Supermart Grocery Sales - Retail Analytics Dataset

1. Introduction

1.1 Project Overview

This project demonstrates the process of Supermart Grocery Sales Data Cleaning, Analysis, and Visualization. We'll walk through the following steps:

1.2 About Dataset

The Supermart Grocery Sales dataset is a comprehensive collection of sales data from a grocery store. The data spans sales transactions from 2016 to 2017, offering insights into customer purchasing patterns and product trends. The dataset includes information such as order ID, customer name, category, subcategory, city, order date, region, sales, discount, profit, and state. This dataset will be analyzed using Python and visualized with matplotlib and seaborn to uncover trends and patterns in grocery sales. The purpose of this dataset is to apply data analysis and visualization techniques to gain insights into the retail industry. The cleaned data and visualizations can be found below.

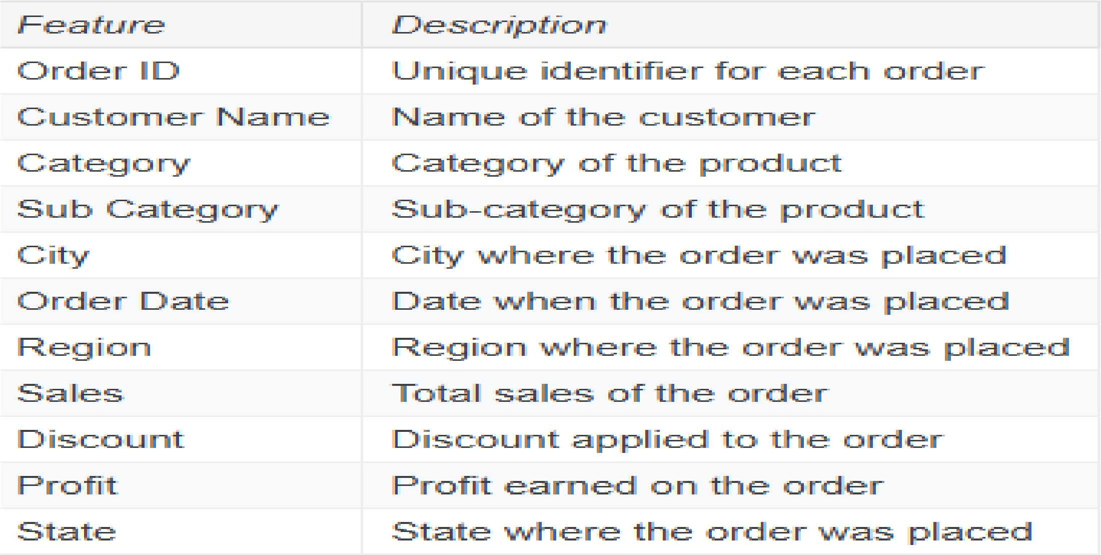
1.3 Motivation

Accurate sales prediction models are crucial in the retail industry, facilitating:

- Understanding customer purchasing patterns and product trends to inform inventory management and supply chain strategies.
- Identifying opportunities to increase sales and customer loyalty through data-driven insights.
- Enhancing the shopping experience by providing personalized product recommendations based on purchase history and preferences.
- Optimizing product placement, pricing, and marketing efforts through data analysis and visualization.
- Gaining a competitive edge in the retail industry by leveraging data insights to drive business decisions.

2. Data Description

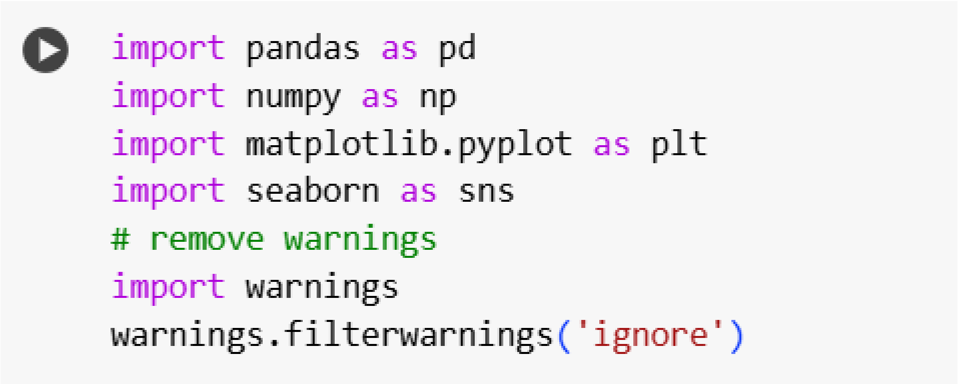
Our dataset comprises the following key features:



Feature	Description
Order ID	Unique identifier for each order
Customer Name	Name of the customer
Category	Category of the product
Sub Category	Sub-category of the product
City	City where the order was placed
Order Date	Date when the order was placed
Region	Region where the order was placed
Sales	Total sales of the order
Discount	Discount applied to the order
Profit	Profit earned on the order
State	State where the order was placed

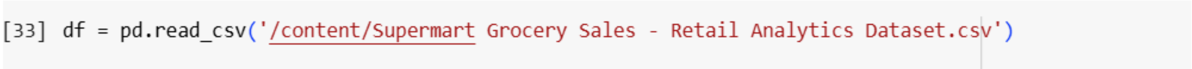
3. Steps Followed in the EDA:

3.1 Import Libraries

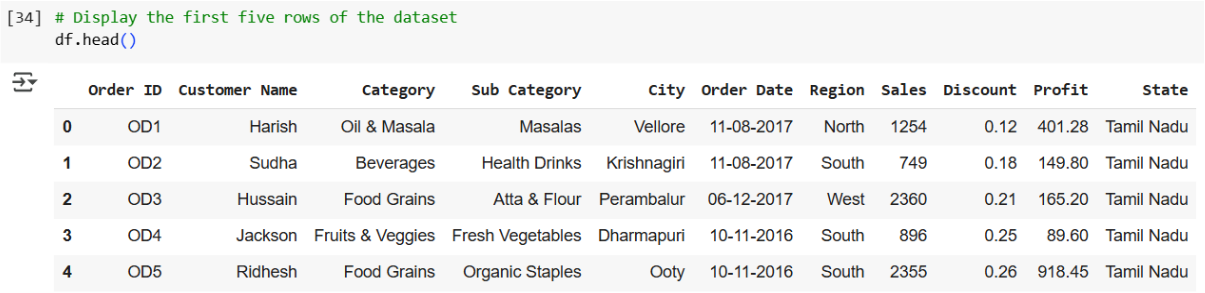


```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# remove warnings
import warnings
warnings.filterwarnings('ignore')
```

3.2 Load the Dataset



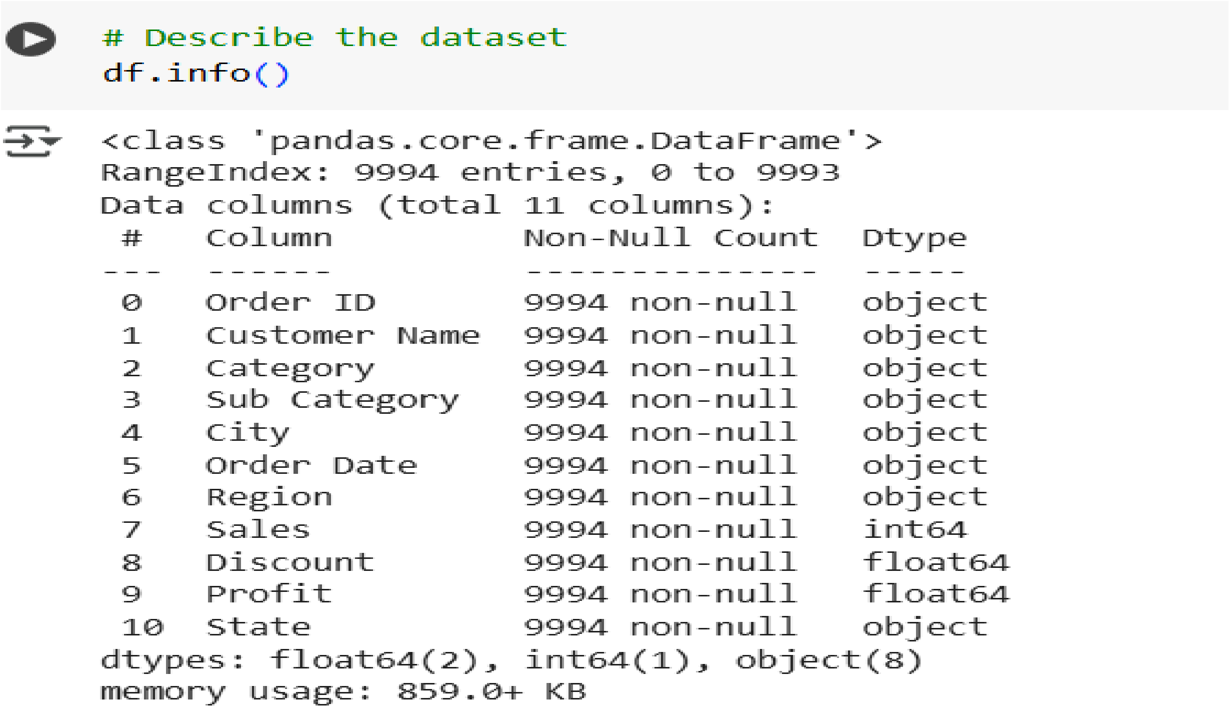
```
[33] df = pd.read_csv('/content/Supermart Grocery Sales - Retail Analytics Dataset.csv')
```



```
[34] # Display the first five rows of the dataset
     df.head()
```

