Python Pandas – I

Class 12 CBSE | Sections 1.7 to 1.12 | Study Notes with Examples

(Continuation from Part 1: Sections 1.2 – 1.6)

1.7 Series Objects vs. 1D Data Structures and 1D NumPy Arrays

1.7.1 Series Objects vs. Lists

Both Series and Lists store sequences of data, but they have important differences:

Feature	Python List	Pandas Series
Index	Only positional (0,1,2...)	Positional + custom labels
Data Types	Mixed types allowed	Homogeneous (same type preferred)
Operations	Loop needed for math	Vector operations (no loop)
Missing Values	No built-in support	Supports NaN
Performance	Slower for large data	Faster (NumPy-based)

```
import pandas as pd

# List – need loop for operations
lst = [10, 20, 30]
doubled_list = [x * 2 for x in lst] # [20, 40, 60]

# Series – direct vector operation
s = pd.Series([10, 20, 30])
doubled_series = s * 2 # 0->20, 1->40, 2->60 (no loop!)
print(doubled_series)
```

1.7.2 Series Objects vs. Dictionaries

Feature	Python Dictionary	Pandas Series
Order	Unordered (Python 3.6-)	Ordered
Slicing	Not supported	Supported
Math Ops	Not directly supported	Directly supported
Missing Values	KeyError if key missing	Returns NaN
Label access	dict[key]	s[label] or s.loc[label]

1.7.3 Difference between NumPy Arrays and Series Objects

Feature	NumPy 1D Array	Pandas Series
Index	Only integer (0,1,2...)	Custom labels allowed
Data Type	Single dtype only	Single dtype (flexible)
NaN support	Limited	Full NaN support
Operations	Arithmetic only	Arithmetic + label alignment

Context	Numerical computing	Data analysis

Note: Key advantage of Series over NumPy arrays: When you add two Series, Pandas automatically aligns values by their INDEX LABELS, not just position.

1.8 DataFrame Data Structure

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

- **Rows** – identified by a row index (like row numbers in Excel)
- **Columns** – identified by column labels (like column headers)
- Each column is essentially a **Series** object

Think of a DataFrame like a spreadsheet or a database table:

	Name	Age	Marks
0	Aman	16	85
1	Bina	17	90
2	Chetan	16	78

↑ Row Index ↑ Column Labels

1.9 Creating and Displaying a DataFrame

Syntax:

```
pandas.DataFrame(data, index=row_labels, columns=col_labels)
```

Method 1 – From a Dictionary of Lists

```
import pandas as pd

data = {
'Name': ['Aman', 'Bina', 'Chetan'],
'Age': [16, 17, 16],
'Marks': [85, 90, 78]
}
df = pd.DataFrame(data)
print(df)

# Name Age Marks
# 0 Aman 16 85
# 1 Bina 17 90
# 2 Chetan 16 78
```

Note: Dictionary keys automatically become Column names. Row index starts from 0.

Method 2 – From a List of Dictionaries

```
import pandas as pd
```

```
data = [
{'Name': 'Aman', 'Marks': 85},
{'Name': 'Bina', 'Marks': 90},
{'Name': 'Chetan', 'Marks': 78}
]
df = pd.DataFrame(data)
print(df)

# Name Marks
# 0 Aman 85
# 1 Bina 90
# 2 Chetan 78
```

Method 3 – With Custom Row Index

```
import pandas as pd

data = {'Maths': [95, 88, 76], 'Science': [89, 92, 70]}
df = pd.DataFrame(data, index=['Aman', 'Bina', 'Chetan'])
print(df)

# Maths Science
# Aman 95 89
# Bina 88 92
# Chetan 76 70
```

Method 4 – From a 2D NumPy Array

```
import pandas as pd
import numpy as np

arr = np.array([[85, 89], [90, 92], [78, 70]])
df = pd.DataFrame(arr, columns=['Maths','Science'],
index=['Aman','Bina','Chetan'])
print(df)

# Maths Science
# Aman 85 89
# Bina 90 92
# Chetan 78 70
```

