Python Pandas – I

Class 12 CBSE | Sections 1.13 to 1.15 | Study Notes with Examples

(Continuation from Part 2: Sections 1.7 – 1.12)

Common DataFrame used throughout this chapter (run this first before trying any example):

```
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'])

# Name Class Maths Science
# S1 Aman 12 85 89
# S2 Bina 11 90 88
# S3 Chetan 12 78 76
# S4 Dia 11 92 95
```

1.13 Adding / Modifying a DataFrame

After creating a DataFrame, you can **add new columns**, **add new rows**, or **modify existing values**.

1.13.1 Adding a New Column

Method A – Direct Assignment

```
# Add a 'Total' column (Maths + Science)
df['Total'] = df['Maths'] + df['Science']
print(df)

# Name Class Maths Science Total
# S1 Aman 12 85 89 174
# S2 Bina 11 90 88 178
# S3 Chetan 12 78 76 154
# S4 Dia 11 92 95 187
```

■ **Note:** New column is added at the **rightmost** position by default.

Method B – assign() method (returns a new DataFrame)

```
df2 = df.assign(Percentage = (df['Total'] / 200) * 100)
print(df2[['Name','Total','Percentage']])

# Name Total Percentage
# S1 Aman 174 87.0
# S2 Bina 178 89.0
```

```
# S3 Chetan 154 77.0
# S4 Dia 187 93.5
```

■ **Note:** assign() does NOT modify the original DataFrame. It returns a **new** DataFrame with the added column.

Method C – insert() to add at a specific position

```
# insert(position, column_name, values)
df.insert(2, 'English', [80, 75, 88, 91])
print(df.columns.tolist())

# ['Name', 'Class', 'English', 'Maths', 'Science', 'Total']
# pos 0 pos 1 pos 2 pos 3 pos 4 pos 5
```

1.13.2 Adding a New Row

Method A – Using loc[] with a new index label

```
# Add a new student row with index 'S5'
df.loc['S5'] = ['Esha', 12, 88, 91]
print(df.tail(2))

# Name Class Maths Science
# S4 Dia 11 92 95
# S5 Esha 12 88 91
```

Method B – Using pd.concat() (preferred in newer Pandas)

```
new_row = pd.DataFrame(
[['Farhan', 11, 79, 83]],
columns=['Name','Class','Maths','Science'],
index=['S6']
)
df = pd.concat([df, new_row])
print(df.tail(2))

# Name Class Maths Science
# S5 Esha 12 88 91
# S6 Farhan 11 79 83
```

■ **Important:** The old **df.append()** method has been **removed** in Pandas 2.0+. Always use **pd.concat()** to add rows in current versions.

1.13.3 Modifying Existing Values

Modify a single cell

```
# Change Bina's Maths marks to 95
df.at['S2', 'Maths'] = 95
print(df.loc['S2'])
# Name Bina
# Class 11
# Maths 95 <- updated
# Science 88
```

Modify an entire column

```
# Apply 5-mark bonus to all Maths scores
df['Maths'] = df['Maths'] + 5
print(df[['Name','Maths']])
# Name Maths
# S1 Aman 90
# S2 Bina 100
# S3 Chetan 83
# S4 Dia 97
```

Modify values using condition

```
# Set Science to 0 for students of Class 12
df.loc[df['Class'] == 12, 'Science'] = 0
print(df[['Name','Class','Science']])
# Name Class Science
# S1 Aman 12 0
# S2 Bina 11 88
# S3 Chetan 12 0
# S4 Dia 11 95
```

1.13.4 Renaming Columns

```
# rename() with a dictionary mapping old -> new names
df = df.rename(columns={'Maths': 'Math', 'Science': 'Sci'})
print(df.columns.tolist())
# ['Name', 'Class', 'Math', 'Sci']

# Rename row index labels
df = df.rename(index={'S1': 'R1', 'S2': 'R2'})
print(df.index.tolist())
# ['R1', 'R2', 'S3', 'S4']
```

■ **Note:** rename() returns a new DataFrame by default. Use **inplace=True** to modify the original: df.rename(columns={...}, inplace=True)

1.14 Deleting Rows and Columns

The **drop()** method is used to delete rows or columns from a DataFrame. By default it returns a **new** DataFrame — use **inplace=True** to modify the original.

