| 0 | h |
| 1 | p |
| 2 | d |
| 3 | A |

**dtype:** object

| | 0 |
|---|---|
| 0 | hello world |
| 1 | p@nd@s libr@ry |
| 2 | d@t@ science |
| 3 | AI |

**dtype:** object

| | 0 |
|---|---|
| 0 | Hello World |
| 1 | Pandas Library |
| 2 | Data Science |
| 3 | Ai |

**dtype:** object

| | 0 |
|---|---|
| 0 | 11 |
| 1 | 14 |
| 2 | 12 |
| 3 | 2 |

**dtype:** int64

# DataFrame

- A DataFrame is Two-dimensional table of data
- Columns are like Series objects
- Each row and column has labels (index and headers)
- Flexible and powerful for filtering, joining, and reshaping data
- Often created from dictionaries, CSVs, or databases

# DataFrame

- Creating a DataFrame from dict

```
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [24, 30, 35],
    "City": ["NY", "LA", "Chicago"]
}

df = pd.DataFrame(data)
print(df)
```

| | Name | Age | City |
|---|---|---|---|
| 0 | Alice | 24 | NY |
| 1 | Bob | 30 | LA |
| 2 | Charlie | 35 | Chicago |

# DataFrame

- Creating a DataFrame from list

```
df = pd.DataFrame([["Alice", 24, "NY"],
                   ["Bob", 30, "LA"],
                   ["Charlie", 35, "Chicago"]],
                  index = [0, 1, 2],
                  columns = ["Name", "Age", "City"])

print(df)
```

| | Name | Age | City |
|---|---|---|---|
| 0 | Alice | 24 | NY |
| 1 | Bob | 30 | LA |
| 2 | Charlie | 35 | Chicago |

# DataFrame Basic Operations

- Accessing data

```
df["Name"]          # Access a column
```

| | Name |
|---|---|
| 0 | Alice |
| 1 | Bob |
| 2 | Charlie |

**dtype:** object

# DataFrame Basic Operations

- Accessing data

```
# Access row by label
df.loc[0, "Name"]          # Alice


# Access row by index
df.iloc[0, 0]              # Alice
```

# DataFrame Basic Operations

- Adding a column

```
df['Salary'] = [50000, 60000, 50000]
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| 0 | Alice | 24 | NY | 50000 |
| 1 | Bob | 30 | LA | 60000 |
| 2 | Charlie | 35 | Chicago | 50000 |

```
df.columns  # Index(['Name', 'Age', 'City', 'Salary'], dtype='object')
```

# DataFrame Basic Operations

- Rename a column

```
df.rename(columns={"Name": "Surname"})
```

| | Surname | Age | City | Salary |
|---|---|---|---|---|
| **0** | Alice | 24 | NY | 50000 |
| **1** | Bob | 30 | LA | 60000 |
| **2** | Charlie | 35 | Chicago | 50000 |

# DataFrame Basic Operations

- Remove a column

```
df.drop("Age", axis=1, inplace=True)
```

| | Name | City | Salary |
|---|---|---|---|
| 0 | Alice | NY | 50000 |
| 1 | Bob | LA | 60000 |
| 2 | Charlie | Chicago | 50000 |

# DataFrame Basic Operations

- Remove duplicates

```
df.drop_duplicates("Salary", inplace=True)
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| 0 | Alice | 24 | NY | 50000 |
| 1 | Bob | 30 | LA | 60000 |

# DataFrame Basic Operations

- Filtering

```
df[df["Age"] > 25]
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| 1 | Bob | 30 | LA | 60000 |
| 2 | Charlie | 35 | Chicago | 70000 |

```
df.query("Age > 25")
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| 1 | Bob | 30 | LA | 60000 |
| 2 | Charlie | 35 | Chicago | 70000 |

# DataFrame Basic Operations

- Sorting

```
df.sort_values(by="City")
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| **2** | Charlie | 35 | Chicago | 50000 |
| **1** | Bob | 30 | LA | 60000 |
| **0** | Alice | 24 | NY | 50000 |

```
df.sort_values(by="City", ascending=False)
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| **0** | Alice | 24 | NY | 50000 |
| **1** | Bob | 30 | LA | 60000 |
| **2** | Charlie | 35 | Chicago | 50000 |

