Data Spark: Illuminating Insights for Global Electronics

Overview

This project, titled **Data Spark: Illuminating Insights for Global Electronics**, aims to deliver a comprehensive Exploratory Data Analysis (EDA) report for Global Electronics. The primary goals include ensuring all datasets are clean and integrated, uncovering trends and insights through EDA, creating visualizations to effectively communicate key findings, and providing actionable recommendations.

Prerequisites

- Python 3.x
- Required Python libraries: pandas, seaborn, matplotlib, mysql-connector-python
- MySQL server
- Power BI

Usage

1. **Load Data**:
   - Ensure all CSV files (Products.csv, Sales.csv, Stores.csv, customers.csv, Exchange_Rates.csv) are in the same directory as the script.
2. **Preprocess Data**:
   - The script cleans and preprocesses the data by handling missing values, converting data types, and standardizing numerical columns.
3. **Merge Data**:
   - The script merges the cleaned datasets into a single DataFrame for further analysis.

4. **Outlier Detection and Reduction**:
    - The script identifies outliers using box plots and reduces them by standardizing the data and removing points beyond 3 standard deviations.
5. **Database Interaction**:
    - The script creates tables in a MySQL database and inserts the processed data.
    - Ensure the MySQL server is running and accessible.

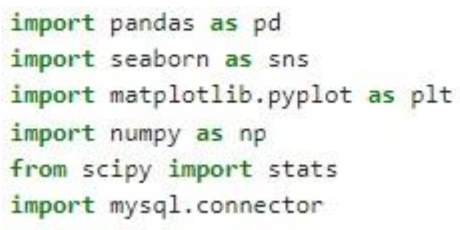
Setup Instructions

1. Install Dependencies

   Installed the required Python packages using pip:

   pip install pandas seaborn matplotlib numpy scipy mysql-connector-python

Imports:

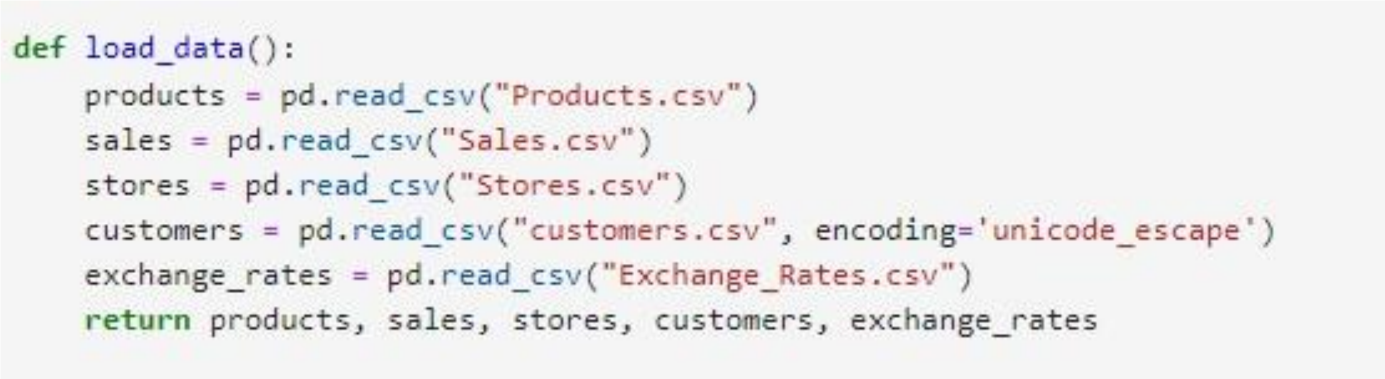


```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import mysql.connector
```

Code Explanation:

