Important Instructions for Pandas-Based Assessments

🔍 1. Follow Function Names Strictly

- **Function names must exactly match** the problem statement.
- is **case-sensitive**.

✔️ Example:

```
def create_visit_df(self, visit_data: list) -> pd.DataFrame:  # Correct
```

❌ Wrong:

```
def Create_Visit_DF(self):  # Will cause test case failure
```

🔍 2. Be Careful with Return Types

- **Always return a DataFrame**, NOT a list or tuple unless explicitly asked.
- Return types must exactly match what is expected:
  - DataFrame → return `pd.DataFrame`
  - Dictionary → return `dict`
  - Integer/Float/String → return as per the requirement.

❌ **Don't print — just return**:

```
return df  # Correct
```

```
print(df)  # Wrong — tests won’t capture printed output
```

🔍 3. Variable Names Matter (Case-Sensitive)

- Use the exact **column names given in the question**:

```
df["PatientID"]  # Correct
df["patientid"]  # Wrong — will raise KeyError
```

✔️ **Common mistake**: Using "charges" instead of "Charges" — case matters.

🔍 4. Read "Implementation Flow" Carefully

Many students fail because they skip points like:

- Use .groupby() for summaries.
- Use .reset_index() when grouping.
- Rename columns if asked (df.rename()).
- Round numeric columns using .round().
- Filtering or sorting as per instructions.

❗ If the question says **“Round to 2 decimals”** — do it!

```
df["NewCol"] = df["OldCol"].round(2)
```

🔍 5. Use Pandas Functions — Not Loops

✔️ You **must use**:

- .groupby(), .mean(), .sum()

- .reset_index(), .rename()
- .loc[], .isin(), .dropna()
- .sort_values()

❌ Avoid loops:

```
for i in df:  # Wrong — use Pandas functions instead.
```

🔍 6. Rounding/Formatting Must Be Exact

✔️ When asked to round:

```
df["CostPerMinute"] = (df["Charges"] / df["Duration"]).round(2)
```

❌ Missing this causes **test failures**.

🔍 7. Avoid Hardcoding Values

❌ Wrong:

```
df["Total Charges"] = 3300.0  # Hardcoded
```

✔️ Right:

```
df.groupby("PatientID")["Charges"].sum().reset_index()
```

✔️ **Never assume sample data — make code dynamic for any input!**

🔍 8. Validation Logic is Mandatory if Asked

✔️ Example:

```
if df.empty:
    return pd.DataFrame()  # Handle empty DataFrame if required.
```

✔️ Use .dropna() or .fillna() if specified.

🔍 9. Condition-based Filtering — Use Pandas Logic

✔️ Example:

```
df.loc[df["Charges"] > 1000]
```

❌ Wrong:

```
for row in df:  # Manual loops = bad Pandas code.
```

🔍 10. Sorting Must Match the Requirement

✔️ Example:

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

❗ If sorting is missed — the output order will be wrong & test case fails.

🔍 11. Always Return Final Output — NOT Print

✔️ Return DataFrame or dict — as per question.

❌ Wrong:

```
print(df)  # The checker cannot capture print output.
```

✔️ Right:

```
return df
```

⚠️ Most Common Mistakes by Previous Batches

Mistake	What Happens
Wrong Function Name	Checker fails — 0 marks
Returned list instead of DataFrame	Type error or fail
Missed `.reset_index()`	Output mismatch
Ignored `.round()`	Floating point error
Used loop instead of groupby	Inefficient, wrong
Extra print	Checker gets None — fails
Used wrong column name	KeyError or empty DataFrame
Missed sorting	Output order mismatch

🎯 Final Pro Tips

✅ **Read the Problem twice.**
✅ Return **exact format** (DataFrame, dict, etc.).
✅ Use **Pandas functions — not loops**.
✅ Handle missing/null data if asked.
✅ Round or format columns properly.
✅ **No hardcoding** — your code must work for any DataFrame.
✅ Check sorting and filtering as per instructions.

📝 Example: Wrong vs Right

❌ Wrong:

```
def get_total_charges(df):
    return 3300.0  # Hardcoded, wrong!
```

✔️ Right:

```
def total_charges_per_patient(self, df: pd.DataFrame) -> pd.DataFrame:
    result = df.groupby("PatientID")["Charges"].sum().reset_index()
    result.rename(columns={"Charges": "Total Charges"}, inplace=True)
    return result
```

⚠️ Pytest or Test Failures Reasons:

Reason	Why it Fails
Print used instead of return	Checker cannot see print output
Hardcoded value	Works only for sample — fails for real
Missed `.reset_index()`	Columns misaligned
Column names wrong case	KeyError
Missed rounding	Floating point mismatch
Loop instead of Pandas method	Wrong result or slow

🎯 Golden Rule:

"Write for the auto-tester, not for your eyes."