# DataFrame Basic Operations

- Sorting by index

```
df.sort_index()
```

| | Name | Age | City | Salary |
|---|---|---|---|---|
| **0** | Alice | 24 | NY | 50000 |
| **1** | Bob | 30 | LA | 60000 |
| **2** | Charlie | 35 | Chicago | 50000 |

# DataFrame Basic Operations

- Reseting index

```
df = pd.DataFrame([["Cat", 12], [ "Dog", 31], ["Duck", 5]],
                  index = [3, 4, 5],
                  columns = ["Pet", "Quantity"])
```

| | Pet | Quantity |
|---|---|---|
| 3 | Cat | 12 |
| 4 | Dog | 31 |
| 5 | Duck | 5 |

```
df.reset_index()
```

| | index | Pet | Quantity |
|---|---|---|---|
| 0 | 3 | Cat | 12 |
| 1 | 4 | Dog | 31 |
| 2 | 5 | Duck | 5 |

```
df.reset_index(drop=True)
```

| | Pet | Quantity |
|---|---|---|
| 0 | Cat | 12 |
| 1 | Dog | 31 |
| 2 | Duck | 5 |

# DataFrame Basic Operations

- Working with nulls

```
df = pd.DataFrame([["A", 1], [ "B", None], ["C", None]],
                  columns = ["Name", "Number"])
```

| | Name | Number |
|---|---|---|
| 0 | A | 1.0 |
| 1 | B | NaN |
| 2 | C | NaN |

```
df.dropna()
```

| | Name | Number |
|---|---|---|
| 0 | A | 1.0 |

```
df.fillna(0)
```

| | Name | Number |
|---|---|---|
| 0 | A | 1.0 |
| 1 | B | 0.0 |
| 2 | C | 0.0 |

# DataFrame Basic Operations

- Grouping it's used for aggregating data together with other aggregation functions like sum, mean, min, max, count, etc.

```
df.groupby("Salary").count()
```

| Salary | Name | Age | City |
|---|---|---|---|
| 50000 | 2 | 2 | 2 |
| 60000 | 1 | 1 | 1 |

# DataFrame Basic Operations

- There can be made custom aggregation with agg() or with lambdas

```
df.groupby('Salary')['Age'].agg(['mean', 'max', 'count'])
```

| Salary | mean | max | count |
|---|---|---|---|
| 50000 | 29.5 | 35 | 2 |
| 60000 | 30.0 | 30 | 1 |

```
df.groupby('Salary')['Age'].agg(lambda x: x.max() - x.min())
```

| Salary | Age |
|---|---|
| 50000 | 11 |
| 60000 | 0 |

dtype: int64

# DataFrame Basic Operations

Aggregation it's use with transformation functions:

- transform()  Apply a function to each group element
- fillna()  Fill missing values
- rank()  Ranks within groups
- apply()  General-purpose function application

```
df['Norm'] = df.groupby('Salary')['Age'].transform(lambda x: x / x.sum())
```

| | Name | Age | City | Salary | Norm |
|---|---|---|---|---|---|
| 0 | Alice | 24 | NY | 50000 | 0.40678 |
| 1 | Bob | 30 | LA | 60000 | 1.00000 |
| 2 | Charlie | 35 | Chicago | 50000 | 0.59322 |

# DataFrame Basic Operations

- More details

`df.describe()`

| | Age | Salary | Norm |
|---|---|---|---|
| **count** | 3.000000 | 3.000000 | 3.000000 |
| **mean** | 29.666667 | 53333.333333 | 0.666667 |
| **std** | 5.507571 | 5773.502692 | 0.303354 |
| **min** | 24.000000 | 50000.000000 | 0.406780 |
| **25%** | 27.000000 | 50000.000000 | 0.500000 |
| **50%** | 30.000000 | 50000.000000 | 0.593220 |
| **75%** | 32.500000 | 55000.000000 | 0.796610 |
| **max** | 35.000000 | 60000.000000 | 1.000000 |

`df.info()`

```
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, 0 to 2
Data columns (total 5 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
 4   Norm    3 non-null      float64
dtypes: float64(1), int64(2), object(2)
memory usage: 144.0+ bytes
None
```