load_data Function Explanation

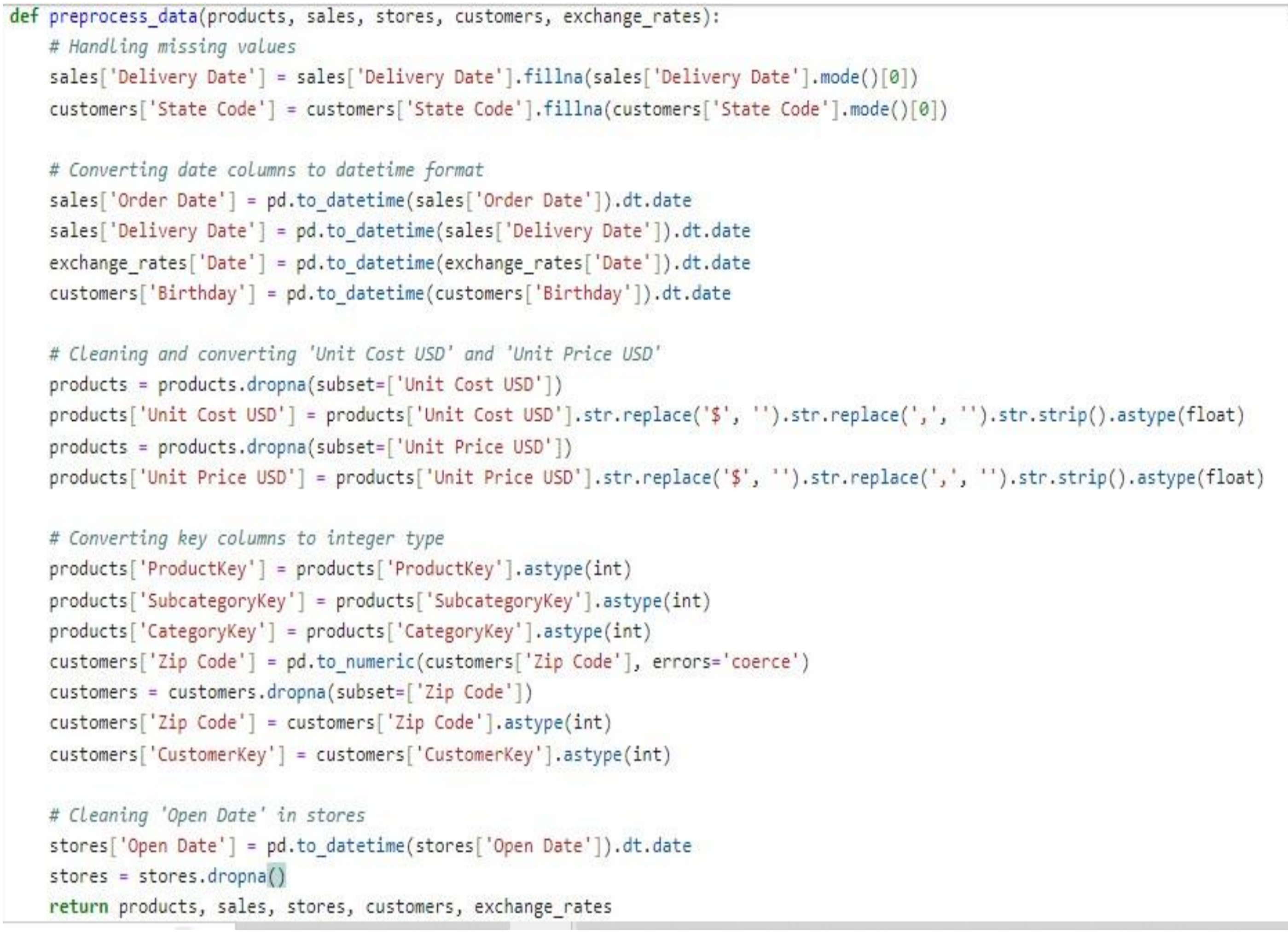
- The **load_data function** is responsible for loading various datasets from CSV files into pandas Data Frames. These datasets include product information, sales data, store details, customer information, and exchange rates.



```
def load_data():
    products = pd.read_csv("Products.csv")
    sales = pd.read_csv("Sales.csv")
    stores = pd.read_csv("Stores.csv")
    customers = pd.read_csv("customers.csv", encoding='unicode_escape')
    exchange_rates = pd.read_csv("Exchange_Rates.csv")
    return products, sales, stores, customers, exchange_rates
```