✔️ No print — only return.
✔️ Exact structure, column names, formats.
✔️ Pandas functions only — no manual code.

Problem Statement:
You are managing the data for a retail chain. The business records product inventory and sales transactions. Your job is to analyze this data using Pandas.

There are two main DataFrames:

1. Product Master Data (Product Info)

Column	Description
ProductID	Unique Product Identifier
ProductName	Name of the product
Category	Category: 'Electronics', 'Clothing', 'Grocery'
UnitPrice	Price per unit (may have missing values)
Stock	Current stock available

2. Sales Transaction Data

Column	Description
SaleID	Unique Sale Identifier
ProductID	Foreign Key to Product Master
QuantitySold	Units sold
SaleDate	Date of sale (may have duplicates)
CustomerType	'Regular' or 'Member'

📌 Tasks to Implement:

Section 1: Data Preparation & Cleaning
Create DataFrames from given lists.

Fill missing 'UnitPrice' with category-wise average.

Remove duplicate sales records.

Replace 'Regular'/'Member' in 'CustomerType' with 'R' and 'M'.

Section 2: Merging & Integration

Merge Sales Data with Product Master on 'ProductID'.

Calculate 'TotalSaleAmount' = 'UnitPrice' * 'QuantitySold' in the merged DataFrame.

Find Most Selling Product (by Total Quantity Sold).

Find the Category with Maximum Total Sales Amount.

Section 3: Filtering & Aggregation

Filter Sales where 'QuantitySold' > 2 and 'Category' is 'Electronics'.

Group by 'ProductName' and calculate Total Sales Quantity and Total Sales Amount.

Group by 'SaleDate' and find the day with Maximum Total Sales.

Section 4: Sorting, Ranking & Statistics

Sort Products by 'Stock' descending — to find low-stock products.

Find Top 3 Products by 'TotalSaleAmount'.

Compute Average, Min, Max 'UnitPrice' per Category.

Categorize products based on 'Stock':

Stock >= 50 → 'High'

20–49 → 'Medium'

<20 → 'Low'

Section 5: Pivot & Advanced Analysis

Create a Pivot Table: 'Category' vs 'CustomerType' showing Total QuantitySold.

Reindex the Product Master DataFrame with custom index starting from 1000.

Section 6: General Operations

Convert Merged DataFrame to list of dictionaries.

Export Sales Summary (Product, TotalSaleAmount) as CSV string.

Find the Product with Maximum 'TotalSaleAmount' (similar to 'most delayed route').

Practice 1:

```
import pandas as pd

class MuseumVisitorDashboard:
    def __init__(self):
        pass

    def create_visit_df(self, visit_data: list) -> pd.DataFrame:
        """
        Converts a list of museum visit records into a pandas DataFrame.
        """
        columns = ["VisitorID", "Exhibit", "Duration", "TicketCost"]
        return pd.DataFrame(visit_data, columns=columns)
    def total_ticket_cost_per_visitor(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Returns total ticket cost per visitor by grouping the DataFrame by 'VisitorID'.
        """
        total_cost = df.groupby("VisitorID")["TicketCost"].sum().reset_index()
        total_cost.rename(columns={"TicketCost": "TotalTicketCost"}, inplace=True)
        return total_cost

    def add_cost_per_minute(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Adds a 'CostPerMinute' column calculated as TicketCost/Duration rounded to 2 decimal
places.
        """
        df["CostPerMinute"] = (df["TicketCost"] / df["Duration"]).round(2)
        return df

    def frequent_visitors(self, df: pd.DataFrame, n: int) -> pd.DataFrame:
        """
        Returns all records of visitors who have visited more than 'n' times.
        """
        visit_counts = df["VisitorID"].value_counts()
        frequent_ids = visit_counts[visit_counts > n].index
        result_df = df[df["VisitorID"].isin(frequent_ids)]
        return result_df

    def clean_and_sort_visits(self, df: pd.DataFrame) -> pd.DataFrame:
        """
        Removes rows with missing values and sorts the remaining rows by 'TicketCost'
descending.
        """
        clean_df = df.dropna()
        sorted_df = clean_df.sort_values(by="TicketCost",
ascending=False).reset_index(drop=True)
        return sorted_df
```

Employee Training Assessment Dashboard

Objective:

Analyze employee training results to summarize scores, determine performance levels, and identify top performers.