# Working with multiple DataFrames

- Appending rows

```
df1 = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [24, 30]
})

df2 = pd.DataFrame({
    "Name": ["Charlie", "Diana"],
    "Age": [35, 28]
})


pd.concat([df1, df2], ignore_index=True)
```

| | Name | Age |
|---|---|---|
| 0 | Alice | 24 |
| 1 | Bob | 30 |

| | Name | Age |
|---|---|---|
| 0 | Charlie | 35 |
| 1 | Diana | 28 |

| | Name | Age |
|---|---|---|
| 0 | Alice | 24 |
| 1 | Bob | 30 |
| 2 | Charlie | 35 |
| 3 | Diana | 28 |

# Working with multiple DataFrames

- Appending columns

```
df1 = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [24, 30]
})

df3 = pd.DataFrame({"Score": [88, 92]})


pd.concat([df1, df3], axis=1)
```

| | Name | Age | Score |
|---|---|---|---|
| 0 | Alice | 24 | 88 |
| 1 | Bob | 30 | 92 |

# Working with multiple DataFrames

- Combining based on common columns (like SQL)

```
df_left = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [24, 30, 35]
})

df_right = pd.DataFrame({
    "Name": ["Alice", "Bob", "Diana"],
    "Score": [88, 92, 75]
})
```

| | Name | Age |
|---|---|---|
| 0 | Alice | 24 |
| 1 | Bob | 30 |
| 2 | Charlie | 35 |

| | Name | Score |
|---|---|---|
| 0 | Alice | 88 |
| 1 | Bob | 92 |
| 2 | Diana | 75 |

```
pd.merge(df_left, df_right, on="Name", how="inner")
```

| | Name | Age | Score |
|---|---|---|---|
| 0 | Alice | 24 | 88 |
| 1 | Bob | 30 | 92 |

# Working with multiple DataFrames

- Other types of merge

```
pd.merge(df_left, df_right, on="Name", how="left")
```

| | Name | Age | Score |
|---|---|---|---|
| 0 | Alice | 24 | 88.0 |
| 1 | Bob | 30 | 92.0 |
| 2 | Charlie | 35 | NaN |

# Working with multiple DataFrames

- Other types of merge

```
pd.merge(df_left, df_right, on="Name", how="right")
```

| | Name | Age | Score |
|---|---|---|---|
| 0 | Alice | 24.0 | 88 |
| 1 | Bob | 30.0 | 92 |
| 2 | Diana | NaN | 75 |

# Working with multiple DataFrames

- Other types of merge

```
pd.merge(df_left, df_right, on="Name", how="outer")
```

| | Name | Age | Score |
|---|---|---|---|
| 0 | Alice | 24.0 | 88.0 |
| 1 | Bob | 30.0 | 92.0 |
| 2 | Charlie | 35.0 | NaN |
| 3 | Diana | NaN | 75.0 |

# More operations

## Pandas Cheat Sheet

levi nine

# Matplotlib

# What is Matplotlib?

Matplotlib is a powerful Python library used to create static, interactive, and animated visualizations.