preprocess_data Function Explanation

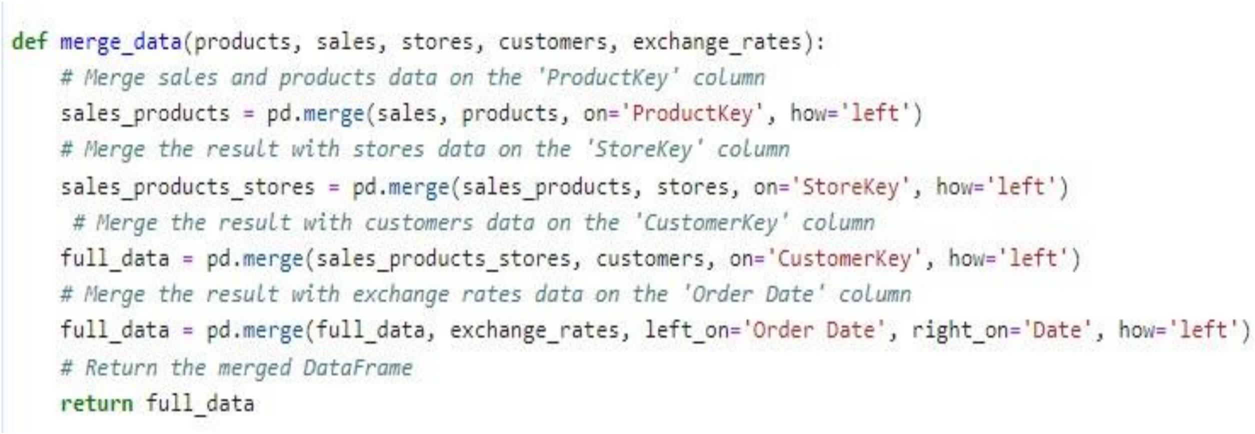
- The **preprocess_data** function handles missing values, converts date columns to a consistent format, cleans and converts monetary columns, ensures key columns are of the correct type, and removes any invalid rows. This ensures the data is clean and ready for analysis.



```
def preprocess_data(products, sales, stores, customers, exchange_rates):
    # Handling missing values
    sales['Delivery Date'] = sales['Delivery Date'].fillna(sales['Delivery Date'].mode()[0])
    customers['State Code'] = customers['State Code'].fillna(customers['State Code'].mode()[0])

    # Converting date columns to datetime format
    sales['Order Date'] = pd.to_datetime(sales['Order Date']).dt.date
    sales['Delivery Date'] = pd.to_datetime(sales['Delivery Date']).dt.date
    exchange_rates['Date'] = pd.to_datetime(exchange_rates['Date']).dt.date
    customers['Birthday'] = pd.to_datetime(customers['Birthday']).dt.date

    # Cleaning and converting 'Unit Cost USD' and 'Unit Price USD'
    products = products.dropna(subset=['Unit Cost USD'])
    products['Unit Cost USD'] = products['Unit Cost USD'].str.replace('$', '').str.replace(',', '').str.strip().astype(float)
    products = products.dropna(subset=['Unit Price USD'])
    products['Unit Price USD'] = products['Unit Price USD'].str.replace('$', '').str.replace(',', '').str.strip().astype(float)

    # Converting key columns to integer type
    products['ProductKey'] = products['ProductKey'].astype(int)
    products['SubcategoryKey'] = products['SubcategoryKey'].astype(int)
    products['CategoryKey'] = products['CategoryKey'].astype(int)
    customers['Zip Code'] = pd.to_numeric(customers['Zip Code'], errors='coerce')
    customers = customers.dropna(subset=['Zip Code'])
    customers['Zip Code'] = customers['Zip Code'].astype(int)
    customers['CustomerKey'] = customers['CustomerKey'].astype(int)

    # Cleaning 'Open Date' in stores
    stores['Open Date'] = pd.to_datetime(stores['Open Date']).dt.date
    stores = stores.dropna()
    return products, sales, stores, customers, exchange_rates
```

Merge_data Function Explanation

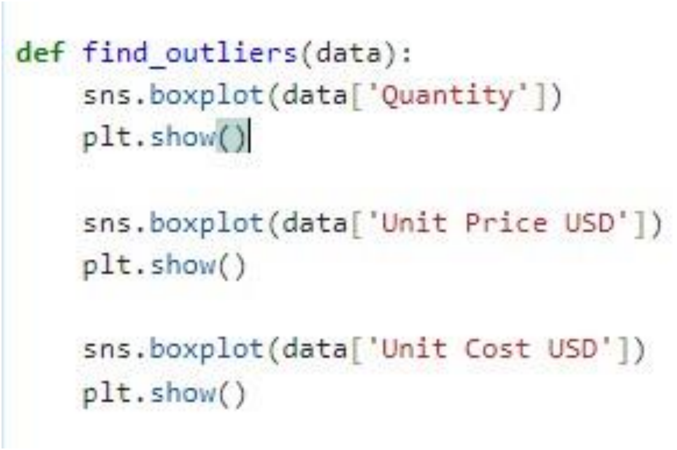
- The **merge_data** function performs a series of left joins to combine the sales, products, stores, customers, and exchange_rates Data Frames into a single DataFrame. This merged Data Frame, full_data, contains comprehensive information from all the source DataFrames, allowing for thorough analysis and insights.