1.10 DataFrame Attributes

Important attributes of a DataFrame (assume df is the DataFrame variable):

Attribute	Description	Example Output
df.index	Row index labels	RangeIndex(start=0, stop=3)
df.columns	Column labels	Index(['Name','Age','Marks'])
df.dtypes	Data type of each column	Name: object, Age: int64, ...

df.shape	Rows x Columns tuple	(3, 3)
df.size	Total number of elements	9
df.ndim	Number of dimensions	2
df.values	Data as 2D NumPy array	array([['Aman',16,85],...])
df.head(n)	First n rows (default 5)	First 5 rows
df.tail(n)	Last n rows (default 5)	Last 5 rows
df.info()	Summary of DataFrame	dtype, non-null count, etc.
df.describe()	Statistical summary	count, mean, std, min, max...

```
import pandas as pd

data = {'Name':['Aman','Bina','Chetan'], 'Age':[16,17,16], 'Marks':[85,90,78]}
df = pd.DataFrame(data)

print(df.shape) # (3, 3)
print(df.columns) # Index(['Name', 'Age', 'Marks'], dtype='object')
print(df.dtypes)
# Name object
# Age int64
# Marks int64

print(df.describe())
# Age Marks
# count 3.0 3.000000
# mean 16.3 84.333333
# std 0.57 6.027714
# min 16.0 78.000000
# max 17.0 90.000000
```

1.11 DataFrame vs. Series and 2D NumPy Array

1.11.1 DataFrame vs. Series

Feature	Series	DataFrame
Dimensions	1D (one column)	2D (rows and columns)
Index	One index (row)	Row index + column labels
Relation	—	Each column IS a Series
Access	s[label]	df[col][row] or df.loc[r,c]
Use case	Single variable data	Multi-variable/tabular data

1.11.2 DataFrame vs. 2D NumPy Array

Feature	2D NumPy Array	DataFrame
Labels	Only integer positions	Named rows and columns
Data Types	Single dtype for all	Different dtypes per column
Missing Data	No native NaN support	Full NaN handling

Operations	Math/matrix operations	Data analysis functions
Column names	Not available	Always available

Note: A DataFrame is like a 2D NumPy array with row and column labels attached, plus the ability to hold different data types in different columns.

1.12 Selecting or Accessing Data

Assume the following DataFrame for all examples in this section:

```
import pandas as pd

data = {
'Name': ['Aman', 'Bina', 'Chetan', 'Dia'],
'Class': [12, 11, 12, 11],
'Maths': [85, 90, 78, 92],
'Science':[89, 88, 76, 95]
}
df = pd.DataFrame(data, index=['S1','S2','S3','S4'])
print(df)
# Name Class Maths Science
# S1 Aman 12 85 89
# S2 Bina 11 90 88
# S3 Chetan 12 78 76
# S4 Dia 11 92 95
```

1.12.1 Selecting / Accessing a Single Column

```
# Method 1: Using column name in square brackets
print(df['Maths'])
# S1 85
# S2 90
# S3 78
# S4 92
# Name: Maths, dtype: int64

# Method 2: Dot notation (only for simple column names)
print(df.Maths) # Same output as above
```

Note: A single column selected from a DataFrame is returned as a **Series**.

1.12.2 Selecting / Accessing Multiple Columns

```
# Pass a LIST of column names inside double brackets
print(df[['Name', 'Maths']])
# Name Maths
# S1 Aman 85
# S2 Bina 90
# S3 Chetan 78
# S4 Dia 92
```

Note: Multiple columns return a **DataFrame**. Use double brackets [[ ]].