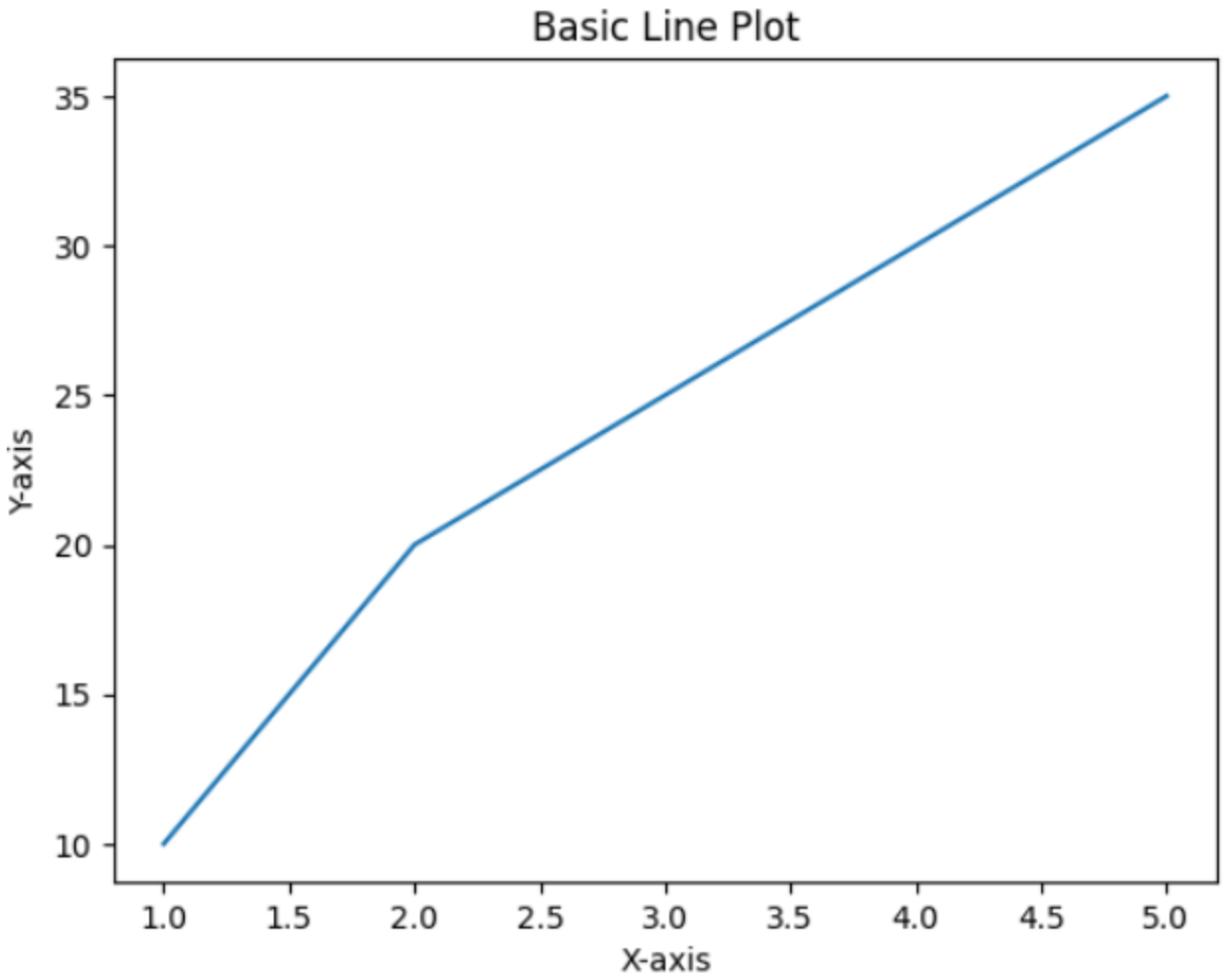
It’s widely used in data science, machine learning, and scientific computing for turning data into clear visual stories.

- Create a wide variety of plots: line, bar, scatter, histograms, pie charts, and more
- Highly customizable (colors, labels, grids, legends, ticks)
- Works seamlessly with NumPy and Pandas
- Compatible with Jupyter Notebooks for interactive data exploration