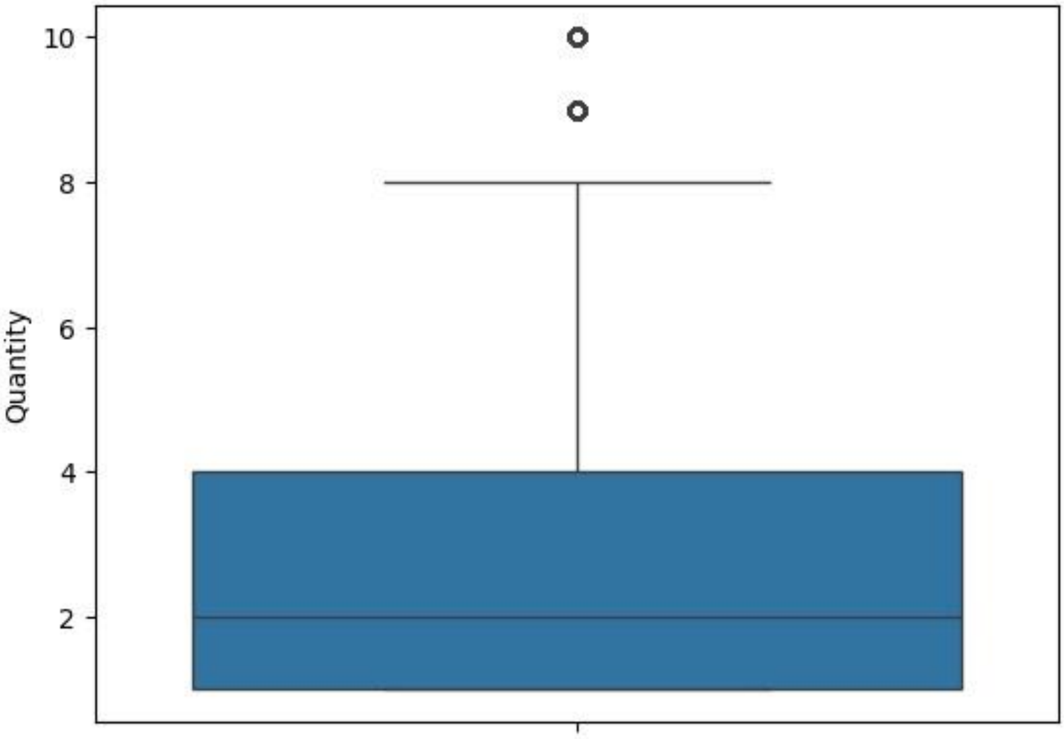
```python
def merge_data(products, sales, stores, customers, exchange_rates):
    # Merge sales and products data on the 'ProductKey' column
    sales_products = pd.merge(sales, products, on='ProductKey', how='left')
    # Merge the result with stores data on the 'StoreKey' column
    sales_products_stores = pd.merge(sales_products, stores, on='StoreKey', how='left')
     # Merge the result with customers data on the 'CustomerKey' column
    full_data = pd.merge(sales_products_stores, customers, on='CustomerKey', how='left')
    # Merge the result with exchange rates data on the 'Order Date' column
    full_data = pd.merge(full_data, exchange_rates, left_on='Order Date', right_on='Date', how='left')
    # Return the merged DataFrame
    return full_data
```