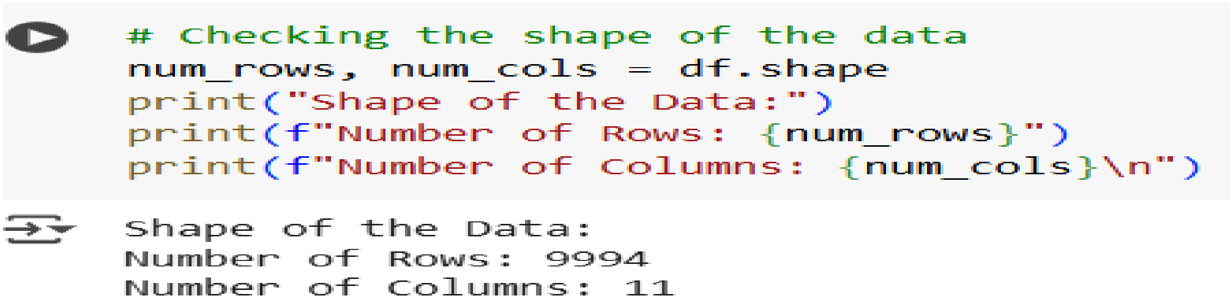
	Order ID	Customer Name	Category	Sub Category	City	Order Date	Region	Sales	Discount	Profit	State
0	OD1	Harish	Oil & Masala	Masalas	Vellore	11-08-2017	North	1254	0.12	401.28	Tamil Nadu
1	OD2	Sudha	Beverages	Health Drinks	Krishnagiri	11-08-2017	South	749	0.18	149.80	Tamil Nadu
2	OD3	Hussain	Food Grains	Atta & Flour	Perambalur	06-12-2017	West	2360	0.21	165.20	Tamil Nadu
3	OD4	Jackson	Fruits & Veggies	Fresh Vegetables	Dharmapuri	10-11-2016	South	896	0.25	89.60	Tamil Nadu
4	OD5	Ridhesh	Food Grains	Organic Staples	Ooty	10-11-2016	South	2355	0.26	918.45	Tamil Nadu

3.3 Understand the Dataset



```python
# Describe the dataset
df.info()
```

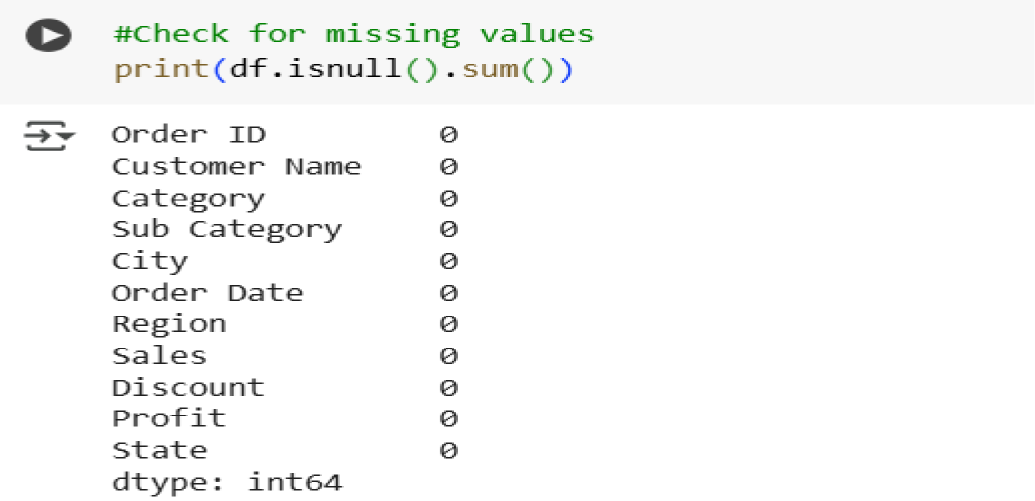
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9994 entries, 0 to 9993
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Order ID        9994 non-null   object 
 1   Customer Name   9994 non-null   object 
 2   Category        9994 non-null   object 
 3   Sub Category    9994 non-null   object 
 4   City            9994 non-null   object 
 5   Order Date      9994 non-null   object 
 6   Region          9994 non-null   object 
 7   Sales           9994 non-null   int64  
 8   Discount        9994 non-null   float64
 9   Profit          9994 non-null   float64
 10  State           9994 non-null   object 
dtypes: float64(2), int64(1), object(8)
memory usage: 859.0+ KB
```



```python
# Checking the shape of the data
num_rows, num_cols = df.shape
print("Shape of the Data:")
print(f"Number of Rows: {num_rows}")
print(f"Number of Columns: {num_cols}\n")
```

```
Shape of the Data:
Number of Rows: 9994
Number of Columns: 11
```

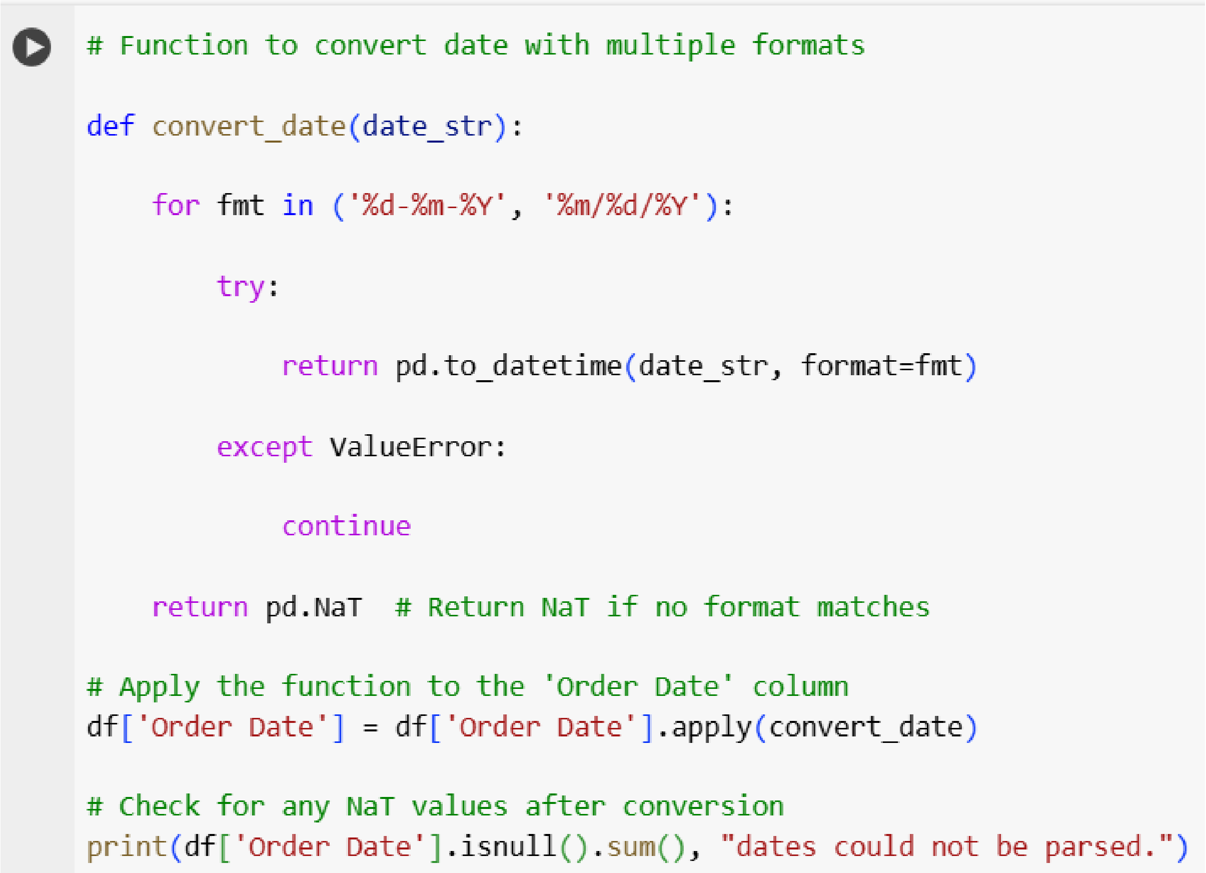
3.4 Data Preprocessing



```
#Check for missing values
print(df.isnull().sum())
```

```
Order ID         0
Customer Name    0
Category         0
Sub Category     0
City             0
Order Date       0
Region           0
Sales            0
Discount         0
Profit           0
State            0
dtype: int64
```

- **Converted Order Date to datetime format.**



```
# Function to convert date with multiple formats

def convert_date(date_str):

    for fmt in ('%d-%m-%Y', '%m/%d/%Y'):

        try:

            return pd.to_datetime(date_str, format=fmt)

        except ValueError:

            continue

    return pd.NaT  # Return NaT if no format matches

# Apply the function to the 'Order Date' column
df['Order Date'] = df['Order Date'].apply(convert_date)

# Check for any NaT values after conversion
print(df['Order Date'].isnull().sum(), "dates could not be parsed.")
```

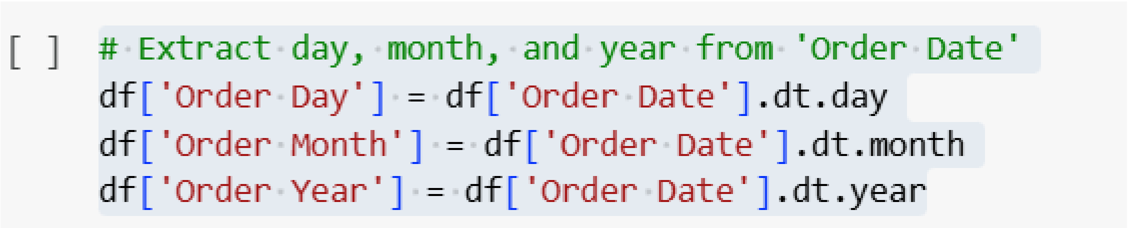
Defines a function convert_date(date_str):

- It tries to convert a date string using two common formats:
  - '%d-%m-%Y' → e.g., 31-12-2024
  - '%m/%d/%Y' → e.g., 12/31/2024
- If both formats fail, it returns NaT (Not a Time), which represents a missing datetime values.

Observation:

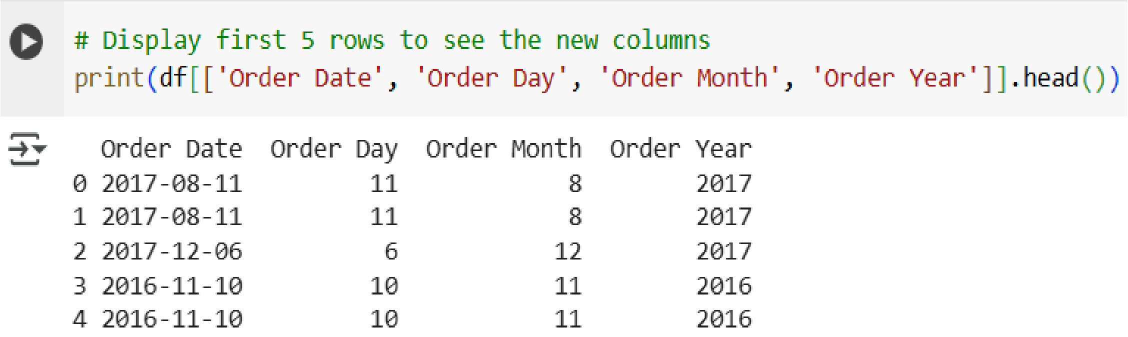
- This means every date in the 'Order Date' column was successfully converted to a valid datetime format.
- No NaT (missing/invalid dates) were found.

Note: Date conversion is essential for performing time-based analysis (e.g., monthly sales trends, yearly growth).



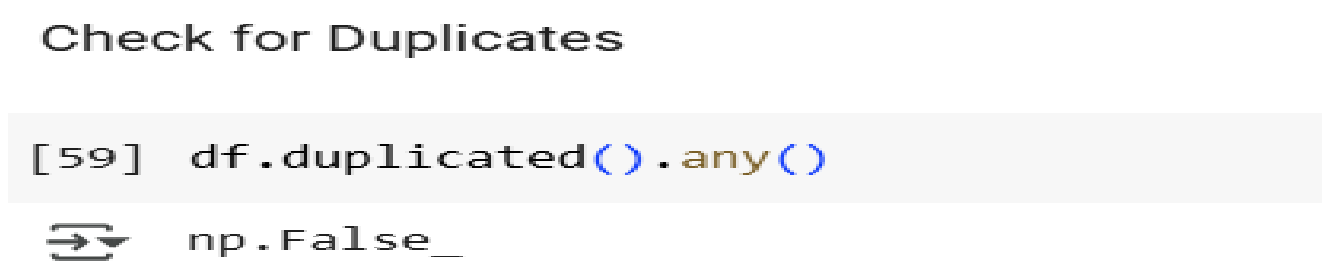
```
[ ]  # Extract day, month, and year from 'Order Date'
     df['Order Day'] = df['Order Date'].dt.day
     df['Order Month'] = df['Order Date'].dt.month
     df['Order Year'] = df['Order Date'].dt.year
```

- **df['Order Day'] = df['Order Date'].dt.day**
  - Extracts the day (1 to 31) from each date.
  - Example: '2023-06-15' → 15
- **df['Order Month'] = df['Order Date'].dt.month**
  - Extracts the month (1 to 12).
  - Example: '2023-06-15' → 6
- **df['Order Year'] = df['Order Date'].dt.year**
  - Extracts the year (e.g., 2023).
  - Example: '2023-06-15' → 2023



```
# Display first 5 rows to see the new columns
print(df[['Order Date', 'Order Day', 'Order Month', 'Order Year']].head())
```

```
  Order Date  Order Day  Order Month  Order Year
0 2017-08-11         11            8        2017
1 2017-08-11         11            8        2017
2 2017-12-06          6           12        2017
3 2016-11-10         10           11        2016
4 2016-11-10         10           11        2016
```



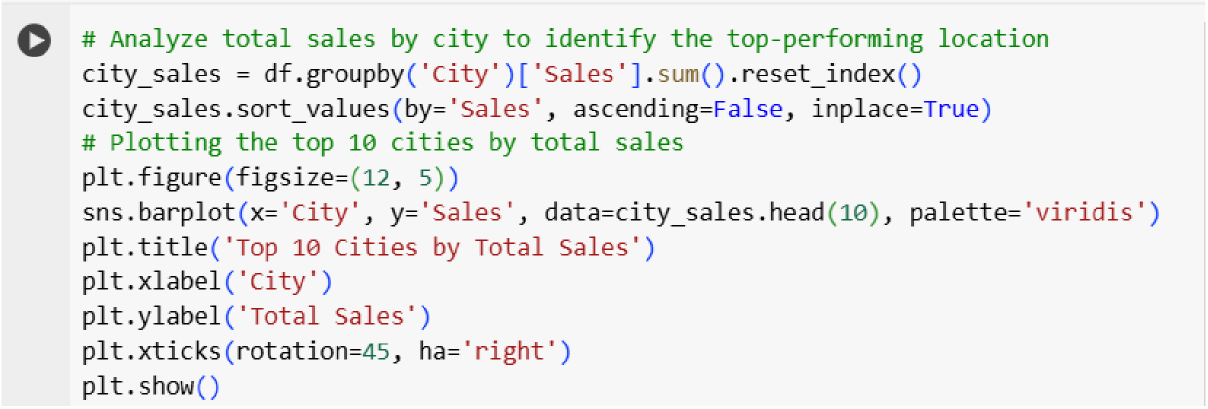
Check for Duplicates

```
[59] df.duplicated().any()
```

```
np.False_
```

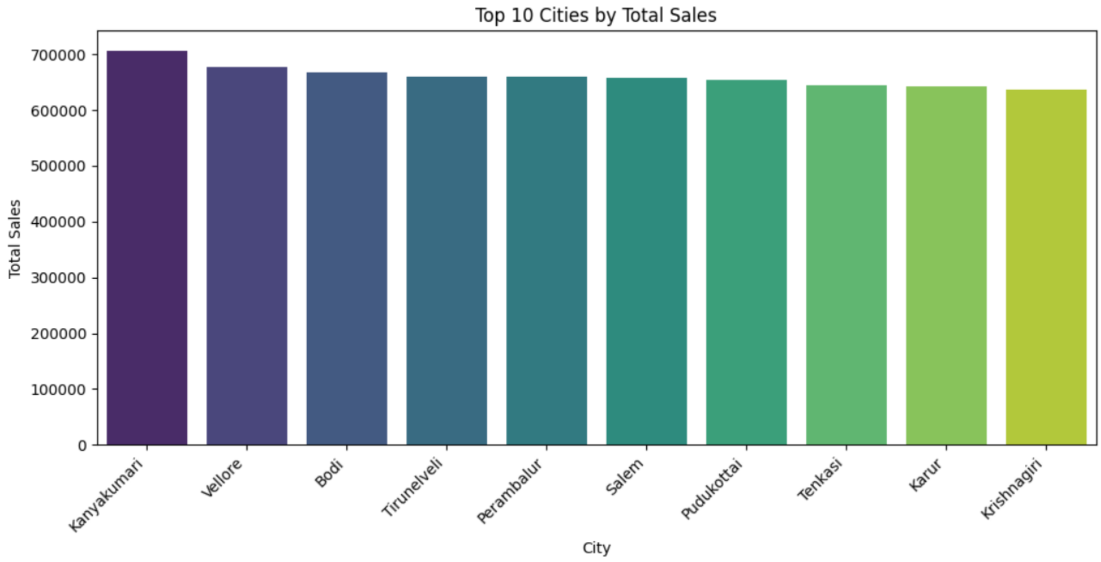
3.5 Exploratory Data Analysis

3.5.1 Top 10 Cities by Sales Performance