# Matplotlib

- Simple structure

```
import matplotlib.pyplot as plt


# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 35]


# Create a simple line plot
plt.plot(x, y)
plt.title("Basic Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



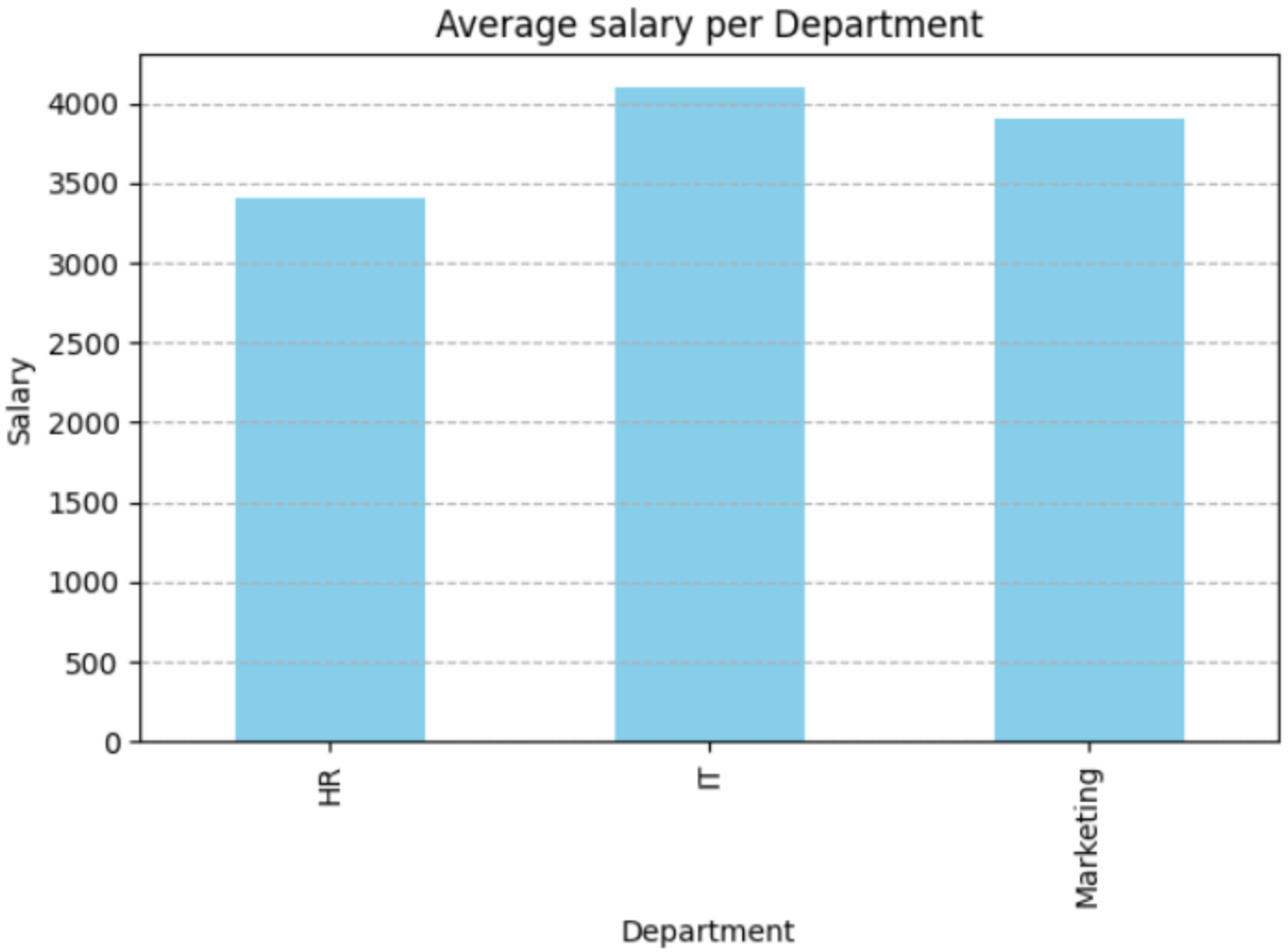
# Types of plots

- Line plot - Trends over time or sequences
- Bar plot - Comparisons between categories
- Histogram - Distribution of values
- Scatter - Relationships/correlations
- Pie chart - Proportions

# Bar chart

```
import pandas as pd
import matplotlib.pyplot as plt

data = {
'Department': ['IT', 'HR', 'IT', 'HR', 'Marketing'],
'Salary': [4000, 3500, 4200, 3300, 3900]
}

df = pd.DataFrame(data)
means = df.groupby('Departament')['Salary'].mean()