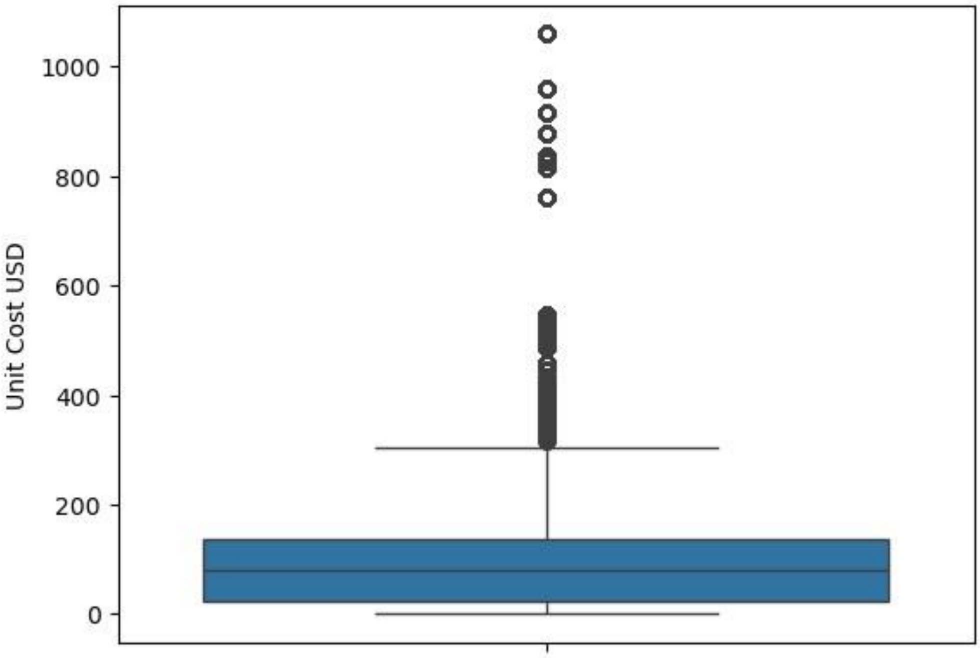
Find_outliers Function Explanation

- The **find_outliers** function creates and displays box plots for the Quantity, Unit Price USD, and Unit Cost USD columns of the input DataFrame.
- These box plots help visually identify outliers in the data. Visually inspecting these box plots, you can easily identify data points that fall outside the expected range and may be considered outliers.
- This is a crucial step in data analysis, as outliers can significantly affect the results and interpretations of statistical analyses.



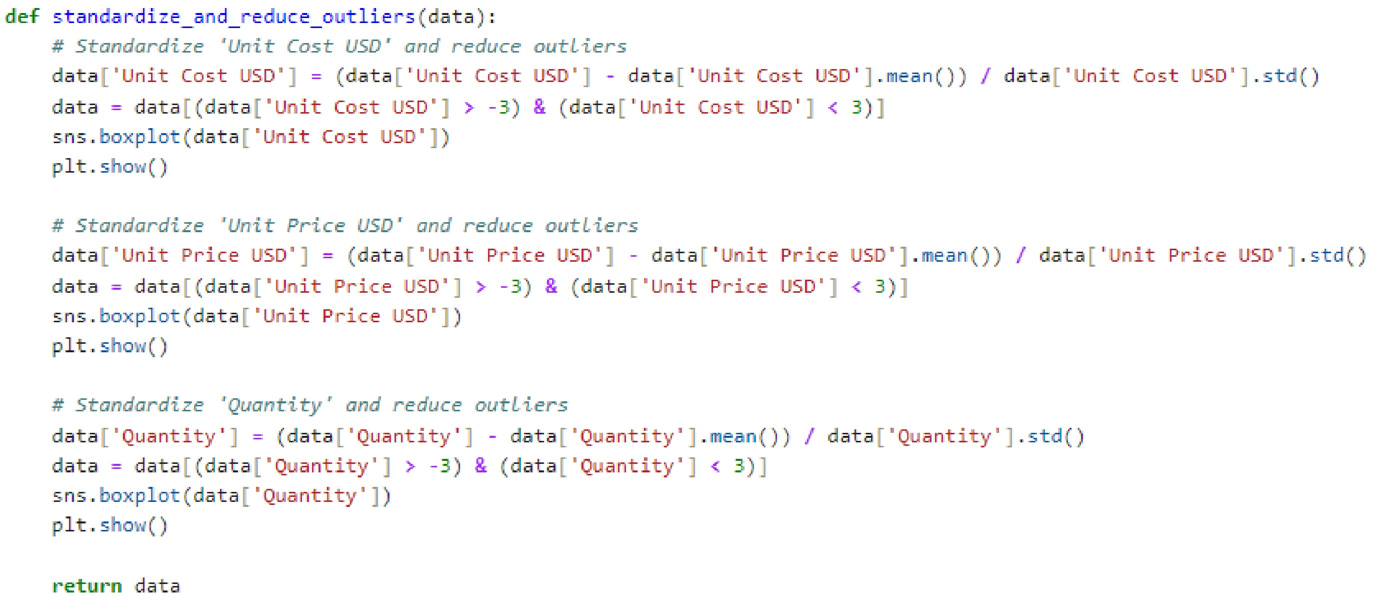
```
def find_outliers(data):
    sns.boxplot(data['Quantity'])
    plt.show()

    sns.boxplot(data['Unit Price USD'])
    plt.show()

    sns.boxplot(data['Unit Cost USD'])
    plt.show()
```





