Python Pandas - I

Complete Exam Questions and Answers

Theory + Descriptive + Practical | Class 12 CBSE | Sections 1.2 to 1.15

Section	Type	Questions	Marks Each	Total Marks
A	Multiple Choice Questions (MCQ)	15	1	15
B	Theory / Descriptive (Short)	12	2	24
C	Theory + Code Mixed (Short Answer)	10	3	30
D	Long Answer (Theory + Code)	5	5	25
	GRAND TOTAL	42 Questions		94 Marks

SECTION A - Multiple Choice Questions (1 Mark Each)

Q1. Which command correctly imports Pandas with alias pd?

[1 Mark]

(a) import pandas
(b) import pandas as pd
(c) import pd as pandas
(d) from pandas import pd

Answer: (B) import pandas as pd

Explanation: pd is the universally accepted short alias for Pandas. Using an alias saves typing time. Without the alias you would need to write pandas.Series() every time instead of pd.Series().

Q2. Which of the following is a ONE-dimensional data structure in Pandas?

[1 Mark]

(a) DataFrame
(b) Matrix
(c) Series
(d) 2D Array

Answer: (C) Series

Explanation: A Series is 1D - it has only one axis (rows). A DataFrame is 2D with both rows and columns. Think of Series as a single column of a spreadsheet.

Q3. What is the default starting index of a Series when no index is specified?

[1 Mark]

(a) 1
(b) None
(c) a
(d) 0

Answer: (D) 0

Explanation: Python and Pandas both follow 0-based indexing. So a Series with 4 elements gets index 0, 1, 2, 3 automatically unless you provide custom index labels.

Q4. Which attribute returns the total number of elements in a Series?

[1 Mark]

(a) .length
(b) .count()
(c) .size
(d) .shape

Answer: (C) .size

Explanation: s.size gives total element count as an integer (e.g. 4). s.shape gives a tuple like (4,). s.count() counts only non-NaN values - different from size!

Q5. df.shape for a DataFrame with 5 rows and 3 columns returns:

[1 Mark]

(a) (3, 5)
(b) (5, 3)
(c) 15
(d) (5,)

Answer: (B) (5, 3)

Explanation: df.shape always returns (number_of_rows, number_of_columns). Rows come first, then columns - same order as matrix notation (m x n).

Q6. Which function returns the LAST n rows of a DataFrame?

[1 Mark]

(a) df.last(n)
(b) df.bottom(n)
(c) df.tail(n)
(d) df.end(n)

Answer: (C) df.tail(n)

Explanation: df.tail(n) returns the last n rows. Default n=5 if not specified. df.head(n) returns the first n rows. These are useful for quickly previewing large datasets.

Q7. loc[] uses _____ based indexing.

[1 Mark]

(a) Position/Integer
(b) Boolean
(c) Label/Name
(d) Random

Answer: (C) Label/Name

Explanation: loc[] uses the actual row/column names (labels) you assigned. iloc[] uses integer position numbers (0, 1, 2...). Memory trick: loc = Labels, iloc = Integers.

Q8. What value appears when two Series with non-matching index labels are added?

[1 Mark]

(a) 0
(b) None
(c) NaN
(d) Error is raised

Answer: (C) NaN

Explanation: NaN means Not a Number. Pandas cannot perform arithmetic on a value that has no matching pair in the other Series, so it inserts NaN. NaN is a float value from the NumPy library.

Q9. Which statement correctly deletes the column 'Marks' from df?

[1 Mark]

(a) df.remove('Marks')
(b) df.drop('Marks', axis=1)
(c) del df.Marks
(d) df.delete('Marks', axis=1)

Answer: (B) df.drop('Marks', axis=1)

Explanation: drop() with axis=1 removes a column. axis=0 removes a row. del df['Marks'] also works but permanently modifies df. drop() is safer as it returns a new DataFrame by default.

Q10. df.isnull().sum() returns:

[1 Mark]

(a) Total NaN count in the whole DataFrame
(b) Count of NaN values in each column
(c) Count of NaN values in each row
(d) A single True/False value

Answer: (B) Count of NaN values in each column

Explanation: isnull() creates a Boolean DataFrame (True where NaN). sum() then adds up the True values (1) column by column. This gives how many missing values exist in each column.

Q11. To apply a function ROW-WISE on a DataFrame, use df.apply(func, axis=___)

[1 Mark]

(a) axis=0
(b) axis=2
(c) axis=-1
(d) axis=1

Answer: (D) axis=1

Explanation: axis=1 means "apply across each row" (go sideways). axis=0 (default) means "apply down each column" (go downward). Example: df.apply(sum, axis=1) gives total marks per student (row).

Q12. Which creates a Series where dictionary keys become index labels?

[1 Mark]

(a) pd.Series([10,20,30])
(b) pd.Series({'Maths':95, 'Science':88})
(c) pd.Series(range(3))
(d) pd.DataFrame({'a':1,'b':2})

Answer: (B) pd.Series({'Maths':95, 'Science':88})

Explanation: When a dictionary is passed to pd.Series(), keys automatically become the index labels and values become the data. This is very convenient for creating labelled data.

Q13. In loc[] slicing, the stop label is _____.

[1 Mark]

(a) Excluded (like Python lists)
(b) Skipped
(c) Included
(d) Converted to integer

Answer: (C) Included

Explanation: This is an important difference: loc["S1":"S3"] includes S1, S2, AND S3 (stop is inclusive). iloc[0:3] includes positions 0, 1, 2 but NOT 3 (stop is exclusive). Always remember: loc = inclusive, iloc = exclusive at stop.

Q14. Which function replaces NaN values with a specified value?

[1 Mark]

(a) df.fillna(value)
(b) df.dropna()
(c) df.replace()
(d) df.isnull()

Answer: (A) df.fillna(value)

Explanation: fillna(value) keeps all rows but replaces NaN with the given value. dropna() removes entire rows that contain NaN. Common use: df["Marks"].fillna(df["Marks"].mean()) fills NaN with column average.