```
# Analyze total sales by city to identify the top-performing location
city_sales = df.groupby('City')['Sales'].sum().reset_index()
city_sales.sort_values(by='Sales', ascending=False, inplace=True)
# Plotting the top 10 cities by total sales
plt.figure(figsize=(12, 5))
sns.barplot(x='City', y='Sales', data=city_sales.head(10), palette='viridis')
plt.title('Top 10 Cities by Total Sales')
plt.xlabel('City')
plt.ylabel('Total Sales')
plt.xticks(rotation=45, ha='right')
plt.show()
```

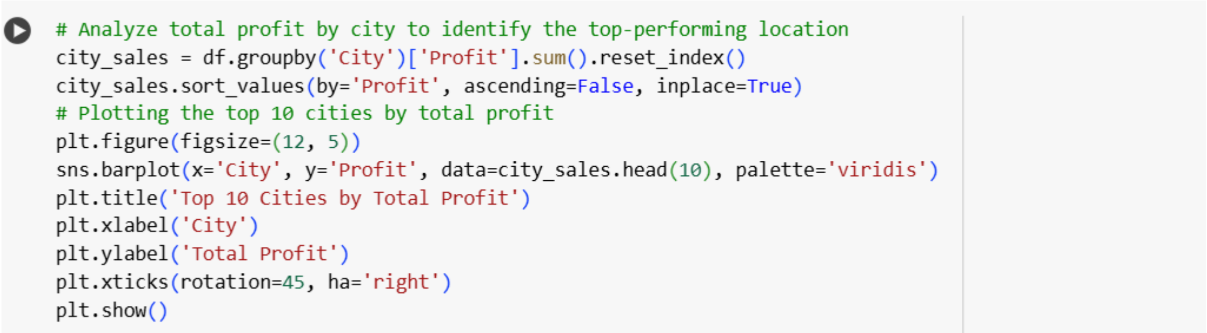
- Group by city and sum up sales
  - **city_sales = df.groupby('City')['Sales'].sum().reset_index()**
    - This groups data by the 'City' column.
    - For each city, it calculates the total sales.
    - .reset_index() flattens the result into a DataFrame.
- Sort by sales in descending order
  - **city_sales.sort_values(by='Sales', ascending=False, inplace=True)**
    - Sorts the cities from highest to lowest total sales**.**
- Plotting the top 10 cities
  - **plt.figure(figsize=(12, 5))**
  - **sns.barplot(x='City', y='Sales', data=city_sales.head(10), palette='viridis')**
    - Creates a barplot of the top 10 cities using the seaborn library.
    - The viridis palette gives a nice color gradient from dark to light.
- **plt.title('Top 10 Cities by Total Sales')**
- **plt.xlabel('City')**
- **plt.ylabel('Total Sales')**
- **plt.xticks(rotation=45, ha='right') # Rotate x-axis labels for better readability**
- **plt.show()**
  - Adds title and axis labels.
  - Rotates city names to avoid overlap and improve readability.



Observation:

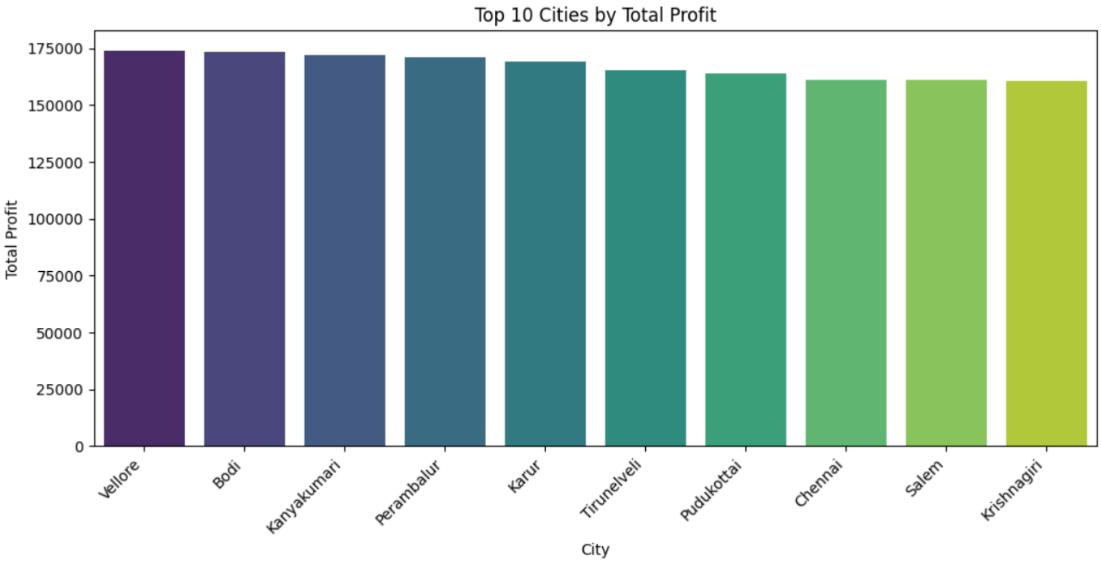
- Kanyakumari had the highest total sales among all cities in the dataset.
- It's followed closely by Vellore, Bodi, and others.
- All top 10 cities have total sales in the range of ~650,000 to 710,000+.

3.5.2 Top 10 Cities by Profit Performance



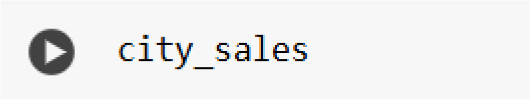
```
# Analyze total profit by city to identify the top-performing location
city_sales = df.groupby('City')['Profit'].sum().reset_index()
city_sales.sort_values(by='Profit', ascending=False, inplace=True)
# Plotting the top 10 cities by total profit
plt.figure(figsize=(12, 5))
sns.barplot(x='City', y='Profit', data=city_sales.head(10), palette='viridis')
plt.title('Top 10 Cities by Total Profit')
plt.xlabel('City')
plt.ylabel('Total Profit')
plt.xticks(rotation=45, ha='right')
plt.show()
```

- **city_sales = df.groupby('City')['Profit'].sum().reset_index()**
- **city_sales.sort_values(by='Profit', ascending=False, inplace=True)**
  - Groups the data by city.
  - Computes total profit per city.
  - Sorts cities in descending order of profit.
- **sns.barplot(x='City', y='Profit', data=city_sales.head(10), palette='viridis')**
  - Plots the top 10 cities based on total profit.

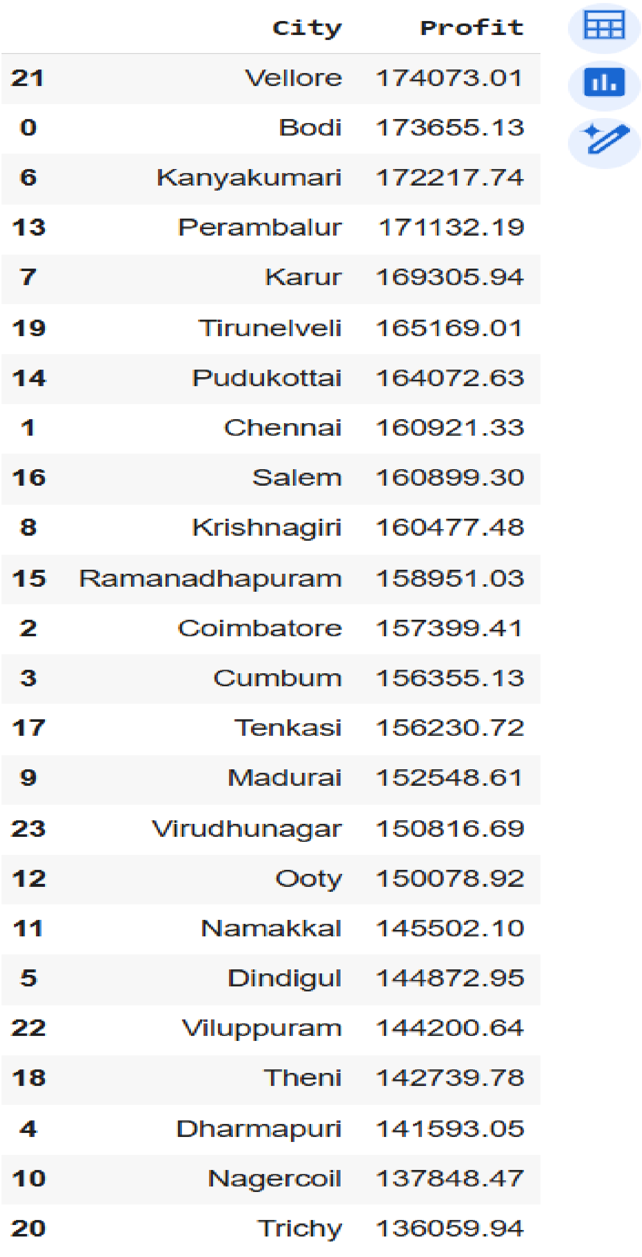


Observation:

- Vellore is the most profitable city.
- Bodi and Kanyakumari also show high profit.
- This plot highlights which cities are most financially beneficial, not just where sales are high.



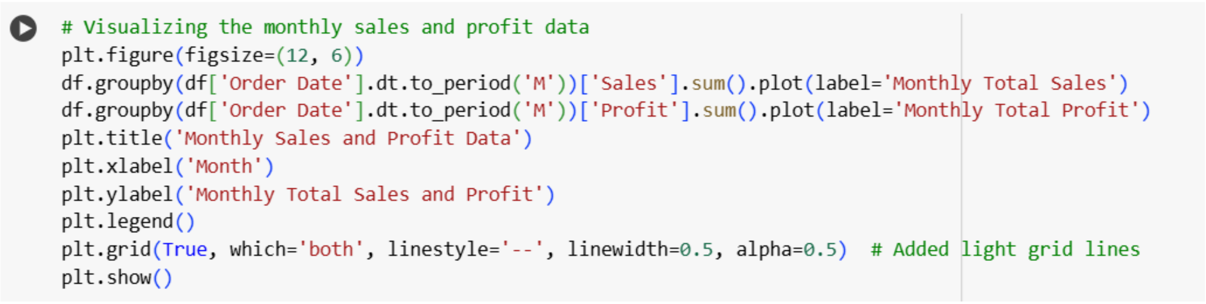
```
city_sales
```



	City	Profit
21	Vellore	174073.01
0	Bodi	173655.13
6	Kanyakumari	172217.74
13	Perambalur	171132.19
7	Karur	169305.94
19	Tirunelveli	165169.01
14	Pudukottai	164072.63
1	Chennai	160921.33
16	Salem	160899.30
8	Krishnagiri	160477.48
15	Ramanadhapuram	158951.03
2	Coimbatore	157399.41
3	Cumbum	156355.13
17	Tenkasi	156230.72
9	Madurai	152548.61
23	Virudhunagar	150816.69
12	Ooty	150078.92
11	Namakkal	145502.10
5	Dindigul	144872.95
22	Viluppuram	144200.64
18	Theni	142739.78
4	Dharmapuri	141593.05
10	Nagercoil	137848.47
20	Trichy	136059.94

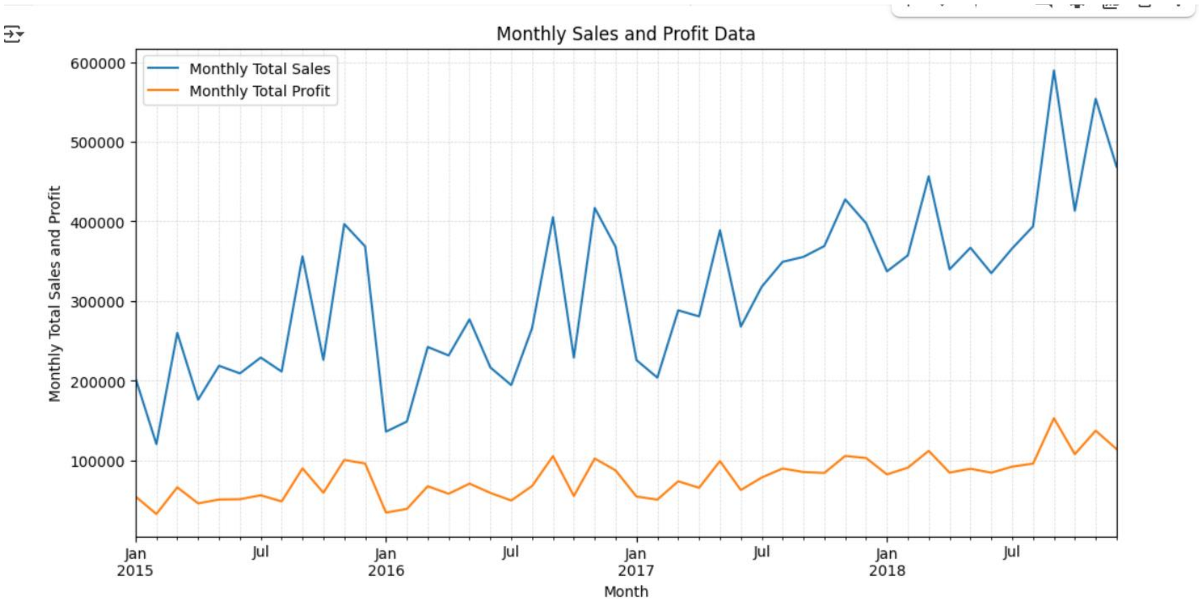
Observation: The output shows all cities ranked by total profit, in descending order.

3.5.3 Sales Trend Over Time