means.plot(kind='bar', title='Average salary per Department',
color='skyblue')
plt.ylabel('Salary')
plt.xlabel('Department')
plt.grid(axis='y', linestyle='--')
plt.tight_layout()
plt.show()
```

# Bar chart


Average salary per Department
Salary
4000
3500
3000
2500
2000
1500
1000
500
0
HR
IT
Marketing
Department

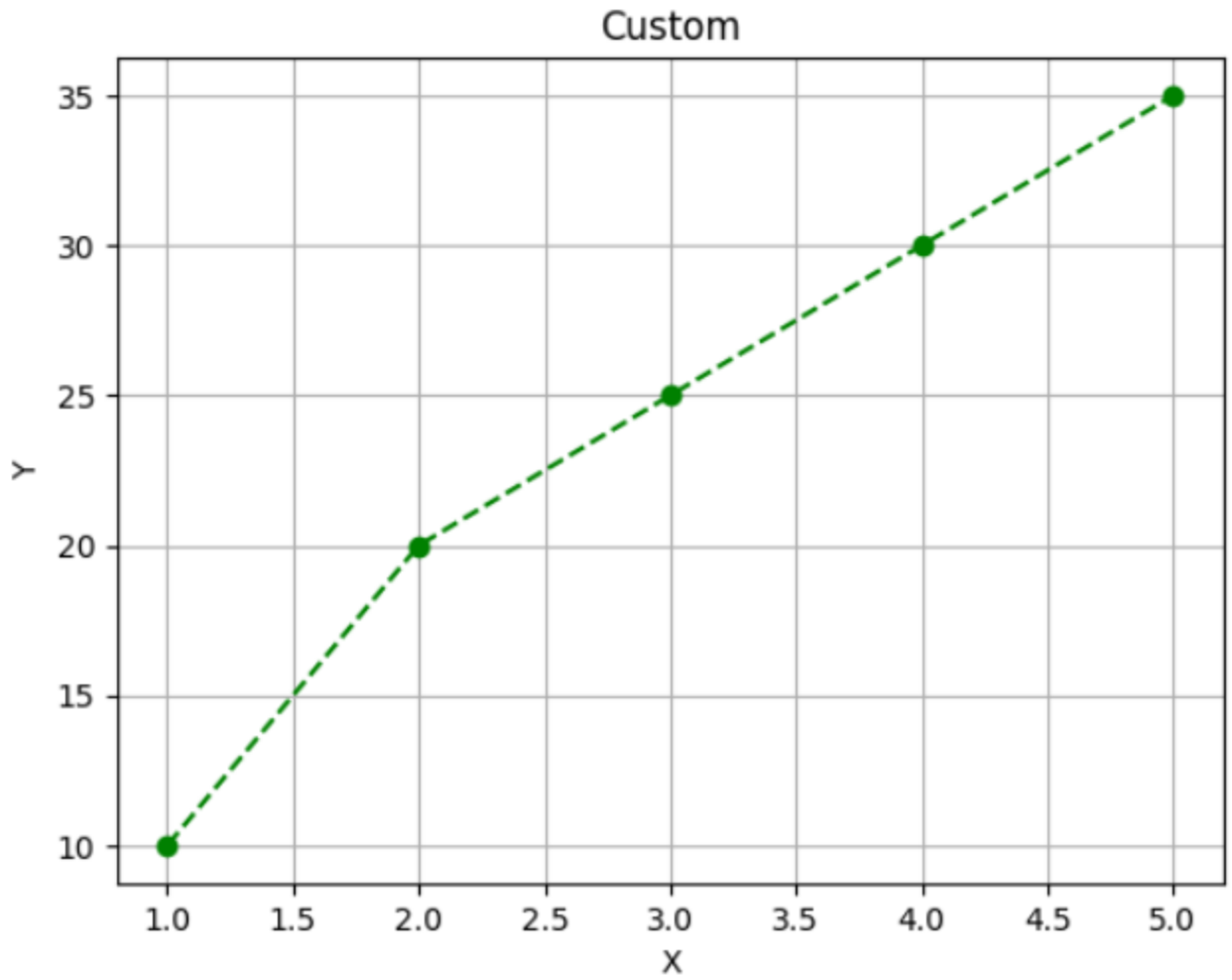
# Customizing your plots

There are multiple options of customization:

- color, linestyle, marker
- title(), xlabel(), ylabel()
- grid(), legend(), xticks(), yticks()

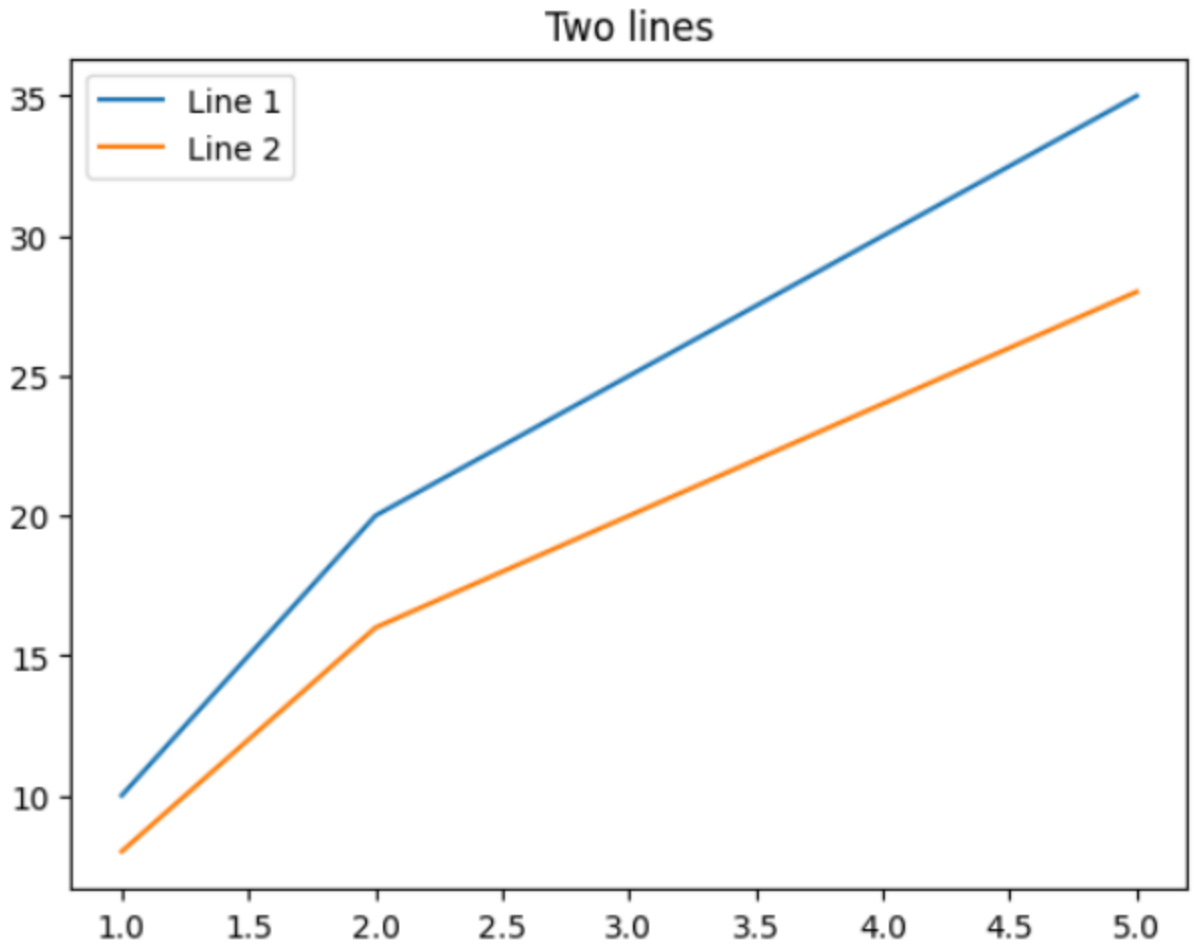
# Customizing your plots