Q15. What does the .T attribute do to a DataFrame?

[1 Mark]

(a) Sorts the DataFrame
(b) Truncates to 5 rows
(c) Transposes - swaps rows and columns
(d) Returns data types

Answer: (C) Transposes - swaps rows and columns

Explanation: Transposing flips the DataFrame so rows become columns and columns become rows. Example: a 4x3 DataFrame becomes 3x4 after .T Useful when you want subjects as rows and students as columns.

SECTION B - Theory / Descriptive Questions (2 Marks Each)

Q1. What is Pandas? State any TWO advantages of using Pandas in Python.

[2 Marks]

Answer:

Pandas is an open-source Python library used for data manipulation, analysis, and cleaning. It is built on top of NumPy and provides fast, flexible, and expressive data structures.

Two Advantages of Pandas:

1. Easy Handling of Missing Data: Pandas automatically handles NaN (missing values) without crashing the program, unlike plain Python lists.
2. Powerful Data Structures: Series (1D) and DataFrame (2D) make it very easy to store, access, and analyse structured data efficiently.
3. Supports Multiple File Formats: Can read/write CSV, Excel, JSON, SQL etc. using simple one-line commands like pd.read_csv().

(Any two valid advantages are accepted)

Explanation: Pandas is named after "Panel Data" - a term from econometrics for multidimensional structured data sets. It is the most widely used library for data analysis in Python alongside NumPy and Matplotlib.

Q2. Define Series in Pandas. What are its TWO main components? Give an example.

[2 Marks]

Answer:

A Series is a ONE-DIMENSIONAL labelled array in Pandas that can hold data of any single type (integers, floats, strings, etc.).

It is similar to a single column in a spreadsheet.

Two Main Components of a Series:

1. Values - The actual data stored in the array (e.g. marks, names, prices).
2. Index - Labels assigned to each value. Default is 0,1,2... but can be custom labels like student names.

```
import pandas as pd
s = pd.Series([85, 90, 78], index=['Aman', 'Bina', 'Chetan'])
print(s)
# Aman 85 <-- index label : value
# Bina 90
# Chetan 78
# dtype: int64
print(s.values) # array([85, 90, 78]) <-- Component 1: Values
print(s.index) # Index(['Aman','Bina','Chetan']) <-- Component 2: Index
```

Explanation: A Series is like a Python dictionary in some ways - it has key-value pairs. But unlike a dictionary, a Series is ordered, supports slicing, and allows direct arithmetic operations on all values at once.

Q3. What is a DataFrame in Pandas? How is it different from a Series? (Give 3 points)

[2 Marks]

Answer:

A DataFrame is a TWO-DIMENSIONAL labelled data structure in Pandas, similar to a table/spreadsheet with rows and columns.

It can hold different data types in different columns.

Each individual COLUMN of a DataFrame is actually a Series object.

Differences between DataFrame and Series:

1. Dimensions: Series is 1D (one column). DataFrame is 2D (rows + columns).
2. Index: Series has only row index. DataFrame has both row index AND column labels.
3. Data: Series holds one type of data. DataFrame can hold different data types in different columns (e.g. string names + integer marks).
4. Use: Series for single-variable data. DataFrame for multi-variable/tabular data.

Explanation: Relationship: A DataFrame is a collection of Series objects sharing the same index. When you select one column from a DataFrame (e.g. df["Maths"]), Pandas returns it as a Series, not a DataFrame.

Q4. Explain any FOUR attributes of a Series object with syntax and example output.

[2 Marks]

Answer:

Consider: s = pd.Series([85, 90, 78], index=["Aman","Bina","Chetan"])

1. s.values - Returns data as a NumPy array.
   Output: array([85, 90, 78])
2. s.index - Returns the index labels.
   Output: Index(["Aman", "Bina", "Chetan"], dtype="object")
3. s.dtype - Returns the data type of elements.
   Output: int64
4. s.size - Returns TOTAL number of elements.
   Output: 3
5. s.shape - Returns dimensions as a tuple.
   Output: (3,)
6. s.name - Returns the name of the Series (None if not set).
7. s.ndim - Returns number of dimensions. Always 1 for Series.

(Any four are accepted)

Explanation: Attributes are properties of an object - they do not use parentheses unlike methods. For example: s.size is an attribute (no brackets), but s.count() is a method (with brackets). Attributes directly store a value; methods compute and return a result.

Q5. Differentiate between loc[] and iloc[] in a DataFrame. Give any 3 points with examples.

[2 Marks]

Answer:

Both loc[] and iloc[] are used to select rows and columns from a DataFrame, but they differ in HOW they identify rows and columns.

loc[] (Label-Based Indexing):

- Uses the actual NAME/LABEL of rows and columns.
- Stop label is INCLUSIVE (included in result).
- Syntax: df.loc["row_label", "col_label"]
- Example: df.loc["S1":"S3", "Maths"] --> gives S1, S2, S3 rows

iloc[] (Integer Position-Based Indexing):

- Uses INTEGER POSITION numbers (0, 1, 2...).
- Stop position is EXCLUSIVE (not included, like Python list slicing).
- Syntax: df.iloc[row_number, col_number]
- Example: df.iloc[0:3, 1] --> gives rows at position 0, 1, 2 (NOT 3)