📌 Operations to Implement

✅ 1. Create Training Result DataFrame

Create a DataFrame from raw training assessment records.

Function Prototype:

```
create_training_df(self, training_data: list) -> pd.DataFrame
```

Example Input:

```
create_training_df([
    ["Team A", "John", 85, "Pass"],
    ["Team B", "Alice", 90, "Pass"],
    ["Team A", "Mike", 60, "Fail"]
])
```

Expected Output Columns:

- Team
- EmployeeName

- Score
- Result

✅ 2. Compute Total and Average Scores per Team

Compute total and average scores for each team.

Function Prototype:

7. team_score_summary(self, df: pd.DataFrame) -> pd.DataFrame

Expected Output Columns:

- Team
- TotalScore
- AverageScore

✅ 3. Add Result Points Column

Add a new column 'Points' — assign **3 points for 'Pass'** and **0 points for 'Fail'**.

Function Prototype:

8. add_result_points(self, df: pd.DataFrame) -> pd.DataFrame

Expected Output Columns:

- Team

- EmployeeName
- Score
- Result
- Points

✅ 4. Filter Employees with High Scores

Return records of employees who scored **more than n marks**.

Function Prototype:

9. filter_high_scorers(self, df: pd.DataFrame, n: int) -> pd.DataFrame

Expected Output Columns:

- Team
- EmployeeName
- Score
- Result

✅ 5. Compute Pass Rate per Team

Calculate **Pass Rate** (percentage of employees who passed) for each team.

Function Prototype:

10. compute_pass_rate(self, df: pd.DataFrame) -> pd.DataFrame

Expected Output Columns:

- Team
- PassRate

✅ 6. Get Top N Teams by Total Points

List the **top N teams** based on their total points.

Function Prototype:

11. top_teams_by_points(self, df: pd.DataFrame, n: int) -> pd.DataFrame

Expected Output Columns:

- Team
- TotalPoints

✅ Summary of Pandas Functions to Use (Same Logic as Sports Match Dashboard):

Task	Pandas Techniques Required
Create DataFrame	pd.DataFrame()
Total & Average Score per Team	.groupby(), .sum(), .mean()

Add Result Points Column	.apply(), .map() or np.where()
Filter High Scores	df[df['Score'] > n]
Compute Pass Rate per Team	.groupby(), .value_counts(), calculations
Top N Teams by Total Points	.groupby(), .sum(),

Problem Set 3:

Analyze library book borrowing patterns, build member profiles, merge borrowing records, and generate insights on borrowing behavior.

📌 Operations to Implement

✅ 1. Create Borrowing Record DataFrame

Create a DataFrame containing borrowing records.

Function Prototype:

```
create_borrow_df(self, borrow_data: list) -> pd.DataFrame
```

Example Input:

```
create_borrow_df([
    [1001, "B001", "2024-06-01"],
    [1002, "B003", "2024-06-02"],
    [1001, "B002", "2024-06-05"]
])
```

Expected Output Columns:

- MemberID
- BookID
- BorrowDate

✅ 2. Create Member Profile DataFrame

Create a DataFrame containing library member details.

Function Prototype:

```
create_member_df(self, member_data: list) -> pd.DataFrame
```

Expected Output Columns:

- MemberID
- MemberName
- Location

✅ 3. Merge Member Info into Borrowing Records

Merge the **member profile DataFrame with the borrowing records** based on MemberID.

Function Prototype:

merge_member_info(self, borrow_df, member_df) -> pd.DataFrame

Expected Output Columns:

- MemberID
- BookID
- BorrowDate
- MemberName
- Location

✅ 4. Get Frequent Borrowers

Identify members who have borrowed **more than once**.

Function Prototype:

get_frequent_borrowers(self, borrow_df) -> pd.DataFrame

Expected Output Columns:

- MemberID

- BorrowCount

✅ 5. Calculate Total Books Borrowed per Member

Compute the **total number of books borrowed by each member**.

Function Prototype:

```
calculate_total_books_borrowed(self, borrow_df) -> pd.DataFrame
```

Expected Output Columns:

- MemberID
- TotalBooksBorrowed

✅ 6. Get Location-wise Active Members

List **each location along with the number of unique active members**.

Function Prototype:

```
location_wise_members(self, merged_df) -> pd.DataFrame
```

Expected Output Columns:

- Location
- ActiveMembers

✅ Summary of Pandas Functions to Use (Same as E-commerce Analyzer):

Task	Required Pandas Techniques
Create DataFrames	pd.DataFrame()
Merge Profiles	pd.merge()
GroupBy & Aggregation	.groupby(), .count(), .nunique()
Filtering Frequent Borrowers	.value_counts(), filtering
Location-wise Aggregation	.groupby('Location'), .nunique()

Pandas Cheat-Sheet Code Explanation

---

1. Creating DataFrame

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

- Creates a DataFrame from a dictionary.
- Each key becomes a column.