Syntax:

```
df.drop(labels, axis=0, inplace=False)
# axis=0 -> delete ROWS (default)
# axis=1 -> delete COLUMNS
```

1.14.1 Deleting a Column

```
# Delete the 'Class' column (axis=1 means column)
df2 = df.drop('Class', axis=1)
print(df2)
# Name Maths Science
# S1 Aman 85 89
# S2 Bina 90 88
# S3 Chetan 78 76
# S4 Dia 92 95

# Delete multiple columns at once
df2 = df.drop(['Class','Science'], axis=1)
print(df2)
# Name Maths
# S1 Aman 85
# S2 Bina 90
# S3 Chetan 78
# S4 Dia 92
```

■ **Note:** axis=1 tells Pandas to operate along columns. Think: axis=1 = 'drop this column'.

Using del keyword (modifies in-place, no return value)

```
del df['Science']
# 'Science' column is permanently removed from df
```

■ **Important:** del is permanent and cannot be undone. Prefer drop() so you can keep the original.

Using pop() – removes column and returns it as a Series

```
removed = df.pop('Science')
print(removed)
# S1 89
# S2 88
# S3 76
# S4 95
# Name: Science, dtype: int64

# 'Science' is now removed from df
```

1.14.2 Deleting a Row

```
# Delete row with index label 'S3'
df2 = df.drop('S3', axis=0)
print(df2)
# Name Class Maths Science
# S1 Aman 12 85 89
# S2 Bina 11 90 88
# S4 Dia 11 92 95

# Delete multiple rows
df2 = df.drop(['S2','S4'], axis=0)
print(df2)
# Name Class Maths Science
# S1 Aman 12 85 89
# S3 Chetan 12 78 76
```

■ **Note:** axis=0 is the default, so you can also write just df.drop('S3').

Delete rows using condition (Boolean masking)

```
# Keep only rows where Maths > 80 (removes rows where Maths <= 80)
df2 = df[df['Maths'] > 80]
print(df2)
# Name Class Maths Science
# S1 Aman 12 85 89
# S2 Bina 11 90 88
# S4 Dia 11 92 95
```

1.14.3 inplace Parameter

```
# Without inplace (original df is unchanged)
df2 = df.drop('S3') # df still has S3
print('S3' in df.index) # True

# With inplace=True (original df is modified)
df.drop('S3', inplace=True) # S3 removed from df itself
print('S3' in df.index) # False
```

Parameter	drop() behaviour	Original df
inplace=False	Returns new DataFrame (default)	Unchanged
inplace=True	Modifies df directly, returns None	Changed

Quick Reference – drop()

Task	Code	axis
Delete one column	df.drop('Col', axis=1)	1
Delete multiple columns	df.drop(['C1','C2'], axis=1)	1
Delete one row	df.drop('RowLabel')	0 (default)
Delete multiple rows	df.drop(['R1','R2'])	0 (default)
Delete & save to df	df.drop('Col', axis=1, inplace=True)	1
Using del	del df['ColName']	N/A
Using pop()	s = df.pop('ColName')	N/A

1.15 Operations on DataFrame

Pandas provides many built-in functions to perform mathematical, statistical, and structural operations on DataFrames.