Memory Trick: loc = Labels | iloc = Integers

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

# loc[] - using label names
print(df.loc['S1':'S3', 'Maths']) # S1,S2,S3 (S3 INCLUDED)
# S1 85
# S2 90
# S3 78

# iloc[] - using position numbers
print(df.iloc[0:3, 0]) # rows 0,1,2 (3 NOT included)
# S1 85
# S2 90
# S3 78
```

Explanation: A common mistake students make: df.loc["S1":"S3"] gives 3 rows (S1,S2,S3) but df.iloc[0:3] also gives 3 rows (0,1,2) - not 4. The stop is inclusive in loc and exclusive in iloc.

Q6. What is NaN in Pandas? When does it appear? Name TWO functions to handle it.

[2 Marks]

Answer:

NaN stands for "Not a Number". It is a special floating-point value used in Pandas to represent MISSING, UNDEFINED, or UNAVAILABLE data.

NaN comes from the NumPy library (numpy.nan).

When does NaN appear?

1. When two Series with non-matching index labels are added - unmatched positions get NaN.
2. When reading a CSV/Excel file that has empty/blank cells.
3. When creating a DataFrame from a dict where some keys have fewer values.
4. When a calculation cannot produce a valid result.

Two Functions to Handle NaN:

1. df.dropna() - Removes ALL rows that contain at least one NaN.
2. df.fillna(val) - Replaces NaN with a specified value (e.g. 0 or column mean).
3. df.isnull() - Returns True where value is NaN (used to detect NaN).

```
import pandas as pd, numpy as np

s1 = pd.Series([10,20,30], index=['a','b','c'])
s2 = pd.Series([1, 2, 3], index=['a','b','d'])
print(s1 + s2)
# a 11.0 <-- matched: 10+1
# b 22.0 <-- matched: 20+2
# c NaN <-- 'c' not in s2 --> NaN
# d NaN <-- 'd' not in s1 --> NaN
```

Explanation: NaN is contagious - any arithmetic involving NaN produces NaN. For example: 85 + NaN = NaN. This is why checking and handling missing values before analysis is essential.

Q7. Explain the difference between inplace=False and inplace=True in drop().

[2 Marks]

Answer:

The inplace parameter controls whether the operation modifies the ORIGINAL DataFrame or returns a NEW one.

inplace=False (DEFAULT behaviour):

- drop() returns a NEW DataFrame with the row/column removed.
- The ORIGINAL DataFrame remains COMPLETELY UNCHANGED.
- You must store the result: df2 = df.drop("S3")
- Safer approach - you can compare old and new DataFrames.

inplace=True:

- drop() modifies the ORIGINAL DataFrame DIRECTLY.
- Returns None (nothing is returned).
- The original data is permanently changed.
- df.drop("S3", inplace=True) --> S3 gone from df itself.

Rule: inplace=True applies to many Pandas operations like rename(), sort_values(), fillna(), dropna(), etc.

```
import pandas as pd
df = pd.DataFrame({'Name':['Aman','Bina','Chetan'],'Marks':[85,90,78]},
index=['S1','S2','S3'])

# inplace=False (default) -- original df unchanged
df2 = df.drop('S2') # df2 has no S2, df still has S2
print('S2' in df.index) # True (original unchanged)
print('S2' in df2.index) # False (new df without S2)

# inplace=True -- original df is modified
df.drop('S2', inplace=True) # S2 permanently removed from df
print('S2' in df.index) # False (original changed)
```

Explanation: Think of it like editing a document: inplace=False is like "Save As" (creates a new copy with changes). inplace=True is like "Save" (overwrites the original). Always prefer inplace=False when learning, to avoid accidental data loss.

Q8. What is the difference between sort_values() and sort_index()? Give one example of each.

[2 Marks]

Answer:

Both functions sort a DataFrame but they sort based on DIFFERENT things.

sort_values(column_name):

- Sorts the DataFrame based on the DATA VALUES in a specified column.
- You must tell it WHICH column to sort by.
- Example: df.sort_values("Marks") --> rows rearranged from lowest to highest Marks.
- Use ascending=False for descending (highest first).

sort_index():

- Sorts the DataFrame based on the ROW INDEX LABELS.
- No column name needed - it uses the index automatically.
- Example: df.sort_index() --> rows arranged as S1, S2, S3, S4...
- Use axis=1 to sort column labels alphabetically.

Both return a new DataFrame by default (inplace=False).

```
import pandas as pd
data = {'Name':['Chetan','Aman','Dia','Bina'], 'Marks':[78,85,92,90]}
df = pd.DataFrame(data, index=['S3','S1','S4','S2'])

# sort_values() -- sort by Marks column DATA
print(df.sort_values('Marks'))
# Name Marks
# S3 Chetan 78
# S1 Aman 85
# S2 Bina 90
# S4 Dia 92

# sort_index() -- sort by INDEX LABELS S1,S2,S3...
print(df.sort_index())
# Name Marks
# S1 Aman 85
# S2 Bina 90
# S3 Chetan 78
# S4 Dia 92
```

Explanation: A DataFrame is not sorted by default after operations like concat() or filtering. Sorting is important before displaying results or before using range-based operations. sort_values() is more commonly used in data analysis.

Q9. What does describe() return? List FIVE statistics it provides and explain each.

[2 Marks]

Answer:

df.describe() generates a comprehensive STATISTICAL SUMMARY of all numeric columns in a DataFrame in one command.

It is extremely useful for understanding the distribution and spread of data.

Five Statistics Provided:

1. count - Number of non-null (non-NaN) values. Tells you if data is complete.
2. mean - Arithmetic average of all values in the column.
3. std - Standard deviation. Measures how spread out values are from the mean.
   (Low std = values close together, High std = values spread out)
4. min - The smallest (minimum) value in the column.
5. max - The largest (maximum) value in the column.
   (Also gives 25%, 50% (median), 75% percentile/quartile values)

Syntax: print(df.describe())

Note: describe() only works on NUMERIC columns by default.

Use df.describe(include="all") to include string columns too.

Explanation: describe() is often the FIRST function called when exploring a new dataset. If count is less than total rows, it means there are NaN values. If min and max are very far apart, the data may have outliers.

Q10. Differentiate between del, drop() and pop() for removing a column. Give syntax for each.

[2 Marks]

Answer:

All three can remove a column from a DataFrame but they differ in behaviour and what they return.

del df["ColName"]:

- Permanently removes the column from the original DataFrame.
- Returns nothing (no value is returned).
- Cannot be stored or undone.
- Syntax: del df["Science"]

df.drop("ColName", axis=1):

- Returns a NEW DataFrame without that column.
- Original df is UNCHANGED (unless inplace=True is used).
- Safer option - allows you to keep original data.
- Syntax: df2 = df.drop("Science", axis=1)

df.pop("ColName"):

- Removes the column from the ORIGINAL DataFrame.
- RETURNS the removed column as a Series (can be stored).
- Useful when you need to use the removed column elsewhere.
- Syntax: removed_col = df.pop("Science")

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

# del -- permanent, no return
del df['Science'] # Science gone from df forever

# Reset df
df['Science'] = [89, 88]

# drop() -- returns new df, original safe
df2 = df.drop('Science', axis=1) # df still has Science

# pop() -- removes & returns as Series
sci = df.pop('Science') # sci = Series([89,88]), df has no Science
print(sci)
# 0 89
# 1 88
# Name: Science, dtype: int64
```