```
plt.plot(x, y, linestyle='--', color='green', marker='o')

plt.title("Custom")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```



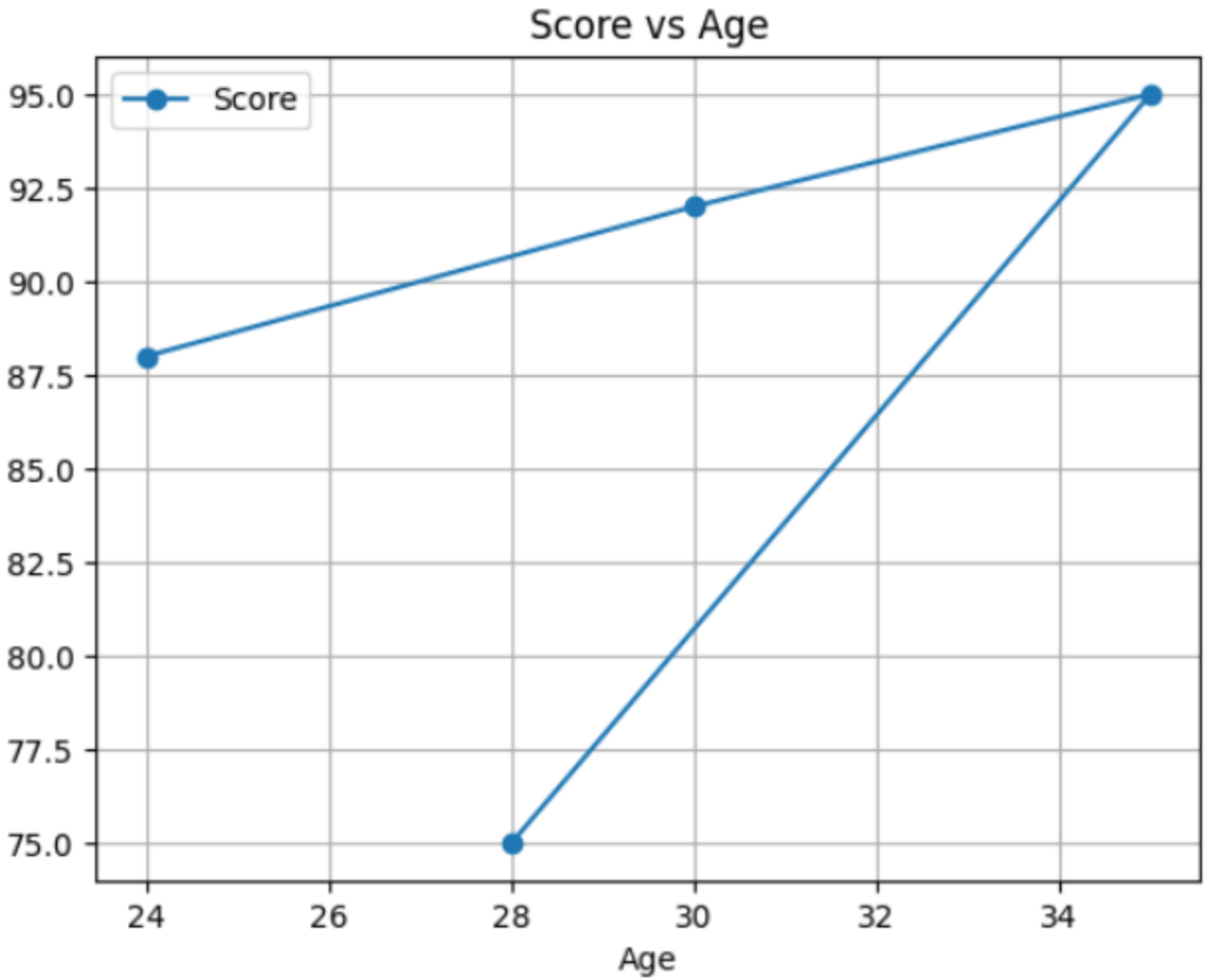
# Multiple plots on the same figure

```
plt.plot(x, y, label='Line 1')
plt.plot(x, [i*0.8 for i in y], label='Line 2')

plt.legend()
plt.title("Two lines")
plt.show()
```



# Matplotlib integrated with Pandas

```
df = pd.DataFrame({
    "Name": ["Alice", "Bob", "Charlie", "Diana"],
    "Age": [24, 30, 35, 28],
    "Score": [88, 92, 95, 75]
})

df.plot(x="Age",
        y="Score",
        kind="line",
        marker="o",
        title="Score vs Age",
        grid=True
)
```



# Matplotlib integrated with Pandas

More plot functions:

- df.plot.scatter()
- df.plot.bar()
- df.plot.hist()
- df.plot.pie()
- df.plot.area()

levi nine

# CSV Handling

# Working with csv files

Pandas has built-in in methods that handle csv files.

```
import pandas as pd

df = pd.read_csv("employees.csv") # reading csv files

data = {
    'Name': ['Ana', 'Mihai', 'Ioana', 'Radu', 'Elena'],
    'Department': ['IT', 'HR', 'IT', 'HR', 'Marketing'],
    'Age': [25, 32, 28, 31, None],
    'Salary': [4000, 3500, 4200, 3300, 3900]
}

df2 = pd.DataFrame(data)
df2.to_csv('new_employees.csv', index=False)  # writing to csv
```

# End to End Example

A full example of working with data includes the next steps:

- Load Data
- Explore
- Clean
- Visualize