1.15.1 Arithmetic Operations on DataFrame

Arithmetic operations are applied **element-wise** on numeric columns:

```
import pandas as pd

df = pd.DataFrame({
'Maths': [85, 90, 78, 92],
'Science': [89, 88, 76, 95]
}, index=['S1','S2','S3','S4'])

# Add 5 to every element
print(df + 5)
# Maths Science
# S1 90 94
# S2 95 93
# S3 83 81
# S4 97 100

# Multiply every element by 2
print(df * 2)

# Two DataFrames can be added (index must match)
df2 = pd.DataFrame({'Maths':[5,5,5,5],'Science':[10,10,10,10]},
index=['S1','S2','S3','S4'])
print(df + df2)
# Maths Science
# S1 90 99
# S2 95 98
# S3 83 86
# S4 97 105
```

1.15.2 Transposing a DataFrame

The **.T** attribute (or **transpose()**) swaps rows and columns:

```
df = pd.DataFrame({
'Maths': [85, 90, 78],
'Science': [89, 88, 76]
}, index=['Aman','Bina','Chetan'])

print(df.T)
# Aman Bina Chetan
# Maths 85 90 78
# Science 89 88 76
```

■ **Note:** Transposing is useful when you want to display subjects as rows and students as columns.

1.15.3 Statistical Functions on DataFrame

These functions compute statistics **column-wise** by default (axis=0):

```
import pandas as pd

df = pd.DataFrame({
'Maths': [85, 90, 78, 92],
'Science': [89, 88, 76, 95]
}, index=['S1','S2','S3','S4'])

print(df.sum()) # Column totals
# Maths 345
# Science 348

print(df.mean()) # Column averages
# Maths 86.25
# Science 87.00

print(df.max()) # Maximum in each column
# Maths 92
# Science 95

print(df.min()) # Minimum in each column
# Maths 78
# Science 76

print(df.count()) # Non-null count per column
# Maths 4
# Science 4

print(df.std()) # Standard deviation
print(df.median()) # Median value
```

Function	Description	Example
df.sum()	Sum of each column	Maths: 345
df.mean()	Average of each column	Maths: 86.25
df.max()	Maximum value per column	Maths: 92
df.min()	Minimum value per column	Maths: 78
df.median()	Middle value per column	Maths: 87.5
df.std()	Standard deviation per column	—
df.count()	Non-null count per column	Maths: 4
df.describe()	Full statistical summary	count,mean,std,min,max

Row-wise operations using axis=1

```
# Sum across each ROW (add Maths + Science for each student)
print(df.sum(axis=1))
# S1 174
# S2 178
# S3 154
```

```
# S4 187

print(df.mean(axis=1)) # Average per student (row-wise)
# S1 87.0
# S2 89.0
# S3 77.0
# S4 93.5
```

axis value	Operates along	Result
axis=0 (default)	Down each column	One value per column
axis=1	Across each row	One value per row

1.15.4 Sorting a DataFrame

sort_values() – sort by column values

```
import pandas as pd

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

# Sort by Maths ascending (default)
print(df.sort_values('Maths'))
# Name Maths Science
# S3 Chetan 78 76
# S1 Aman 85 89
# S2 Bina 90 88
# S4 Dia 92 95

# Sort by Maths descending
print(df.sort_values('Maths', ascending=False))
# Name Maths Science
# S4 Dia 92 95
# S2 Bina 90 88
# S1 Aman 85 89
# S3 Chetan 78 76

# Sort by multiple columns: first by Class, then by Maths
df2 = pd.DataFrame({'Name':['Aman','Bina','Chetan','Dia'],
'Class':[12,11,12,11],'Maths':[85,90,78,92]})
print(df2.sort_values(['Class','Maths'], ascending=[True, False]))
```

sort_index() – sort by row index labels

```
# Sort rows by their index label
print(df.sort_index()) # Ascending (default)
print(df.sort_index(ascending=False)) # Descending

# Sort columns alphabetically
print(df.sort_index(axis=1))
# Maths Name Science
```

```
# ...
```

■ **Note:** sort_values() sorts by data. sort_index() sorts by index/column labels. Both return a new DataFrame unless inplace=True is used.

1.15.5 Iterating over a DataFrame

```
import pandas as pd
data = {'Maths':[85,90,78],'Science':[89,88,76]}
df = pd.DataFrame(data, index=['Aman','Bina','Chetan'])

# iterrows() – iterates row by row (index, Series)
for idx, row in df.iterrows():
print(idx, row['Maths'], row['Science'])
# Aman 85 89
# Bina 90 88
# Chetan 78 76

# iteritems() / items() – iterates column by column
for col_name, col_data in df.items():
print(col_name)
print(col_data.values)
# Maths
# [85 90 78]
# Science
# [89 88 76]
```

✔ **Tip:** Use iterrows() when you need to process each row individually. For performance, prefer vectorized operations over loops whenever possible.