Explanation: Comparison summary: del --> permanent removal, no return. drop() --> safe, returns new df, original safe. pop() --> permanent removal but returns the column as Series. For exam questions, drop() is the most commonly tested.

Q11. Explain apply() and map() functions in Pandas. What is the key difference between them?

[2 Marks]

Answer:

Both apply() and map() are used to apply a function to Pandas data, but they work at DIFFERENT levels.

apply():

- Works on an ENTIRE DataFrame or an entire Series (column/row).
- Can apply function COLUMN-WISE (axis=0) or ROW-WISE (axis=1).
- Used for aggregation (sum, max) or transformation across rows/columns.
- Syntax: df.apply(function, axis=0 or 1)

map():

- Works on a SINGLE SERIES (single column) ONLY.
- Applies the function to EACH INDIVIDUAL ELEMENT one by one.
- Used for element-wise transformation like converting values to grades.
- Syntax: df["col"].map(function)

Key Difference:

apply() --> operates on whole rows or columns (axis-wise)

map() --> operates on each single element (element-wise)

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

# apply() -- column-wise: range (max-min) of each column
print(df.apply(lambda x: x.max() - x.min(), axis=0))
# Maths 14
# Science 19

# apply() -- row-wise: total marks per student
df['Total'] = df.apply(lambda x: x.sum(), axis=1)
# S1->174, S2->178, S3->154, S4->187

# map() -- element-wise: convert each Maths mark to grade
def grade(m):
return 'A+' if m >= 90 else ('A' if m >= 80 else 'B')
df['Grade'] = df['Maths'].map(grade)
# S1->A, S2->A+, S3->B, S4->A+
```

Explanation: A good way to remember: apply() is like a conveyor belt that processes whole rows/columns at once. map() is like a stamp that marks each individual element one by one.

Q12. What is Transposing a DataFrame? Write the syntax. When would you use it?

[2 Marks]

Answer:

Transposing a DataFrame means SWAPPING its rows and columns.

After transposing: what were rows become columns, and what were columns become rows.

If original DataFrame has shape (4, 3), after transpose it becomes (3, 4).

Syntax:

Method 1: df.T (using the .T attribute)

Method 2: df.transpose() (using the transpose() method)

Both methods return a NEW transposed DataFrame. Original is unchanged.

When to use Transpose:

1. When you want to display subjects as rows and students as columns.
2. When comparing column statistics side by side becomes easier as rows.
3. When reading data from a file where rows/columns are swapped.
4. When preparing data for certain types of charts or reports.

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

print('Transposed:')
print(df.T)
# Aman Bina Chetan
# Maths 85 90 78
# Science 89 88 76
# Now subjects are rows and students are columns!
```

Explanation: Transposing is a concept from mathematics (matrix operations). In a matrix, the transpose of matrix A is written as A^T. Pandas uses the same concept. Transposing twice returns the original.

SECTION C - Short Answer Questions (Theory + Code) (3 Marks Each)

Q1. Explain THREE ways to create a Series in Pandas with code examples.

[3 Marks]

Answer:

A Series can be created from several data types:

1. From a List - Most basic method. Index is 0,1,2... by default.
2. From a Dictionary - Keys become index labels automatically.
3. From a List with Custom Index - Assign meaningful labels to each value.

```
import pandas as pd

# Method 1: From a List (default index 0,1,2...)
s1 = pd.Series([85, 90, 78])
print(s1)
# 0 85
# 1 90
# 2 78

# Method 2: From a Dictionary (keys become index)
s2 = pd.Series({'Maths':95, 'Science':88, 'English':76})
print(s2)
# Maths 95
# Science 88
# English 76

# Method 3: From a List with Custom Index
s3 = pd.Series([85, 90, 78], index=['Aman','Bina','Chetan'])
print(s3)
# Aman 85
# Bina 90
# Chetan 78
```

Explanation: The most common method in CBSE exams is Method 3 (custom index). Dictionary-based creation (Method 2) is very practical as it gives meaningful labels automatically without a separate index parameter.

Q2. Explain Filtering in a Series. Write code to filter students scoring above 80 AND below 95.

[3 Marks]

Answer:

Filtering means selecting only those elements from a Series that satisfy a given condition.

It works using BOOLEAN INDEXING:

Step 1: Write a condition (e.g. s > 80) -- creates True/False Series.

Step 2: Pass that inside s[] -- keeps only True positions.

