SCENARIO:

The Data Cycle: From Collection to Decision-Making

The **data cycle** is a structured process that transforms raw data into actionable insights. It consists of the following key phases:

1. **Data Collection** begins with gathering data from various sources, such as sensors, databases, APIs, or user inputs. The quality of data at this stage directly impacts the analysis.
2. **Data Processing** – Once collected, raw data is structured, formatted, and integrated into a usable format. This may involve data transformation, type conversion, and preliminary calculations. (--> our ***.CSV** file)
3. **Data Cleaning** – Errors, inconsistencies, and missing values are identified and corrected. This step ensures that the dataset is reliable and free from inaccuracies, making it suitable for analysis.
4. **Communication of Visual Reports** – The cleaned data is analyzed and presented through dashboards, charts, and reports. Visual representation helps stakeholders quickly grasp trends, correlations, and patterns.
5. **Decision-Making** – Insights derived from the data drive informed decisions. Whether in business, healthcare, or automation, data-driven decision-making enhances efficiency and strategic planning.

This cycle repeats continuously as new data emerges, ensuring continuous improvement and adaptation to changing conditions.

This step-by-step guide:

1. Loads and cleans the dataset using **Pandas**.
2. Handles missing values using explicit functions (avoiding lambda functions for clarity).
3. Extracts useful date components.
4. Aggregates sales data.
5. Visualizes the results with **matplotlib**.
6. Utilizes **numpy** for statistical calculations.

1. Importing Libraries

```
import pandas as pd              # For data manipulation and analysis.
import numpy as np               # For numerical operations and array handling.
import matplotlib.pyplot as plt # For data visualization.
```

- **Pandas**: Handles data structures and operations for manipulating numerical tables.
- **numpy**: Offers support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions.
- **matplotlib**: Provides an interface for creating static, animated, and interactive visualizations.

2. Loading the Dataset

- `# Load the CSV file into a DataFrame and convert the 'TransactionDate' column to datetime.`
- `df = pd.read_csv("sales_data.csv", parse_dates=["TransactionDate"])`

- **pd.read_csv()**: Reads data from a CSV file.
- **parse_dates**: Automatically converts the specified column(s) into datetime objects.

---

3. Displaying Column Names

- `print("Columns in the dataset:", df.columns)`

- **df.columns**: Returns the list of column names, which is useful to verify the dataset structure.

---

4. Renaming Columns Without Using Lambda

Instead of a lambda to strip whitespace, define an explicit function:

- `def clean_column_name(col_name):`
- `    return col_name.strip()`
- `df.rename(columns=clean_column_name, inplace=True)`

- **clean_column_name()**: A function that removes any leading or trailing spaces.
- **df.rename()**: Applies the function to each column name.

---

5. Filling Missing Values in Categorical Columns

For example, if your dataset contains columns like `"Region"` or `"Segment"`:

- `if "Region" in df.columns:`
- `    region_mode = df["Region"].mode()[0]`
- `    df["Region"] = df["Region"].fillna(region_mode)`
-
- `if "Segment" in df.columns:`

- `    segment_mode = df["Segment"].mode()[0]`
- `    df["Segment"] = df["Segment"].fillna(segment_mode)`

- **df["Region"].mode()[0]**: Finds the most frequent value in the "Region" column.
- **fillna()**: Fills missing values with the mode.

---

6. Filling Missing Numeric Values (e.g., Prices) with the Median Per Product

Here, we define a function to fill missing prices for each product without using a lambda:

```
def fill_missing_with_median(series):
    return series.fillna(series.median())
if "Price" in df.columns and "Product" in df.columns:
    df["Price"] = df.groupby("Product")["Price"].transform(fill_missing_with_median)
```

- **fill_missing_with_median()**: Takes a pandas Series, computes its median, and fills missing values with that median.
- **groupby() & transform()**: Apply the function on each group (here, grouped by "Product").

---

7. Dropping Rows Where a Critical Column Is Missing

If the product name is essential, drop rows with missing values in the `"Product"` column:

- `df.dropna(subset=["Product"], inplace=True)`

- **dropna(subset=[...])**: Removes rows where the specified column(s) have missing values.

---

8. Saving the Cleaned Dataset

Once cleaned, save the DataFrame to a new CSV file:

- `df.to_csv("cleaned_sales_data.csv", index=False)`

- **index=False**: Ensures that the DataFrame index is not saved as a separate column.

---

9. Displaying Dataset Summary

Print a summary to check data types and count missing values:

- `print("Final Dataset Summary:")`
- `print(df.info())`
- `print("\nMissing Values After Cleaning:\n", df.isnull().sum())`

- **df.info()**: Displays information about the DataFrame including data types and non-null counts.
- **df.isnull().sum()**: Gives a count of missing values per column.

---

10. Reloading the Cleaned Dataset

If needed, reload your cleaned data for further analysis:

- `df = pd.read_csv("cleaned_sales_data.csv", parse_dates=["TransactionDate"])`

---

11. Converting 'TransactionDate' to DateTime Format

Ensure that your date column is in the correct format:

- `df["TransactionDate"] = pd.to_datetime(df["TransactionDate"])`

- **pd.to_datetime()**: Converts the column into datetime format for time-series analysis.

---

12. Extracting Year and Month from the Date

Create new columns for year and month for further analysis:

- `df["Year"] = df["TransactionDate"].dt.year`
- `df["Month"] = df["TransactionDate"].dt.month`

- **dt.year** and **dt.month**: Extract the year and month components.

13. Displaying the Updated Dataset

View the first few rows and a summary:

- `print(df.head())`
- `print(df.info())`

- **df.head()**: Displays the first five rows to verify that new columns are added correctly.

---

14. Aggregating Total Sales by Region

Using Pandas' grouping functionality:

- `region_sales = df.groupby("Region")["Price"].sum().sort_values(ascending=False)`
- `print("Total Sales by Region:")`
- `print(region_sales)`

- **groupby() & sum()**: Groups the data by region and sums the sales (price).
- **sort_values()**: Sorts the results in descending order.

---

15. Plotting Total Sales by Region with matplotlib

Visualize the aggregated sales using a bar chart:

- `plt.figure(figsize=(10, 5))`
- `plt.bar(region_sales.index, region_sales.values, color='skyblue')`
- `plt.title("Total Sales by Region")`
- `plt.xlabel("Region")`
- `plt.ylabel("Total Sales")`
- `plt.xticks(rotation=45)`
- `plt.tight_layout()  # Adjust layout to prevent clipping of labels`
- `plt.show()`

- **plt.bar()**: Creates a bar chart.
- **plt.xticks(rotation=45)**: Rotates the x-axis labels for readability.

16. Additional Analysis Using numpy

You can also perform statistical analysis with numpy. For instance, calculating the overall average sale price:

- `average_sales = np.mean(df["Price"].dropna())`
- `print("Overall Average Sales: $", round(average_sales, 2))`

- **np.mean()**: Computes the mean of the "Price" column after dropping any missing values.