Creating a Virtual Environment (Why & How)

Why Use a Virtual Environment?

A **virtual environment** is an isolated Python workspace that:

- Avoids package conflicts
- Keeps projects clean
- Uses specific library versions

Create Virtual Environment (Windows)

```
python -m venv analytics_env
```

Activate Virtual Environment

```
analytics_env\Scripts\activate
```

Once activated, you will see:

```
(analytics_env)
```

Installing Pandas

After activating the environment:

```
pip install pandas
```

Verify Installation

```
python
import pandas as pd
print(pd.__version__)
```

What is a Pandas Series? (Analytics Perspective)

A **Series** is a **one-dimensional labeled array** that can store:

- Numbers
- Text
- Boolean values
- Dates

Each value in a Series has:

- **Index** → label
- **Value** → data

In Data Analytics:
Series represents a **single column of data**, such as:

- Student marks
- Monthly sales
- Temperature readings
- Age of users

Creating a Series (Basic)

Series from a List

```
import pandas as pd

marks = pd.Series([85, 90, 78, 88])
print(marks)
```

Output:

```
0    85
1    90
2    78
3    88
dtype: int64
```

Creating Series Using Different Data Types

Integer Data

```
ages = pd.Series([18, 19, 20, 21])
```

Float Data

```
prices = pd.Series([99.5, 120.75, 89.99])
```

String Data

```
names = pd.Series(["Amit", "Neha", "Rahul"])
```

Boolean Data

```
passed = pd.Series([True, False, True])
```

Mixed Data Types

```
mixed = pd.Series([101, "Data", 95.5, True])
```

Note: *Mixed data types automatically become `object` dtype.*

Creating Series with Custom Index (Very Important)

In analytics, **index acts as an identifier**.

```
sales = pd.Series([5000, 7000, 6500],
                  index=["Jan", "Feb", "Mar"])
print(sales)
```

Useful for:

- Time-based data
- Category-based analysis

Creating Series from Dictionary (Real-World Use)

```
student_marks = pd.Series({
    "Math": 85,
    "Science": 90,
    "English": 88
})
```

Dictionary keys → Index
Dictionary values → Data

Important Series Attributes (Must Learn)

Attribute	Use
`series.index`	Shows index labels
`series.values`	Shows values
`series.dtype`	Data type
`series.shape`	Size of series
`series.size`	Number of elements
`series.name`	Name of series

```
marks = pd.Series([80, 85, 90], name="Student Marks")

print(marks.dtype)
print(marks.size)
print(marks.name)
```

Accessing Data in Series (Core Skill)

Access by Index Position

```
marks[0]
```

Access by Label

```
sales["Jan"]
```

Multiple Values

```
marks[[0, 2]]
```

Series Methods (Important for Data Analytics)

Basic Statistics

```
marks.sum()
marks.mean()
marks.max()
```

```
marks.min()
marks.count()
```

Used for:

- Average marks
- Highest sales
- Minimum temperature

Sorting

```
marks.sort_values()
marks.sort_index()
```

Checking Missing Data

```
marks.isnull()
marks.notnull()
```

Handling Missing Values

```
marks.fillna(0)
marks.dropna()
```

Vectorized Operations (Very Important Concept)

```
marks + 5
```

Adds 5 to every value
Faster than loops (industry standard)

Filtering Data (Analytics Skill)

```
marks[marks > 80]
```

Used for:

- Top performers
- High sales
- Above-average values

Series vs List (Why Series is Better)

Feature	List	Series
Index	No labels	Labeled
Vector operations	No	Yes
Missing values	Poor handling	Excellent
Analytics methods	No	Yes

What Should Learn About Series?

Must-Know Topics:

- Creating Series from different sources
- Understanding index
- Data types (`dtype`)
- Statistical methods
- Handling missing values
- Filtering & sorting
- Vectorized operations

Series = foundation of Pandas & Data Analytics

Real-World Analytics Example

```
daily_sales = pd.Series(
    [1200, 1500, 900, 1800],
    index=["Mon", "Tue", "Wed", "Thu"]
)

print(daily_sales.mean())
print(daily_sales[daily_sales > 1300])
```

Conclusion

- Pandas Series represents **single-column data**
- Extremely important for analytics
- Used in:
    - Reports
    - Dashboards
    - Machine Learning preprocessing

What is a DataFrame?

A **DataFrame** is a **two-dimensional, tabular data structure** with:

- Rows
- Columns
- Index
- Column labels

In Data Analytics:
A DataFrame represents:

- Excel sheet
- SQL table
- CSV dataset

Creating a DataFrame

From Dictionary (Most Common)

```
import pandas as pd
```

```
data = {
    "Name": ["Amit", "Neha", "Rahul"],
    "Age": [22, 21, 23],
    "Marks": [85, 90, 88]
}

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

From List of Lists

```
data = [
    ["Amit", 22, 85],
    ["Neha", 21, 90],
    ["Rahul", 23, 88]
]

df = pd.DataFrame(data, columns=["Name", "Age", "Marks"])
```

DataFrame Attributes (Must Learn)

Attribute	Use
`df.shape`	Rows & columns
`df.columns`	Column names
`df.index`	Row index
`df.dtypes`	Data types
`df.head()`	First 5 rows
`df.tail()`	Last 5 rows
`df.info()`	Dataset summary