Multiple conditions can be combined using & (AND) and | (OR).

```
import pandas as pd

marks = pd.Series([85, 45, 90, 38, 72, 96],
index=['Aman','Bina','Chetan','Dia','Esha','Farhan'])

# Single condition: marks above 80
above80 = marks[marks > 80]
print('Above 80:')
print(above80)
# Aman 85
# Chetan 90
# Farhan 96

# Multiple conditions: above 80 AND below 95
# Each condition MUST be in parentheses
filtered = marks[(marks > 80) & (marks < 95)]
print('Above 80 AND Below 95:')
print(filtered)
# Aman 85
# Chetan 90
```

Explanation: Common mistake: Writing marks > 80 & marks < 95 (without parentheses) will give an error because & has higher precedence than >. Always wrap each condition in its own parentheses: (marks > 80) & (marks < 95).

Q3. Write code to demonstrate the following on a DataFrame: (i) Display head(3) (ii) Display shape and dtypes (iii) Select rows where Class == 12

[3 Marks]

Answer:

Assume the following DataFrame for all three parts:

```
import pandas as pd

data = {
'Name': ['Aman','Bina','Chetan','Dia','Esha'],
'Class': [12, 11, 12, 11, 12],
'Maths': [85, 90, 78, 92, 88],
'Science': [89, 88, 76, 95, 84]
}
df = pd.DataFrame(data)

# (i) head(3) -- first 3 rows
print(df.head(3))
# Name Class Maths Science
# 0 Aman 12 85 89
# 1 Bina 11 90 88
# 2 Chetan 12 78 76

# (ii) shape and dtypes
print(df.shape) # (5, 4) --> 5 rows, 4 columns
print(df.dtypes)
# Name object (string)
# Class int64
# Maths int64
# Science int64

# (iii) Filter rows where Class is 12
class12 = df[df['Class'] == 12]
print(class12)
# Name Class Maths Science
# 0 Aman 12 85 89
# 2 Chetan 12 78 76
# 4 Esha 12 88 84
```

Explanation: df.dtypes tells you the data type of each column. "object" means string/text type in Pandas. Knowing data types is important before performing operations - you cannot do arithmetic on object/string columns.

Q4. Explain adding a new column and a new row to a DataFrame with examples.

[3 Marks]

Answer:

After creating a DataFrame, you can easily add new columns and rows.

Adding a New Column:

- Use direct assignment: df["NewCol"] = values
- The values can be a list, a constant, or a calculated expression.

Adding a New Row:

- Use df.loc["new_index_label"] = [values...]
- Or use pd.concat() to join a new DataFrame row.

```
import pandas as pd

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

# Adding a new COLUMN 'Total'
df['Total'] = df['Maths'] + df['Science']
print(df)
# Name Maths Science Total
# S1 Aman 85 89 174
# S2 Bina 90 88 178
# S3 Chetan 78 76 154

# Adding a new ROW for student 'Dia' at index S4
df.loc['S4'] = ['Dia', 92, 95, 187]
print(df.tail(2))
# Name Maths Science Total
# S3 Chetan 78 76 154
# S4 Dia 92 95 187
```

Explanation: When adding a row, the values list must match the number of columns in the correct order. If you miss a value, Pandas will raise an error. Use pd.concat() for adding many rows at once efficiently.

Q5. Write Python code to demonstrate drop() to: (i) Delete a column (ii) Delete a row (iii) Delete using inplace=True

[3 Marks]

Answer:

drop() is the primary method to remove rows or columns.

axis=0 --> removes a ROW | axis=1 --> removes a COLUMN

```
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'])

# (i) Delete COLUMN 'Science' (axis=1)
df2 = df.drop('Science', axis=1)
print(df2)
# Name Maths
# S1 Aman 85
# S2 Bina 90
# S3 Chetan 78
# S4 Dia 92
# Original df still has Science column!

# (ii) Delete ROW 'S3' (axis=0 is default)
df3 = df.drop('S3') # axis=0 is default
print(df3)
# Name Maths Science
# S1 Aman 85 89
# S2 Bina 90 88
# S4 Dia 92 95

# (iii) inplace=True -- modifies ORIGINAL df permanently
df.drop('S2', inplace=True)
print('S2' in df.index) # False -- S2 permanently removed
```

Explanation: Important: drop() without inplace=True never changes the original. You MUST either store the result (df2 = df.drop(...)) or use inplace=True to change the original.

Q6. Explain statistical functions sum(), mean(), max(), min() on a DataFrame with both column-wise and row-wise examples.

[3 Marks]

Answer:

Pandas provides built-in statistical functions that work on numeric columns.

By default they work COLUMN-WISE (axis=0) -- one result per column.

With axis=1 they work ROW-WISE -- one result per row.

```
import pandas as pd

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

# Column-wise (axis=0 -- default)
print(df.sum()) # Maths: 345, Science: 348
print(df.mean()) # Maths: 86.25, Science: 87.0
print(df.max()) # Maths: 92, Science: 95
print(df.min()) # Maths: 78, Science: 76

# Row-wise (axis=1) -- total/average per student
print(df.sum(axis=1))
# Aman 174 (85+89)
# Bina 178 (90+88)
# Chetan 154 (78+76)
# Dia 187 (92+95)

print(df.mean(axis=1))
# Aman 87.0
# Bina 89.0
# Chetan 77.0
# Dia 93.5
```

Explanation: axis=0 goes DOWN each column (like reading a column). axis=1 goes ACROSS each row (like reading a row). Remember: axis=0 gives one answer per column, axis=1 gives one answer per row.

Q7. Write Python code to handle missing values (NaN) using: (i) isnull() to detect (ii) dropna() to remove (iii) fillna() to replace

[3 Marks]

Answer:

Missing values (NaN) must be handled before performing analysis.