1.15.6 Applying Functions – apply() and map()

apply() – apply a function to each column or row

```
import pandas as pd
df = pd.DataFrame({'Maths':[85,90,78,92],'Science':[89,88,76,95]},
index=['S1','S2','S3','S4'])

# Apply to each column (axis=0 default): range = max - min
print(df.apply(lambda x: x.max() - x.min()))
# Maths 14
# Science 19

# Apply to each row (axis=1): total marks
print(df.apply(lambda x: x.sum(), axis=1))
# S1 174
# S2 178
# S3 154
# S4 187
```

map() – element-wise transformation on a column (Series)

```
# Convert numeric grades to letter grades for Maths column
def grade(m):
```

```
if m >= 90: return 'A+'
elif m >= 80: return 'A'
else: return 'B'

df['Grade'] = df['Maths'].map(grade)
print(df)
# Maths Science Grade
# S1 85 89 A
# S2 90 88 A+
# S3 78 76 B
# S4 92 95 A+
```

1.15.7 Handling Missing Values (NaN)

NaN (Not a Number) represents missing or undefined data in Pandas. Key functions to handle NaN:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
'Name': ['Aman', 'Bina', 'Chetan', 'Dia'],
'Maths': [85, np.nan, 78, 92],
'Science': [89, 88, np.nan, 95]
})

# 1. isnull() – returns True where value is NaN
print(df.isnull())
# Name Maths Science
# 0 False False False
# 1 False True False
# 2 False False True
# 3 False False False

# 2. notnull() – opposite of isnull()
print(df.notnull())

# 3. Count missing values per column
print(df.isnull().sum())
# Name 0
# Maths 1
# Science 1

# 4. dropna() – remove rows with ANY NaN
print(df.dropna())
# Name Maths Science
# 0 Aman 85.0 89.0
# 3 Dia 92.0 95.0

# 5. fillna() – replace NaN with a value
print(df.fillna(0)) # Replace NaN with 0
```

```
print(df['Maths'].fillna(df['Maths'].mean())) # Replace with column mean
```

Function	Description	Returns
df.isnull()	True where value is NaN	Boolean DataFrame
df.notnull()	True where value is NOT NaN	Boolean DataFrame
df.isnull().sum()	Count of NaN per column	Series
df.dropna()	Remove rows containing NaN	New DataFrame
df.fillna(val)	Replace NaN with specified value	New DataFrame

■ **Important:** Always check for missing values using df.isnull().sum() before performing any analysis. NaN values can give incorrect statistical results.

Quick Revision – Important Points (Part 3)

Topic	Key Point
Add new column	df['NewCol'] = values or df.assign(col=values)
Add at position	df.insert(pos, 'ColName', values)
Add new row	df.loc['NewIdx'] = [...] or pd.concat([df, new_row])
Modify single cell	df.at['row','col'] = value
Modify by condition	df.loc[df['col']>val, 'col2'] = new_val
Rename columns	df.rename(columns={'old':'new'})
Delete column (drop)	df.drop("Col", axis=1)
Delete row (drop)	df.drop("RowLabel", axis=0)
Delete permanently	del df['Col'] or df.drop(..., inplace=True)
pop()	s = df.pop('Col') — removes & returns column as Series
sum/mean/max/min	df.sum(), df.mean(), df.max(), df.min()
Row-wise stats	df.sum(axis=1), df.mean(axis=1)
Sort by values	df.sort_values('Col', ascending=True/False)
Sort by index	df.sort_index()
Transpose	df.T or df.transpose()
apply()	df.apply(func, axis=0/1) — column or row wise
map()	df['Col'].map(func) — element-wise on a Series
Check NaN	df.isnull().sum()
Remove NaN rows	df.dropna()
Fill NaN	df.fillna(value)