```
df.info()
```

Industry me pehle `info()` hi use hota hai

Selecting Columns (Basic but Important)

```
df["Marks"]
```

Returns → **Series**

```
df[["Name", "Marks"]]
```

Returns → **DataFrame**

Adding a New Column

```
df["Passed"] = df["Marks"] > 80
```

Removing Columns

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

Indexing & Selection (Very Important Topic)

`loc` → Label based

```
df.loc[0, "Marks"]
```

`iloc` → Position based

```
df.iloc[0, 2]
```

Interview favorite topic

Filtering Data (Analytics Core Skill)

```
df[df["Marks"] > 85]
```

Sorting Data

```
df.sort_values("Marks", ascending=False)
```

What are Missing Values?

Missing values are empty or unavailable data points.

Examples:

- Student marks missing
- Salary not provided
- Age unknown

Pandas represents missing values as:

```
NaN  (Not a Number)
```

Creating Data with Missing Values

```
import pandas as pd
import numpy as np

data = {
    "Name": ["Amit", "Neha", "Rahul", "Priya"],
    "Marks": [85, 90, np.nan, 88],
    "Age": [22, np.nan, 23, 21]
}

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

Detecting Missing Values (MUST KNOW)

Check Missing Values

```
df.isnull()
```

Count Missing Values

```
df.isnull().sum()
```

Analytics Use:
Decide which column needs cleaning.

Handling Missing Values

1. Drop Missing Values

```
df.dropna()
```

Removes rows with missing data
Risky if dataset is small

2. Fill Missing Values (Recommended)

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

Fill with Mean (Very Common)

```
df["Marks"].fillna(df["Marks"].mean(), inplace=True)
```

Used in:

- Marks
- Salary
- Sales data

Handling Duplicate Data

Check Duplicates

```
df.duplicated()
```

Remove Duplicates

```
df.drop_duplicates()
```

Important for:

- User data
- Transaction data

Data Type Conversion (Critical Topic)

Check Data Types

```
df.dtypes
```

Convert Data Type

```
df["Age"] = df["Age"].astype(int)
```

Cleaning step before analysis or ML

Renaming Columns (Professional Practice)

```
df.rename(columns={"Marks": "Total_Marks"}, inplace=True)
```

Replacing Values

```
df["Name"] = df["Name"].replace("Amit", "Amit Kumar")
```

Used in:

- Category cleaning
- Standardization

Descriptive Statistics (Analytics Foundation)

```
df.describe()
```

Includes:

- Mean
- Min
- Max
- Std deviation

GroupBy (HEART OF DATA ANALYTICS)

What is GroupBy?

Used to **group data and perform calculations**.

Example:

- Average marks per class
- Total sales per month
- Salary per department

GroupBy Example

```
data = {
    "Department": ["IT", "IT", "HR", "HR"],
    "Salary": [50000, 60000, 45000, 48000]
}

df = pd.DataFrame(data)

df.groupby("Department")["Salary"].mean()
```

Aggregation Functions

```
df.groupby("Department")["Salary"].agg(["mean", "max", "min"])
```

Interview + industry favorite

Sorting & Ranking

```
df.sort_values("Salary", ascending=False)
```

Ranking

```
df["Rank"] = df["Salary"].rank(ascending=False)
```

Apply, Map & Lambda (Advanced but Important)

Apply

```
df["Salary"] = df["Salary"].apply(lambda x: x + 5000)
```

Map

```
df["Department_Code"] = df["Department"].map({"IT": 1, "HR": 2})
```

Reading CSV Files (REAL DATA)

```
df = pd.read_csv("students.csv")
```

Check Dataset

```
df.head()
df.info()
```

Merging, Pivot Tables, Time Series & Exporting Data

(Advanced Data Analytics Notes)

Merging & Joining DataFrames

Why Merging is Needed?

Real datasets are **never in one file**.

Examples:

- Student details → one table
- Marks → another table
- Attendance → third table

Types of Joins in Pandas

Join Type	Meaning
Inner	Common data only

Join Type	Meaning
Left	All from left + matching
Right	All from right + matching
Outer	All data from both

Merge Example (Most Used)

```
import pandas as pd

students = pd.DataFrame({
    "ID": [1, 2, 3],
    "Name": ["Amit", "Neha", "Rahul"]
})

marks = pd.DataFrame({
    "ID": [1, 2, 4],
    "Marks": [85, 90, 88]
})

result = pd.merge(students, marks, on="ID", how="inner")
print(result)
```

Left, Right & Outer Join

```
pd.merge(students, marks, on="ID", how="left")
pd.merge(students, marks, on="ID", how="right")
pd.merge(students, marks, on="ID", how="outer")
```

Note: Most common in analytics: LEFT JOIN

Concatenation (concat)

When structure is same

```
df1 = pd.DataFrame({"A": [1, 2]})
df2 = pd.DataFrame({"A": [3, 4]})

pd.concat([df1, df2])
```