Three key tools for dealing with NaN in Pandas:

isnull() -- DETECT where NaN exists.

dropna() -- REMOVE rows/columns containing NaN.

fillna() -- REPLACE NaN with a meaningful value.

```
import pandas as pd, numpy as np

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

# (i) isnull() -- detect NaN
print(df.isnull())
# Name Maths Science
# 0 False False False
# 1 False True False <-- Bina's Maths is NaN
# 2 False False True <-- Chetan's Science is NaN
# 3 False False False
print(df.isnull().sum()) # Count per column
# Maths 1
# Science 1

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

# (iii) fillna() -- replace NaN with column mean
df['Maths'] = df['Maths'].fillna(df['Maths'].mean())
df['Science'] = df['Science'].fillna(df['Science'].mean())
print(df)
# Bina's Maths filled with mean(85,78,92)=85.0
# Chetan's Science filled with mean(89,88,95)=90.67
```

Explanation: fillna() with the column mean is the most common technique because it preserves the overall average of the data. dropna() is used when the rows with missing data are not important or when missing data is random.

Q8. Write Python code to sort a DataFrame by multiple columns. Explain the use of ascending parameter.

[3 Marks]

Answer:

sort_values() can sort by one or multiple columns.

When sorting by multiple columns: primary sort first, then secondary sort within groups that have the same primary value.

The ascending parameter:

ascending=True (default) --> smallest to largest (A to Z)

ascending=False --> largest to smallest (Z to A)

For multiple columns, pass a list: ascending=[True, False]

```
import pandas as pd

data = {
'Name': ['Aman','Bina','Chetan','Dia','Esha'],
'Class': [12, 11, 12, 11, 12],
'Maths': [85, 90, 78, 92, 85]
}
df = pd.DataFrame(data)

# Sort by single column -- Maths ascending
print(df.sort_values('Maths'))
# Name Class Maths
# 2 Chetan 12 78
# 0 Aman 12 85
# 4 Esha 12 85
# 1 Bina 11 90
# 3 Dia 11 92

# Sort by MULTIPLE columns:
# First by Class (ascending), then by Maths (descending)
print(df.sort_values(['Class','Maths'], ascending=[True, False]))
# Name Class Maths
# 3 Dia 11 92 <-- Class 11 first, highest Maths
# 1 Bina 11 90
# 0 Aman 12 85 <-- Class 12 next, highest Maths
# 4 Esha 12 85
# 2 Chetan 12 78
```

Explanation: Multi-column sorting is very useful in real-world data analysis. Example: Sort employee records by department first, then by salary within each department. The order of columns in the list matters - leftmost column is sorted first.

Q9. Write Python code to demonstrate accessing data using: (i) Single column (ii) Multiple columns (iii) loc[] (iv) iloc[]

[3 Marks]

Answer:

Assume df has students S1-S4 with columns Name, Maths, Science.

```
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'])

# (i) Single column -- returns a Series
print(df['Maths'])
# S1 85
# S2 90
# S3 78
# S4 92

# (ii) Multiple columns -- returns a DataFrame
print(df[['Name', 'Maths']])
# Name Maths
# S1 Aman 85
# S2 Bina 90

# (iii) loc[] -- label-based (inclusive stop)
print(df.loc['S1':'S3', 'Maths':'Science'])
# Maths Science
# S1 85 89
# S2 90 88
# S3 78 76 <-- S3 included

# (iv) iloc[] -- position-based (exclusive stop)
print(df.iloc[0:3, 1:3])
# Maths Science
# S1 85 89
# S2 90 88
# S3 78 76 <-- position 3 not included
```

Explanation: Note the difference: df["Maths"] returns a Series. df[["Maths"]] (double brackets) returns a DataFrame. This is a very commonly tested concept in CBSE exams.

Q10. Explain iterrows() in Pandas. Write code using iterrows() to print each student name and their total marks.

[3 Marks]

Answer:

iterrows() is a method to iterate (loop) over a DataFrame ROW BY ROW.

At each step it returns two values:

1. index -- the row index label (e.g. "S1", "S2")
2. row -- the entire row as a Series object

You can then access individual column values using row["ColumnName"].

When to use iterrows():

- When you need to process each row individually with custom logic.
- When you want to print or log row-by-row results.
- Note: For large datasets, vectorized operations (df.apply) are faster.

```
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'])

# iterrows() -- loops through each row
print('Student Name | Total Marks')
print('-' * 30)
for idx, row in df.iterrows():
total = row['Maths'] + row['Science']
print(f"{row['Name']:12} | {total}")

# Output:
# Student Name | Total Marks
# ----------------------------
# Aman | 174
# Bina | 178
# Chetan | 154
# Dia | 187
```

Explanation: iterrows() is slower than vectorized operations for large data. For simply adding columns, prefer: df["Total"] = df["Maths"] + df["Science"]. Use iterrows() only when you need complex per-row logic that cannot be vectorized.

SECTION D - Long Answer Questions (5 Marks Each)

Q1. Define DataFrame. Explain FOUR different methods to create a DataFrame in Pandas with syntax and example for each.

[5 Marks]

Answer:

Definition: A DataFrame is a TWO-DIMENSIONAL, size-mutable, labelled data structure in Pandas. It is like a table or spreadsheet where:

- Rows are identified by a row index.
- Columns are identified by column labels.
- Each column is a Series object.
- Different columns can hold different data types.