```
# Visualizing the monthly sales and profit data
plt.figure(figsize=(12, 6))
df.groupby(df['Order Date'].dt.to_period('M'))['Sales'].sum().plot(label='Monthly Total Sales')
df.groupby(df['Order Date'].dt.to_period('M'))['Profit'].sum().plot(label='Monthly Total Profit')
plt.title('Monthly Sales and Profit Data')
plt.xlabel('Month')
plt.ylabel('Monthly Total Sales and Profit')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.5)  # Added light grid lines
plt.show()
```

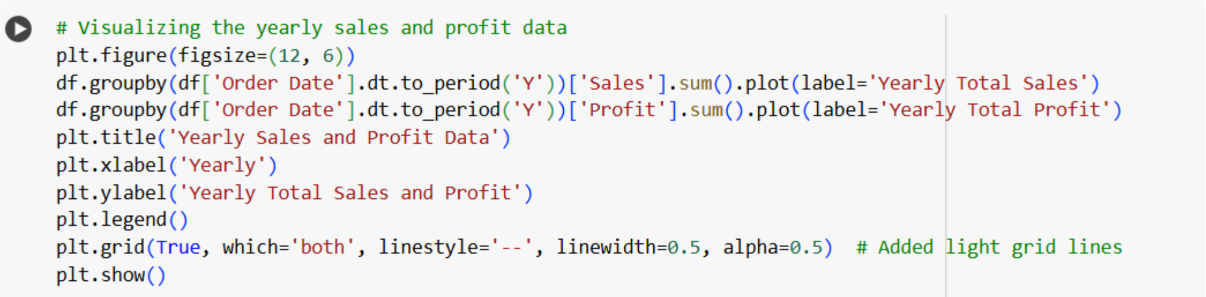
- **plt.figure(figsize=(12, 6))**
  - Sets the size of the figure to 12x6 inches.
- **df.groupby(df['OrderDate'].dt.to_period('M'))['Sales'].sum().plot(label='Monthly Total Sales')**
  - Groups the data by month (using .dt.to_period('M')).
  - Sums the 'Sales' for each month.
  - Plots the monthly total sales with a label.
- **df.groupby(df['OrderDate'].dt.to_period('M'))['Profit'].sum().plot(label='Monthly Total Profit')**
  - Sums the 'Profit' for each month.
- **plt.title('Monthly Sales and Profit Data')**
- **plt.xlabel('Month')**
- **plt.ylabel('Monthly Total Sales and Profit')**
  - Sets the plot title and axes labels.
- **plt.legend()**
  - Shows the legend with labels from the plot() calls.
- **plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.5)**
  - Adds light grid lines to improve readability.
- **plt.show()**
  - Displays the plot.



Observation:

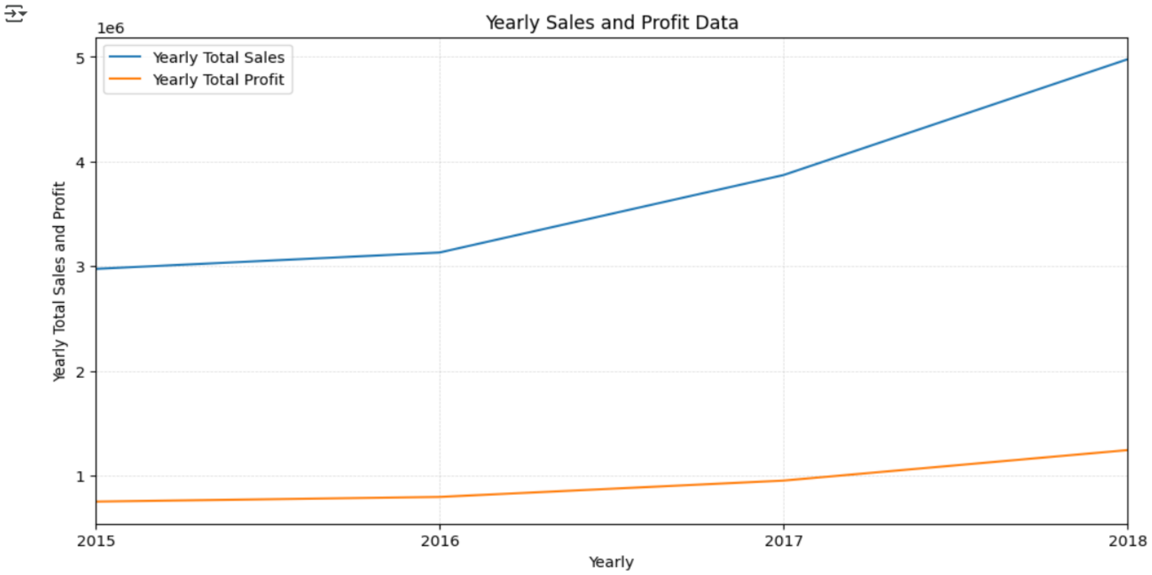
- **X-axis:** Time in months (from Jan 2015 to around the end of 2018).
- **Y-axis:** Numeric values representing monthly total sales (blue line) and profit (orange line).

- Sales fluctuate monthly with seasonal patterns, and generally increase over time.
- Profit also increases slightly over time but is much lower than total sales.
- Both lines show occasional spikes and drops.



```
# Visualizing the yearly sales and profit data
plt.figure(figsize=(12, 6))
df.groupby(df['Order Date'].dt.to_period('Y'))['Sales'].sum().plot(label='Yearly Total Sales')
df.groupby(df['Order Date'].dt.to_period('Y'))['Profit'].sum().plot(label='Yearly Total Profit')
plt.title('Yearly Sales and Profit Data')
plt.xlabel('Yearly')
plt.ylabel('Yearly Total Sales and Profit')
plt.legend()
plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.5)  # Added light grid lines
plt.show()
```

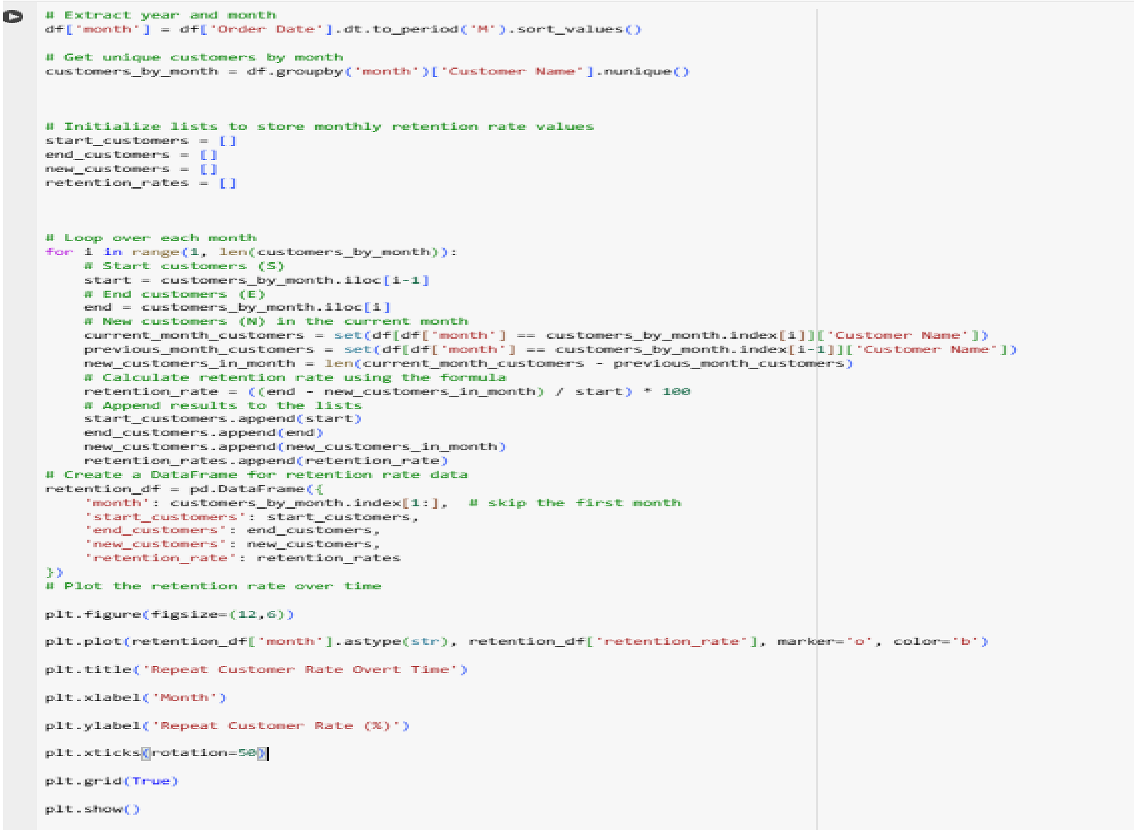
- **plt.figure(figsize=(12, 6))**
  - Creates a new figure of size 12 by 6 inches.
- **df.groupby(df['OrderDate'].dt.to_period('Y'))['Sales'].sum().plot(label='Yearly Total Sales')**
  - Groups the DataFrame df by year using dt.to_period('Y').
  - Aggregates total 'Sales' per year.
  - Plots the result with label Yearly Total Sales.
- **df.groupby(df['OrderDate'].dt.to_period('Y'))['Profit'].sum().plot(label='Yearly Total Profit')**
  - Aggregates total 'profit' per year.
- **plt.title('Yearly Sales and Profit Data')**
- **plt.xlabel('Yearly')**
- **plt.ylabel('Yearly Total Sales and Profit')**
- **plt.legend()**
  - Adds title, X/Y axis labels, and legend.
- **plt.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.5)**
- **plt.show()**
  - Adds light dashed grid lines and displays the plot.



Observation:

- X-axis: Years (2015 to 2018).
- Y-axis: Total sales and profit values for each year.
- Blue Line (Sales): Gradual increase in total sales year-over-year. Around 3 million in 2015 to 5 million in 2018.
- Orange Line (Profit): Also increasing, from under 800k in 2015 to about 1.3 million in 2018.
- Both sales and profits are steadily increasing, which suggests healthy business growth over the four-year span.
- The growth in sales is stronger than in profit, implying either increased operational costs or lower profit margins in later years.

3.5.4 Rate of Repeat Customers Over Time



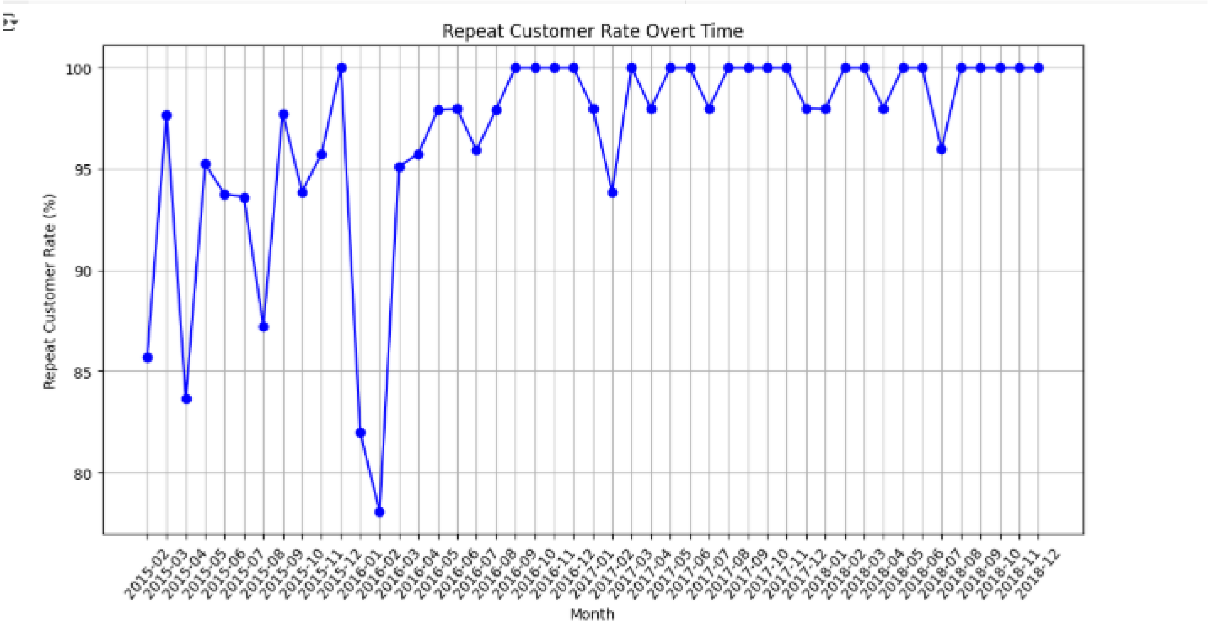
```python
# Extract year and month
df['month'] = df['Order Date'].dt.to_period('M').sort_values()