1.12.3 Selecting a Subset using Row/Column – loc[] and iloc[]

loc[] – Label-based selection (uses index labels and column names)

```
# Single row by label
print(df.loc['S2'])
# Name Bina
# Class 11
```

```
# Maths 90
# Science 88

# Single value: row S1, column Maths
print(df.loc['S1', 'Maths']) # Output: 85

# Multiple rows and columns
print(df.loc['S1':'S3', 'Maths':'Science'])
# Maths Science
# S1 85 89
# S2 90 88
# S3 78 76
```

Note: loc[] is INCLUSIVE on both ends (both start and stop labels are included).

iloc[] – Position-based selection (uses integer positions 0, 1, 2...)

```
# Row at position 0
print(df.iloc[0]) # First row (S1 - Aman)

# Single value: row 0, column 2
print(df.iloc[0, 2]) # Output: 85 (Maths of Aman)

# Rows 0 to 2, columns 2 to 3
print(df.iloc[0:3, 2:4])
# Maths Science
# S1 85 89
# S2 90 88
# S3 78 76
```

Note: iloc[] is EXCLUSIVE on the stop (like Python list slicing). iloc[0:3] gives rows 0, 1, 2 — NOT row 3.

	loc[]	iloc[]
Type	Label-based	Position-based (integer)
Stop	INCLUSIVE	EXCLUSIVE
Example	df.loc['S1','Maths']	df.iloc[0, 2]

1.12.4 Selecting Rows / Columns from a DataFrame

```
# Select specific rows using loc
print(df.loc[['S1', 'S3']])
# Name Class Maths Science
# S1 Aman 12 85 89
# S3 Chetan 12 78 76

# Select all rows, specific columns
print(df.loc[:, ['Maths', 'Science']])
# Maths Science
# S1 85 89
# S2 90 88
# S3 78 76
```

```
# S4 92 95
```

1.12.5 Selecting Rows/Columns from a DataFrame (Condition-based)

```
# Select rows where Class is 12
print(df[df['Class'] == 12])
# Name Class Maths Science
# S1 Aman 12 85 89
# S3 Chetan 12 78 76

# Students with Maths > 85
print(df[df['Maths'] > 85])
# Name Class Maths Science
# S2 Bina 11 90 88
# S4 Dia 11 92 95

# Multiple conditions: Class 11 AND Maths > 88
print(df[(df['Class'] == 11) & (df['Maths'] > 88)])
# Name Class Maths Science
# S2 Bina 11 90 88
# S4 Dia 11 92 95
```

Note: Use **&** for AND, **|** for OR between conditions. Always wrap each condition in parentheses ( ).

1.12.6 Selecting/Accessing Individual Value

```
# at[] – Label-based, single value (faster than loc for single cell)
print(df.at['S2', 'Maths']) # Output: 90

# iat[] – Position-based, single value (faster than iloc for single cell)
print(df.iat[1, 2]) # Output: 90 (row 1, col 2)
```

Method	Type	Use For	Example
at[]	Label-based	Single cell only	df.at['S2','Maths']
iat[]	Position-based	Single cell only	df.iat[1,2]
loc[]	Label-based	Rows/cols/slices	df.loc['S1':'S3','Maths']
iloc[]	Position-based	Rows/cols/slices	df.iloc[0:3, 2:4]

Quick Revision – Important Points (Part 2)

Topic	Key Point
Series vs List	Series supports custom index, vector ops, NaN
Series vs Dict	Series is ordered and supports slicing
Series vs NumPy	Series aligns by label; NumPy aligns by position
DataFrame	2D table; each column is a Series
Create DataFrame	From dict of lists, list of dicts, 2D array
df.shape	Returns (rows, columns) tuple
df.describe()	Statistical summary (mean, min, max, std, ...)
Single column	df['ColName'] → returns Series
Multiple columns	df[['Col1','Col2']] → returns DataFrame
loc[]	Label-based; stop is INCLUSIVE
iloc[]	Position-based; stop is EXCLUSIVE
at[] / iat[]	Fastest way to get a single cell value
Condition filtering	df[df[col] > value] — use & for AND, | for OR