Four Methods to Create a DataFrame:

Method 1: From a Dictionary of Lists (most common)

Method 2: From a List of Dictionaries

Method 3: With Custom Row Index

Method 4: From a 2D NumPy Array

```
import pandas as pd
import numpy as np

# METHOD 1: Dictionary of Lists
# Keys = column names, Values = lists of column data
data1 = {'Name':['Aman','Bina','Chetan'], 'Maths':[85,90,78]}
df1 = pd.DataFrame(data1)
print('Method 1:')
print(df1)
# Name Maths
# 0 Aman 85
# 1 Bina 90
# 2 Chetan 78

# METHOD 2: List of Dictionaries
# Each dict = one row, keys = column names
data2 = [{'Name':'Aman','Maths':85},{'Name':'Bina','Maths':90}]
df2 = pd.DataFrame(data2)
print('Method 2:')
print(df2)
# Name Maths
# 0 Aman 85
# 1 Bina 90

# METHOD 3: With Custom Row Index
# index parameter assigns custom labels to rows
data3 = {'Maths':[85,90,78], 'Science':[89,88,76]}
df3 = pd.DataFrame(data3, index=['Aman','Bina','Chetan'])
print('Method 3:')
print(df3)
# Maths Science
# Aman 85 89
# Bina 90 88
```

```
# Chetan 78 76

# METHOD 4: From 2D NumPy Array
# columns parameter gives column names, index gives row labels
arr = np.array([[85,89],[90,88],[78,76]])
df4 = pd.DataFrame(arr, columns=['Maths','Science'],
index=['Aman','Bina','Chetan'])
print('Method 4:')
print(df4)
# Maths Science
# Aman 85 89
# Bina 90 88
# Chetan 78 76
```

Explanation: Method 1 (dict of lists) is most commonly used in practice. Method 2 (list of dicts) is useful when reading records from a database or API. Method 3 (custom index) is important for meaningful row labels. Method 4 (NumPy array) is used in scientific computing applications.

Q2. Write a complete Python program to: (i) Create a DataFrame of 5 students with Name, Class, Maths and Science (ii) Add Total and Percentage columns (iii) Filter only passed students (Total >= 120 out of 200) (iv) Sort by Percentage in descending order (v) Display final result with rank

[5 Marks]

Answer:

This is a comprehensive program combining DataFrame creation, column operations, filtering, and sorting.

```
import pandas as pd

# (i) Create DataFrame
data = {
'Name': ['Aman', 'Bina', 'Chetan', 'Dia', 'Esha'],
'Class': [12, 11, 12, 11, 12],
'Maths': [85, 90, 55, 92, 48],
'Science': [89, 88, 60, 95, 52]
}
df = pd.DataFrame(data, index=['S1','S2','S3','S4','S5'])
print('Original DataFrame:')
print(df)
# Name Class Maths Science
# S1 Aman 12 85 89
# S2 Bina 11 90 88
# S3 Chetan 12 55 60
# S4 Dia 11 92 95
# S5 Esha 12 48 52

# (ii) Add Total and Percentage columns
df['Total'] = df['Maths'] + df['Science']
df['Percentage'] = (df['Total'] / 200) * 100
print('\nWith Total and Percentage added:')
print(df[['Name','Maths','Science','Total','Percentage']])

# (iii) Filter passed students (Total >= 120)
passed = df[df['Total'] >= 120].copy()
print('\nPassed Students (Total >= 120):')
print(passed[['Name','Total','Percentage']])
# Name Total Percentage
# S1 Aman 174 87.0
# S2 Bina 178 89.0
# S4 Dia 187 93.5

# (iv) Sort by Percentage descending
result = passed.sort_values('Percentage', ascending=False)

# (v) Add Rank and display final result
result['Rank'] = range(1, len(result)+1)
print('\nFinal Result with Rank:')
print(result[['Rank','Name','Class','Total','Percentage']])
# Rank Name Class Total Percentage
# S4 1 Dia 11 187 93.5
```

```
# S2 2 Bina 11 178 89.0
# S1 3 Aman 12 174 87.0

print(f'\nTotal students: {len(df)}')
print(f'Passed: {len(passed)}, Failed: {len(df)-len(passed)}')
```

Explanation: This type of comprehensive program is common in CBSE board exams. Key points: .copy() prevents SettingWithCopyWarning when modifying filtered data. range(1, len+1) creates rank numbers starting from 1.

Q3. Explain with code examples: (i) Vector operations on Series (ii) Arithmetic between two Series with NaN (iii) Renaming columns (iv) Modifying values using condition (v) Transposing a DataFrame

[5 Marks]

Answer:

These are five important operations in Pandas that are frequently tested in CBSE examinations.

```
import pandas as pd

# (i) VECTOR OPERATIONS on Series
# Operations applied element-wise - no loop needed!
s = pd.Series([10, 20, 30, 40], index=['a','b','c','d'])
print(s + 5) # Adds 5 to EVERY element
# a 15
# b 25
# c 35
# d 45
print(s * 2) # Multiplies EVERY element by 2
print(s > 15) # Compares EVERY element: [False,True,True,True]

# (ii) ARITHMETIC with NaN (non-matching index)
s1 = pd.Series([10,20,30], index=['a','b','c'])
s2 = pd.Series([1, 2, 3], index=['a','b','d'])
print(s1 + s2)
# a 11.0 (matched: 10+1)
# b 22.0 (matched: 20+2)
# c NaN ('c' not in s2)
# d NaN ('d' not in s1)

# (iii) RENAMING COLUMNS
data = {'Maths':[85,90],'Science':[89,88],'Class':[12,11]}
df = pd.DataFrame(data, index=['Aman','Bina'])
df = df.rename(columns={'Maths':'Math_Marks','Science':'Sci_Marks'})
print(df.columns.tolist())
# ['Math_Marks', 'Sci_Marks', 'Class']

# (iv) MODIFYING VALUES USING CONDITION
# Give 5 bonus marks to Class 12 students in Maths
df.loc[df['Class'] == 12, 'Math_Marks'] += 5
print(df)
# Math_Marks Sci_Marks Class
# Aman 90 89 12 <-- was 85, now 90
# Bina 90 88 11 <-- unchanged

# (v) TRANSPOSING
data2 = {'Maths':[85,90,78],'Science':[89,88,76]}
df2 = pd.DataFrame(data2, index=['Aman','Bina','Chetan'])
print('Original:', df2.shape) # (3, 2)
print('Transposed:', df2.T.shape) # (2, 3)
print(df2.T)
# Aman Bina Chetan
# Maths 85 90 78
```

```
# Science 89 88 76
```

Explanation: Vector operations (i) are much faster than loops for large datasets. The NaN behaviour (ii) is a key distinguishing feature of Pandas Series. rename() (iii) is non-destructive - always returns a new DataFrame. Condition-based modification (iv) is called "fancy indexing" or "boolean masking".