# Get unique customers by month
customers_by_month = df.groupby('month')['Customer Name'].nunique()



# Initialize lists to store monthly retention rate values
start_customers = []
end_customers = []
new_customers = []
retention_rates = []



# Loop over each month
for i in range(1, len(customers_by_month)):
    # Start customers (S)
    start = customers_by_month.iloc[i-1]
    # End customers (E)
    end = customers_by_month.iloc[i]
    # New customers (N) in the current month
    current_month_customers = set(df[df['month'] == customers_by_month.index[i]]['Customer Name'])
    previous_month_customers = set(df[df['month'] == customers_by_month.index[i-1]]['Customer Name'])
    new_customers_in_month = len(current_month_customers - previous_month_customers)
    # Calculate retention rate using the formula
    retention_rate = ((end - new_customers_in_month) / start) * 100
    # Append results to the lists
    start_customers.append(start)
    end_customers.append(end)
    new_customers.append(new_customers_in_month)
    retention_rates.append(retention_rate)
# Create a DataFrame for retention rate data
retention_df = pd.DataFrame({
    'month': customers_by_month.index[1:],  # skip the first month
    'start_customers': start_customers,
    'end_customers': end_customers,
    'new_customers': new_customers,
    'retention_rate': retention_rates
})
# Plot the retention rate over time

plt.figure(figsize=(12,6))

plt.plot(retention_df['month'].astype(str), retention_df['retention_rate'], marker='o', color='b')

plt.title('Repeat Customer Rate Overt Time')

plt.xlabel('Month')

plt.ylabel('Repeat Customer Rate (%)')

plt.xticks(rotation=50)

plt.grid(True)

plt.show()
```

This code calculates and plots monthly customer retention rates — i.e., the percentage of customers in a given month who were not new (they also existed in the previous month).

- **Extract Month**: Adds a 'month' column by converting 'Order Date' to period (month).
- **Unique Customers by Month**: Uses .groupby() and .nunique() to get the number of unique customers per month.
- **Retention Calculation**:
    - Iterates over each month (starting from the second).
    - Compares current and previous month customer sets.
    - Calculates retention rate:

$$\text{Retention Rate} = \left(\frac{\text{End Customers} - \text{New Customers}}{\text{Start Customers}}\right) \times 100$$

- **Store & Plot**:
    - Stores monthly retention data in a DataFrame.
    - Plots retention over time with months on the x-axis and repeat customer rate (%) on the y-axis.

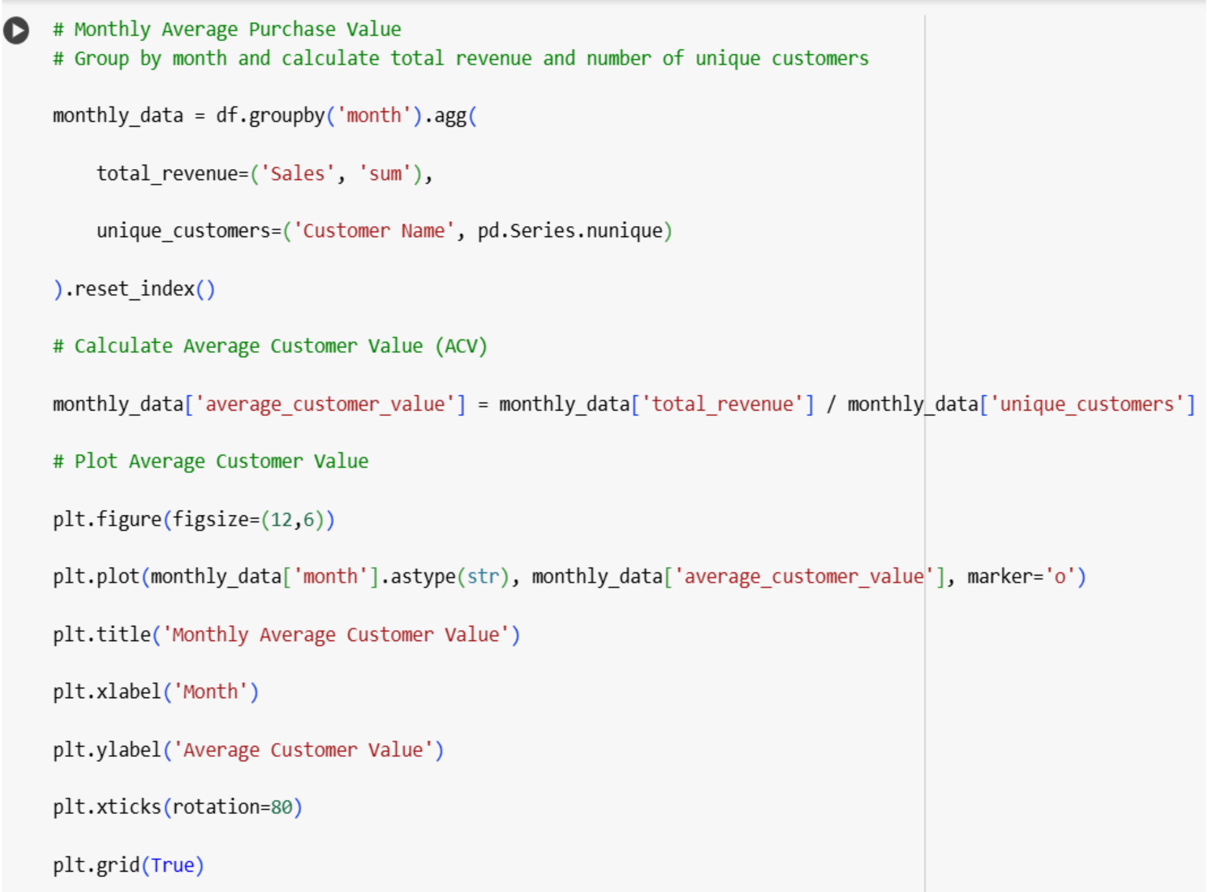


Observation:

- Shows fluctuations in repeat customer rate from 2015 to 2018.
- Generally high retention (95–100%) after 2016, suggesting strong customer loyalty.
- Some early dips (e.g., mid-2015), possibly due to seasonal effects or customer acquisition pushes.

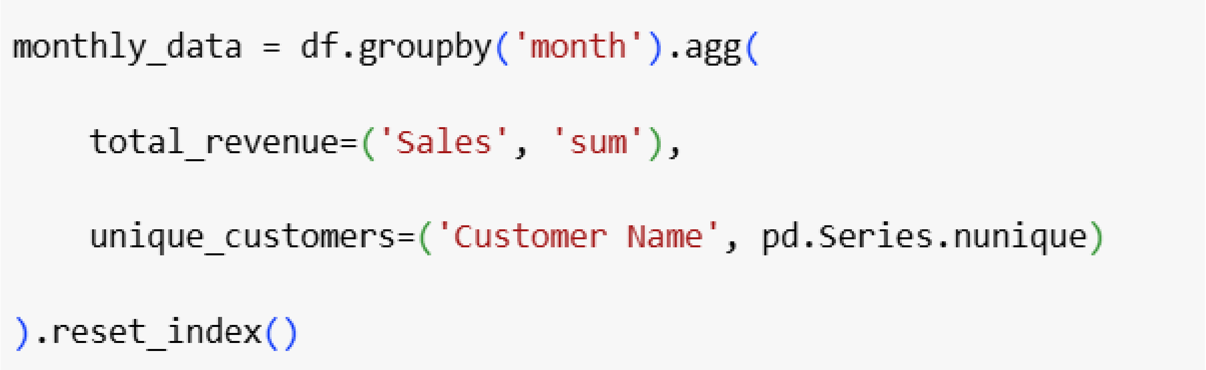
4. Average Purchase Value

4.1 Monthly Average Purchase Value



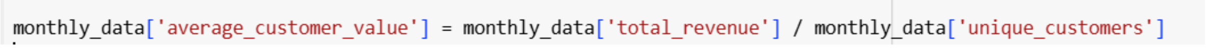
```
# Monthly Average Purchase Value
# Group by month and calculate total revenue and number of unique customers

monthly_data = df.groupby('month').agg(

    total_revenue=('Sales', 'sum'),

    unique_customers=('Customer Name', pd.Series.nunique)