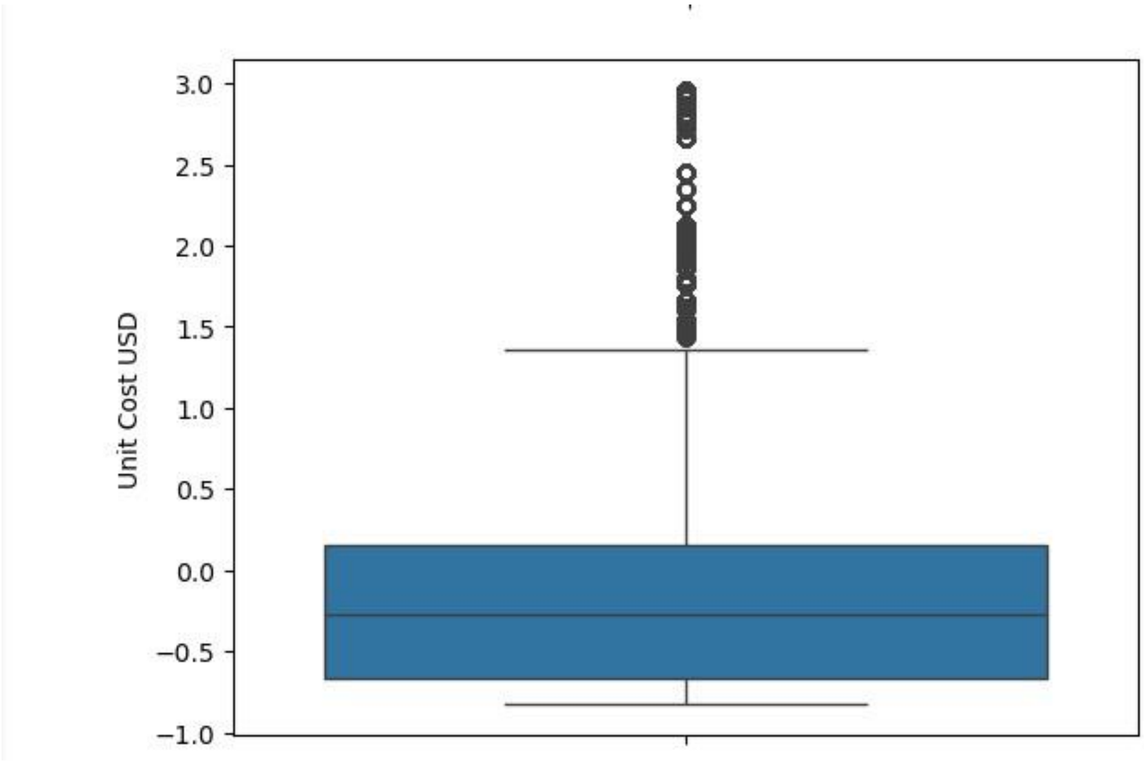
Standardize_and_reduce_outliers Function Explanation

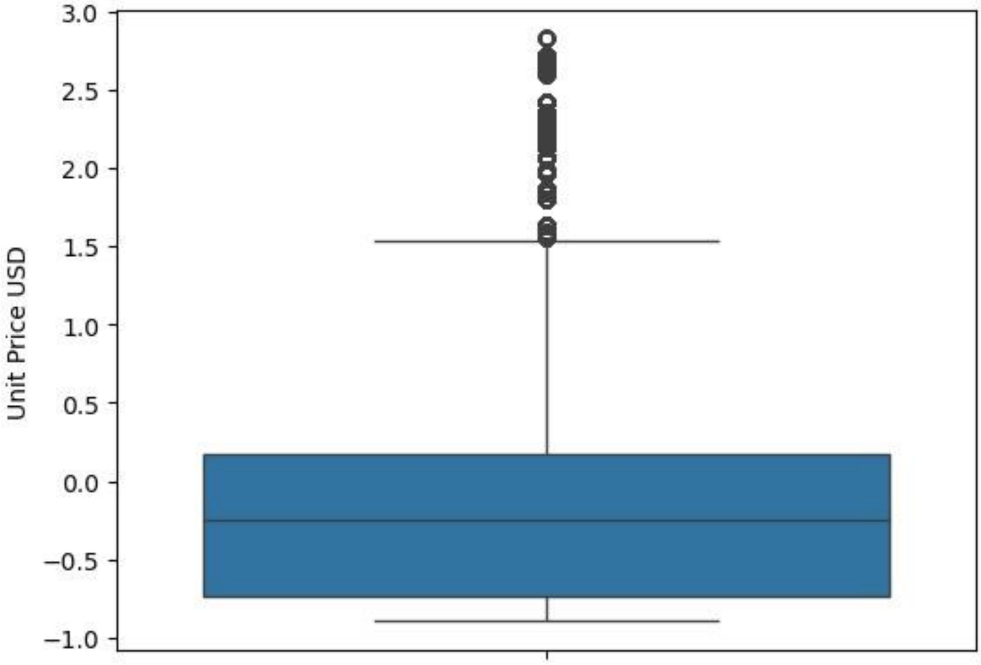
- The standardize_and_reduce_outliers function standardizes the Unit Cost USD, Unit Price USD, and Quantity columns of the DataFrame, removing data points that are more than 3 standard deviations away from the mean.
- This process helps in mitigating the influence of outliers and ensures a more normally distributed dataset. The function also provides visualizations of the standardized data using box plots, aiding in the identification of remaining outliers and understanding the distribution of the data.