---

2. Basic Data Exploration

```
df.head()           # First 5 rows
df.tail(2)         # Last 2 rows
df.shape           # (rows, columns)
df.columns         # List of column names
df.dtypes          # Data type of each column
```

- Useful to understand data structure, shape, and schema.

---

3. Selecting Columns and Rows

```
df['Name']               # Single column (Series)
df[['Name', 'Age']]      # Multiple columns (DataFrame)
df.loc[1]                # Row by label/index
df.iloc[2]               # Row by position
```

- Important for row/column slicing MCQs.

---

4. Filtering Data

```
df[df['Age'] > 28]  # Filter rows where Age > 28
df[(df['Age'] > 28) & (df['Department'] == 'IT')]
```

- Uses Boolean indexing.
- Common in conditions-based MCQs.

---

5. Aggregation Operations

```
df['Salary'].sum()    # Total salary
df['Age'].mean()      # Average age
df['Salary'].max()    # Highest salary
```

- Frequently asked in aggregation function MCQs.

---

6. GroupBy Aggregations

```
df.groupby('Department')['Salary'].mean()  # Average salary by
department
df.groupby('Department').agg({'Age':'mean', 'Salary':'sum'})
```

- Important for MCQs on data summarization.

---

7. Sorting

```
df.sort_values(by='Age', ascending=False)
```

- Sorting data by column values.
- Ascending/Descending order often questioned.

---

8. Handling Missing Data

```
df.isna().sum()              # Total NaNs per column
df.dropna()                  # Drop rows with NaNs
df.fillna({'Age': 29})       # Replace NaNs in 'Age' with 29
```

- Most common MCQ area — handling NULLs.

---

9. Apply Function

```
df['AgePlusTen'] = df['Age'].apply(lambda x: x + 10)
```

- Applying custom functions on columns.
- MCQs often ask about `.apply()` usage.

---

10. Merging DataFrames

```
pd.merge(df, df2, on='Department', how='left')
```

- Joins two DataFrames on a key column.
- `how=` parameter (left, right, inner) is a common MCQ.

---

11. Concatenation

```
pd.concat([df, df3], ignore_index=True)
```

- Stacking DataFrames vertically or horizontally.
- `axis=0` (row-wise), `axis=1` (column-wise).

---

12. Pivot Table

```
pd.pivot_table(df, index='Department', values='Salary',
aggfunc='mean')
```

- Reshaping and summarizing data.
- MCQ: **Pivot vs GroupBy difference**.

---

13. Renaming Columns

```
df.rename(columns={'Salary': 'MonthlySalary'}, inplace=True)
```

- Renaming column names.
- Useful for schema adjustment in assessments.

---

14. Removing Duplicates

```
df.drop_duplicates(subset=['Name'])
```

- Removes duplicate rows based on specific columns.

---

15. Changing Data Types

```
df['Age'] = df['Age'].astype('float')
```

- Casting columns to another data type (int, float, str).

---

16. String Operations

```
df['Department'].str.upper()
```

- String transformations in columns.
- `.str` accessor MCQs are common.

---

17. Resetting and Setting Index

```
df.reset_index(drop=True, inplace=True)  # Remove index
df.set_index('Name', inplace=True)       # Set column as index
```

- Index management — essential for multi-index handling.

---

18. Exporting Data

```
df.to_csv('output.csv', index=False)
```

- Save DataFrame to CSV.
- `.to_excel()` for Excel output.

✅ Important MCQ Areas from This Explanation:

Concept	MCQ/Assessment Focus
DataFrame Creation	From dict/list
Column/Row Selection	`.loc[]`, `.iloc[]`, slicing
Filtering Conditions	Boolean masks, `&`, `
Aggregation & GroupBy	`sum()`, `mean()`, `agg()`
Missing Data Handling	`.isna()`, `.dropna()`, `.fillna()`
Apply / Lambda Functions	Column-wise transformations
Merge/Join	`pd.merge()`, join types (`left`, `inner`)
Pivot Tables	Index, columns, values, aggfunc usage
Duplicates Removal	`.drop_duplicates()`
Type Conversion	`.astype()`
String Operations	`.str.upper()`, `.str.contains()`
Index Management	`.reset_index()`, `.set_index()`
Exporting Data	`.to_csv()`, `.to_excel()`