).reset_index()

# Calculate Average Customer Value (ACV)

monthly_data['average_customer_value'] = monthly_data['total_revenue'] / monthly_data['unique_customers']

# Plot Average Customer Value

plt.figure(figsize=(12,6))

plt.plot(monthly_data['month'].astype(str), monthly_data['average_customer_value'], marker='o')

plt.title('Monthly Average Customer Value')

plt.xlabel('Month')

plt.ylabel('Average Customer Value')

plt.xticks(rotation=80)

plt.grid(True)
```

This Code Explains:



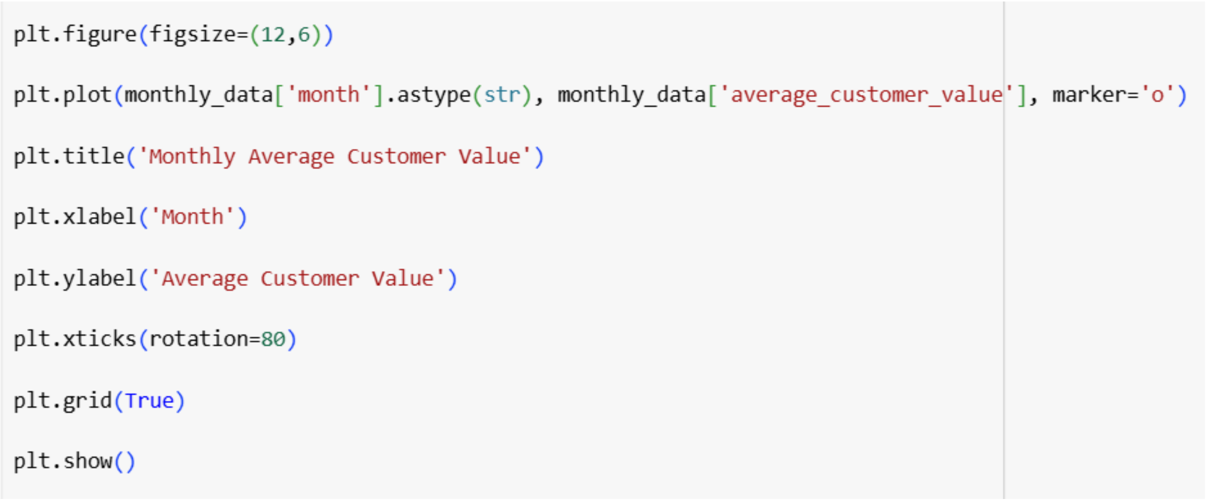
```
monthly_data = df.groupby('month').agg(

    total_revenue=('Sales', 'sum'),

    unique_customers=('Customer Name', pd.Series.nunique)

).reset_index()
```

- The dataset is grouped by month.
- For each month:
    - total_revenue: Sum of sales.
    - unique_customers: Count of distinct customers.



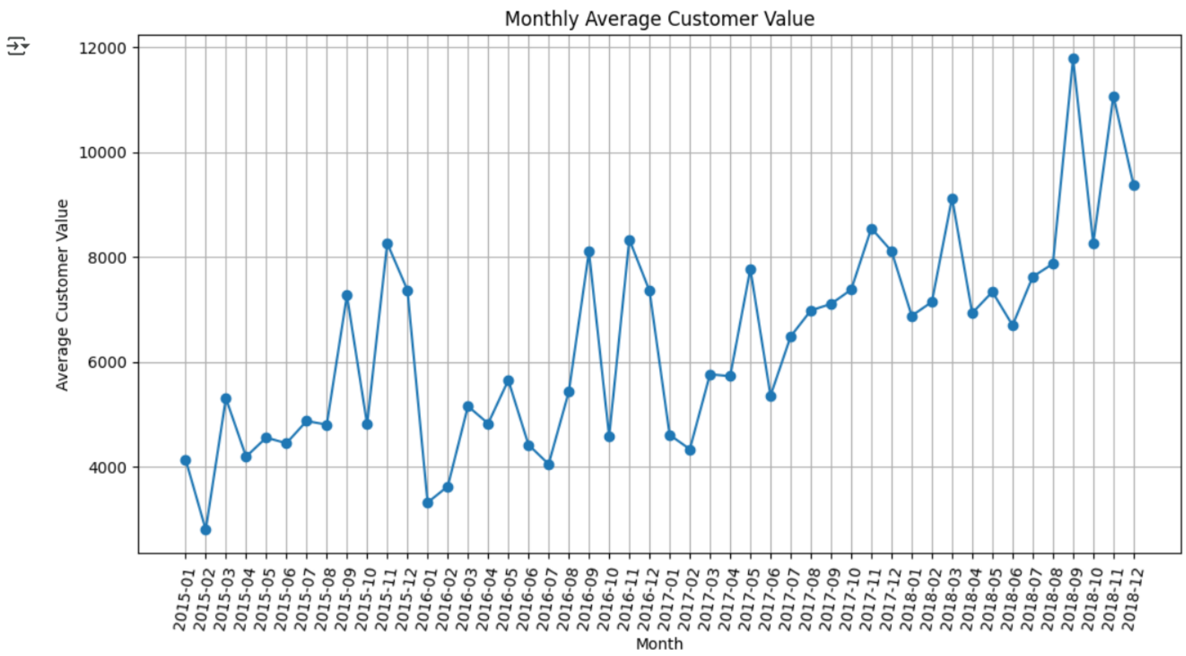
```
monthly_data['average_customer_value'] = monthly_data['total_revenue'] / monthly_data['unique_customers']
```

- ACV = Total Revenue / Number of Unique Customers
- Indicates how much each customer spends on average in a month.



```
plt.figure(figsize=(12,6))

plt.plot(monthly_data['month'].astype(str), monthly_data['average_customer_value'], marker='o')

plt.title('Monthly Average Customer Value')

plt.xlabel('Month')

plt.ylabel('Average Customer Value')

plt.xticks(rotation=80)

plt.grid(True)

plt.show()
```

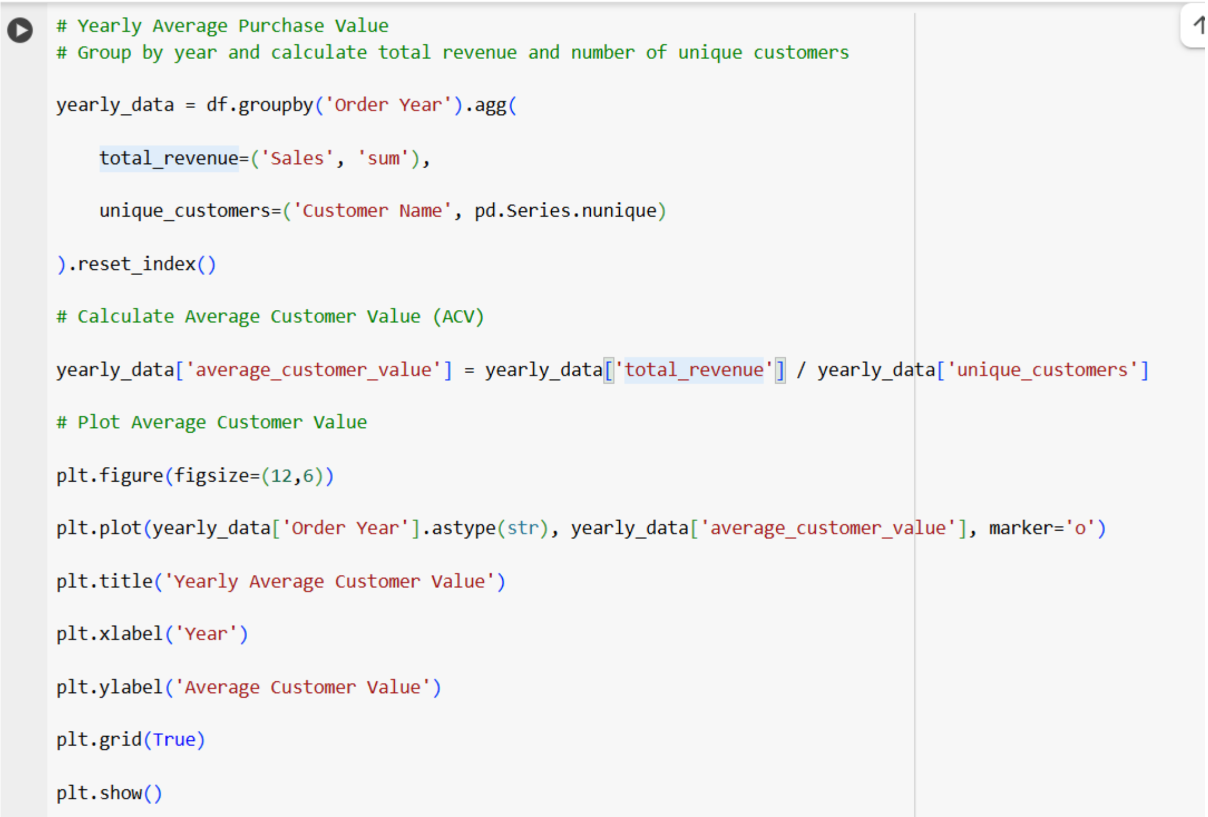
- A line plot is created to show how ACV changes month over month.
- X-axis: Time (Months)
- Y-axis: Average Customer Value (in currency units)



Observation:

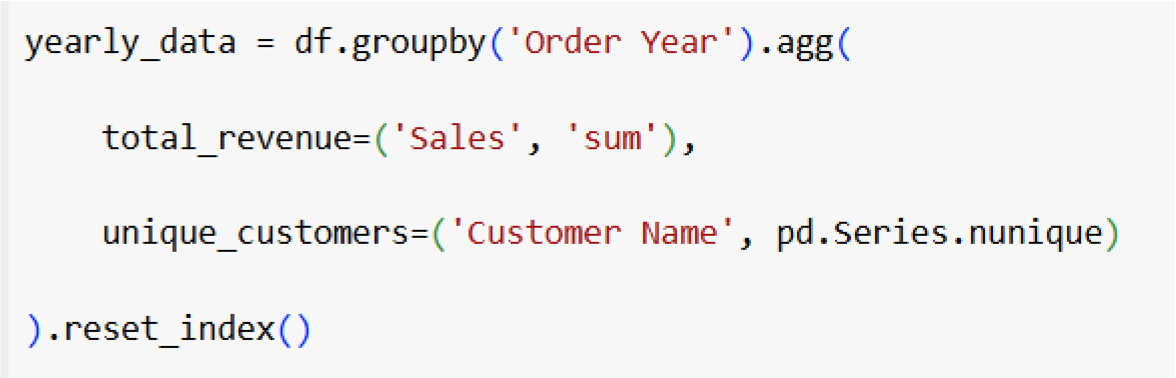
- The plot shows fluctuations in average spending per customer each month.
    - There's a general upward trend, especially from 2017 onward, suggesting:
    - Increasing customer spend.
    - Possibly higher-value products/services.
- Spikes or dips may indicate promotions, seasonality, or customer acquisition events

4.2 Yearly Average Purchase Value



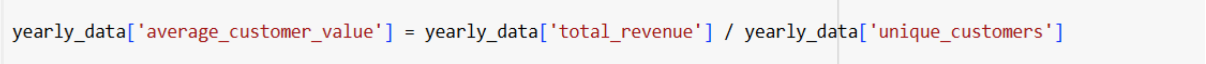
```
# Yearly Average Purchase Value
# Group by year and calculate total revenue and number of unique customers