Q4. Write a complete Python program demonstrating: (i) Creating Series with all three methods (ii) Slicing Series by label and position (iii) Filtering with multiple conditions (iv) sort_values() and sort_index() (v) Series attributes: index, values, dtype, size, shape

[5 Marks]

Answer:

This program covers all major Series operations tested in CBSE exams.

```
import pandas as pd

# (i) THREE METHODS to create Series
s_list = pd.Series([85,90,78,92,88]) # From list
s_dict = pd.Series({'Maths':95,'Science':88}) # From dict
s_cust = pd.Series([85,90,78,92], # Custom index
index=['Aman','Bina','Chetan','Dia'])
print('Series with custom index:')
print(s_cust)
# Aman 85
# Bina 90
# Chetan 78
# Dia 92

# (ii) SLICING
# Label-based (loc style) -- stop IS included
print(s_cust['Bina':'Dia'])
# Bina 90
# Chetan 78
# Dia 92 <-- included

# Position-based -- stop NOT included
print(s_cust[1:4])
# Bina 90
# Chetan 78
# Dia 92 (position 4 not included)

# (iii) FILTERING with multiple conditions
# Students with marks between 85 and 91 (inclusive)
filtered = s_cust[(s_cust >= 85) & (s_cust <= 91)]
print('Marks between 85 and 91:')
print(filtered)
# Aman 85
# Bina 90

# (iv) SORTING
print('Sort by values (ascending):')
print(s_cust.sort_values())
# Chetan 78
# Aman 85
# Bina 90
# Dia 92

print('Sort by index (alphabetical):')
print(s_cust.sort_index())
```

```
# Aman 85
# Bina 90
# Chetan 78
# Dia 92

# (v) SERIES ATTRIBUTES
print('Values:', s_cust.values) # array([85, 90, 78, 92])
print('Index:', s_cust.index.tolist()) # ['Aman','Bina','Chetan','Dia']
print('dtype:', s_cust.dtype) # int64
print('size:', s_cust.size) # 4
print('shape:', s_cust.shape) # (4,)
print('ndim:', s_cust.ndim) # 1
```

Explanation: For CBSE exams remember: Label slicing ["Bina":"Dia"] includes Dia. Position slicing [1:4] excludes position 4. This is a very commonly asked question in board exams.

Q5. A school stores student data in a Pandas DataFrame. Write complete Python code to: (i) Create DataFrame with some NaN values (ii) Display all DataFrame attributes (iii) Fill NaN with column mean (iv) Add Grade column using map() (v) Display final sorted result

[5 Marks]

Answer:

This is an end-to-end data analysis program combining all major DataFrame operations.

```
import pandas as pd
import numpy as np

# (i) Create DataFrame WITH NaN values
data = {
'Name': ['Aman', 'Bina', 'Chetan', 'Dia', 'Esha'],
'Maths': [85, np.nan, 78, 92, 88],
'Science': [89, 88, np.nan, 95, 84],
'English': [80, 75, 88, 91, 82]
}
df = pd.DataFrame(data, index=['S1','S2','S3','S4','S5'])
print('Original (with NaN):')
print(df)

# (ii) Display DataFrame Attributes
print('Shape:', df.shape) # (5, 4)
print('Columns:', df.columns.tolist())# ['Name','Maths','Science','English']
print('Index:', df.index.tolist()) # ['S1','S2','S3','S4','S5']
print('ndim:', df.ndim) # 2
print('size:', df.size) # 20 (5 rows x 4 cols)
print('dtypes:')
print(df.dtypes)
# Name object
# Maths float64 (float because NaN makes int become float)
# Science float64
# English int64
print('Missing values:')
print(df.isnull().sum())
# Maths 1
# Science 1

# (iii) Fill NaN with column MEAN
df['Maths'] = df['Maths'].fillna(round(df['Maths'].mean(), 1))
df['Science'] = df['Science'].fillna(round(df['Science'].mean(), 1))
print('After filling NaN with mean:')
print(df[['Name','Maths','Science']])
# S2 Bina's Maths = mean(85,78,92,88) = 85.75 ~ 85.8
# S3 Chetan's Sci = mean(89,88,95,84) = 89.0

# Calculate Total after filling NaN
df['Total'] = df['Maths'] + df['Science'] + df['English']
df['Pct'] = (df['Total'] / 300 * 100).round(1)

# (iv) Add Grade using map()
def assign_grade(pct):
```

```
if pct >= 90: return 'A+'
elif pct >= 80: return 'A'
elif pct >= 70: return 'B'
elif pct >= 60: return 'C'
else: return 'F'

df['Grade'] = df['Pct'].map(assign_grade)

# (v) Sort by Percentage descending and display
final = df.sort_values('Pct', ascending=False)
print('Final Result (sorted by Percentage):')
print(final[['Name','Total','Pct','Grade']])
# Name Total Pct Grade
# S4 Dia 278.0 92.7 A+
# S1 Aman 258.0 86.0 A
# S5 Esha 254.0 84.7 A
# S2 Bina 248.8 82.9 A
# S3 Chetan 255.0 85.0 A
```

Explanation: This program demonstrates real-world data analysis workflow: Load data -> Check for missing values -> Clean data -> Compute results -> Grade -> Sort. Notice how column becomes float64 when NaN is present - this is because NaN is a float value in Python.

Marking Scheme Summary

Section	Type	No. of Questions	Marks Each	Total Mark
A	Multiple Choice Questions (MCQ)	15	1	15
B	Theory / Descriptive Questions	12	2	24
C	Short Answer (Theory + Code)	10	3	30
D	Long Answer Questions	5	5	25
	GRAND TOTAL	42 Questions		94 Marks