yearly_data = df.groupby('Order Year').agg(

    total_revenue=('Sales', 'sum'),

    unique_customers=('Customer Name', pd.Series.nunique)

).reset_index()

# Calculate Average Customer Value (ACV)

yearly_data['average_customer_value'] = yearly_data['total_revenue'] / yearly_data['unique_customers']

# Plot Average Customer Value

plt.figure(figsize=(12,6))

plt.plot(yearly_data['Order Year'].astype(str), yearly_data['average_customer_value'], marker='o')

plt.title('Yearly Average Customer Value')

plt.xlabel('Year')

plt.ylabel('Average Customer Value')

plt.grid(True)

plt.show()
```

This Code Explains:



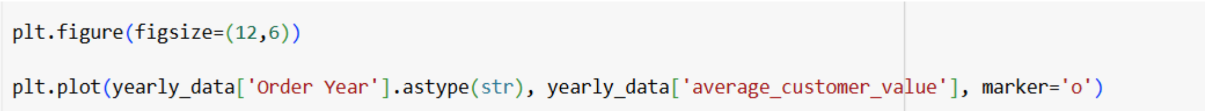
```
yearly_data = df.groupby('Order Year').agg(

    total_revenue=('Sales', 'sum'),

    unique_customers=('Customer Name', pd.Series.nunique)

).reset_index()
```

- df.groupby('Order Year') → groups all transactions by each year.
- agg() → computes:
  - total_revenue: Sum of Sales for that year.
  - unique_customers: Count of distinct customers that made purchases that year.



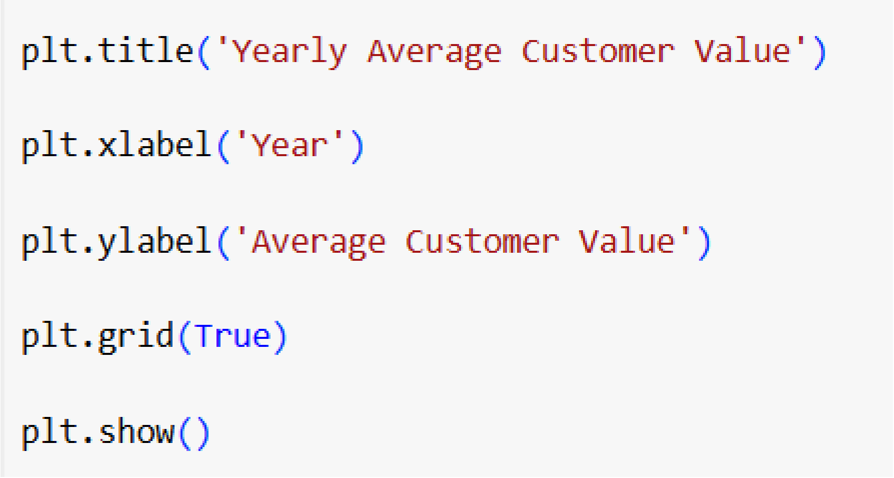
```
yearly_data['average_customer_value'] = yearly_data['total_revenue'] / yearly_data['unique_customers']
```

- This adds a new column average_customer_value:
- on average, how much one customer spent in that year.



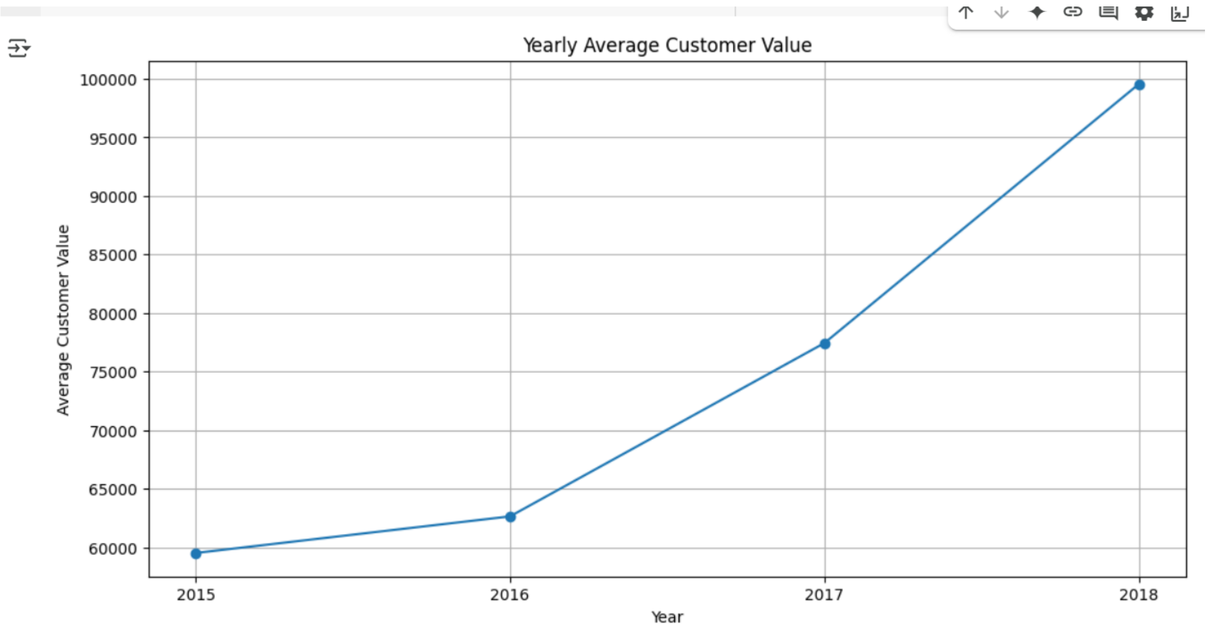
```
plt.figure(figsize=(12,6))

plt.plot(yearly_data['Order Year'].astype(str), yearly_data['average_customer_value'], marker='o')
```

- astype(str) is used to treat years as categorical labels, not numerical scale.
- marker='o' adds dots on each year point.



```
plt.title('Yearly Average Customer Value')

plt.xlabel('Year')

plt.ylabel('Average Customer Value')

plt.grid(True)

plt.show()
```

- This creates a clean line plot showing the trend of ACV over the years.



Observation:

- ACV increases over time → customers are spending more per year on average.
- ACV decreases → could mean more customers are buying, but spending less each.

4.3 Both methods are valuable:

- The monthly average purchase value is useful for optimizing short-term strategies and understanding monthly behavior.
- The yearly average purchase value is essential for long-term planning and assessing the overall health of the business.

5. Conclusion and Insights

Through this analysis, we have obtained valuable insights into grocery sales patterns, customer behavior, and profitability. By examining various aspects such as city-wise performance, sales trends over time, customer retention rates, and average purchase value, we can make informed decisions to optimize business operations in the grocery retail space.

Key Insights:

- Kanyakumari, Vellore, and Bodi had the highest total sales, with Kanyakumari leading. These cities should be prioritized for marketing and expansion.
- Vellore and Bodi generated the most revenue, but Perambalur had the highest profit margin, suggesting the need for region-specific pricing or discount strategies.
- Sales and profits fluctuated seasonally, with spikes during holidays, highlighting the need for stock management and promotions during peak periods. Year-over-year data showed consistent growth, especially in Year1.
- The repeat customer rate increased steadily, likely due to successful retention efforts like loyalty programs and targeted offers.

- Monthly average purchase values varied, influenced by promotions and holidays, providing insights to refine pricing and promotional strategies.