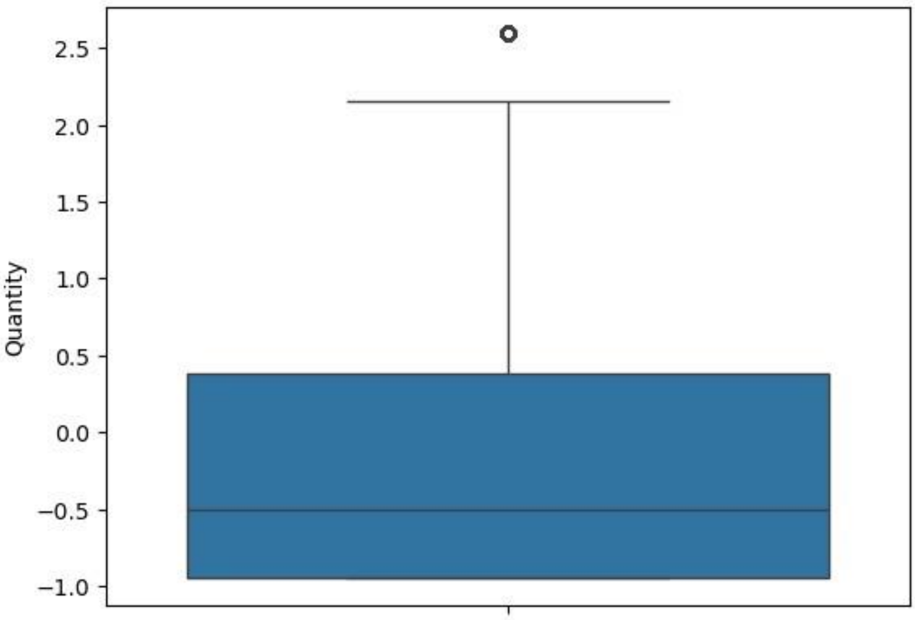


```
def standardize_and_reduce_outliers(data):
    # Standardize 'Unit Cost USD' and reduce outliers
    data['Unit Cost USD'] = (data['Unit Cost USD'] - data['Unit Cost USD'].mean()) / data['Unit Cost USD'].std()
    data = data[(data['Unit Cost USD'] > -3) & (data['Unit Cost USD'] < 3)]
    sns.boxplot(data['Unit Cost USD'])
    plt.show()

    # Standardize 'Unit Price USD' and reduce outliers
    data['Unit Price USD'] = (data['Unit Price USD'] - data['Unit Price USD'].mean()) / data['Unit Price USD'].std()
    data = data[(data['Unit Price USD'] > -3) & (data['Unit Price USD'] < 3)]
    sns.boxplot(data['Unit Price USD'])
    plt.show()

    # Standardize 'Quantity' and reduce outliers
    data['Quantity'] = (data['Quantity'] - data['Quantity'].mean()) / data['Quantity'].std()
    data = data[(data['Quantity'] > -3) & (data['Quantity'] < 3)]
    sns.boxplot(data['Quantity'])
    plt.show()

    return data
```

Box Plot:

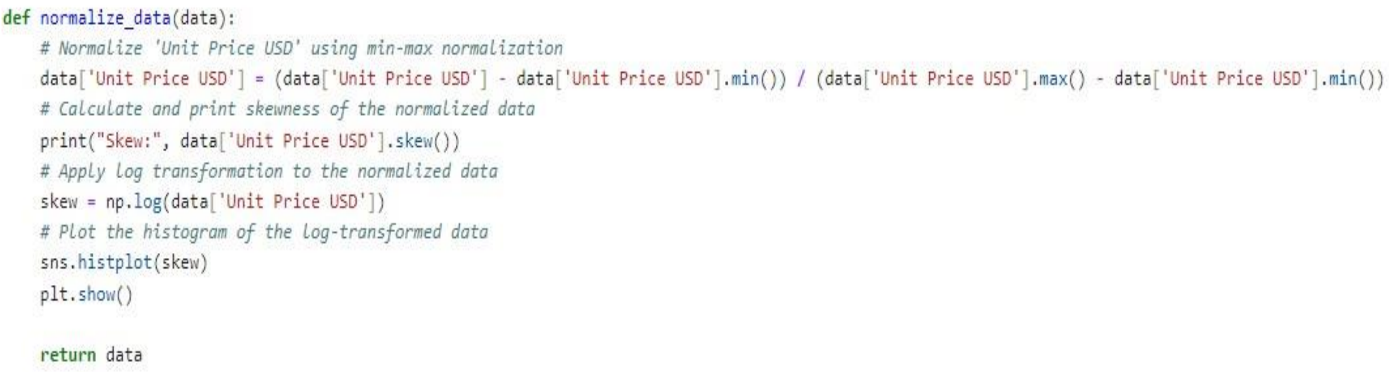



3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Unit Price USD


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
Quantity

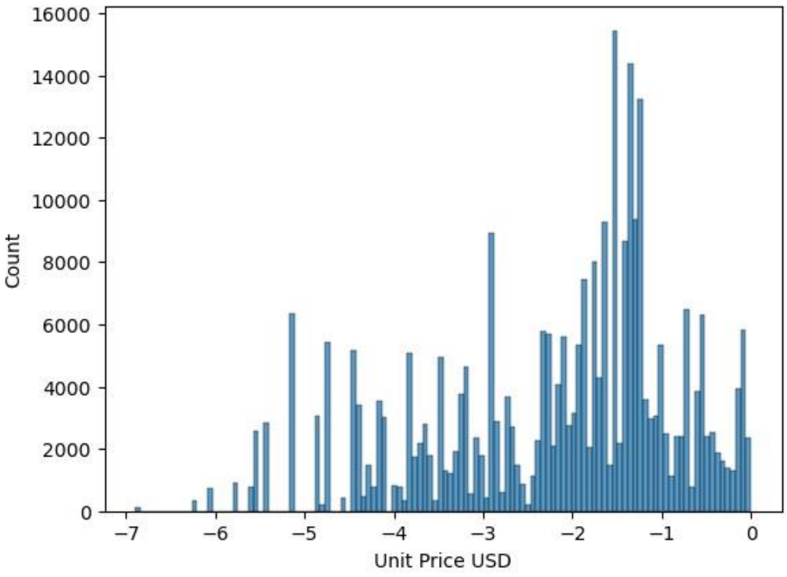
Normalize_data Function Explanation:

- The normalize_data function normalizes the 'Unit Price USD' column in the dataset, calculates its skewness, and visualizes the data distribution before and after applying a logarithmic transformation.
- This transformation scales the values of 'Unit Price USD' to be between 0 and 1.



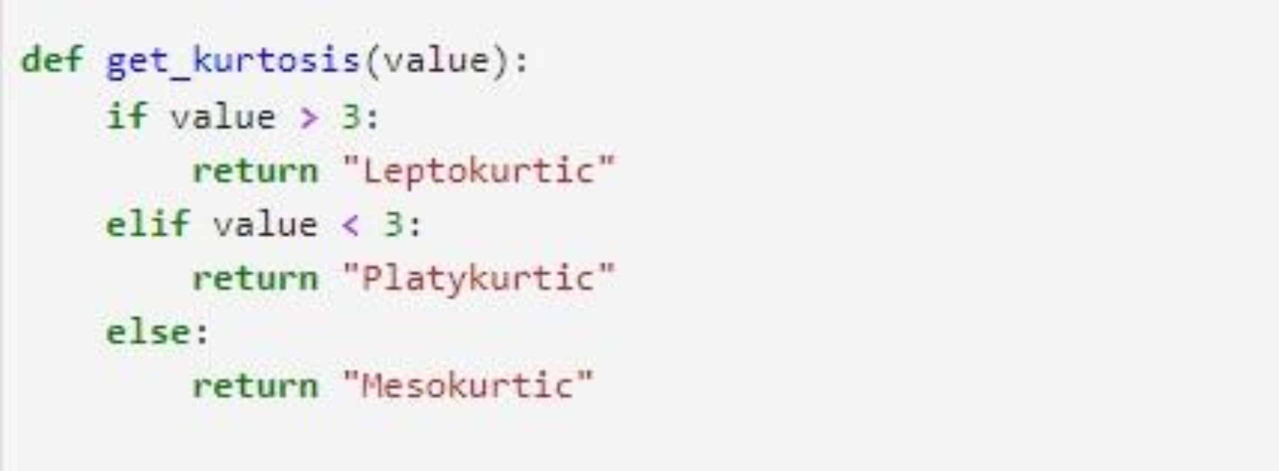
```
def normalize_data(data):
    # Normalize 'Unit Price USD' using min-max normalization
    data['Unit Price USD'] = (data['Unit Price USD'] - data['Unit Price USD'].min()) / (data['Unit Price USD'].max() - data['Unit Price USD'].min())
    # Calculate and print skewness of the normalized data
    print("Skew:", data['Unit Price USD'].skew())
    # Apply log transformation to the normalized data
    skew = np.log(data['Unit Price USD'])
    # Plot the histogram of the log-transformed data
    sns.histplot(skew)
    plt.show()

    return data
```

Histplot:



Get_kurtosis Function Explanation:

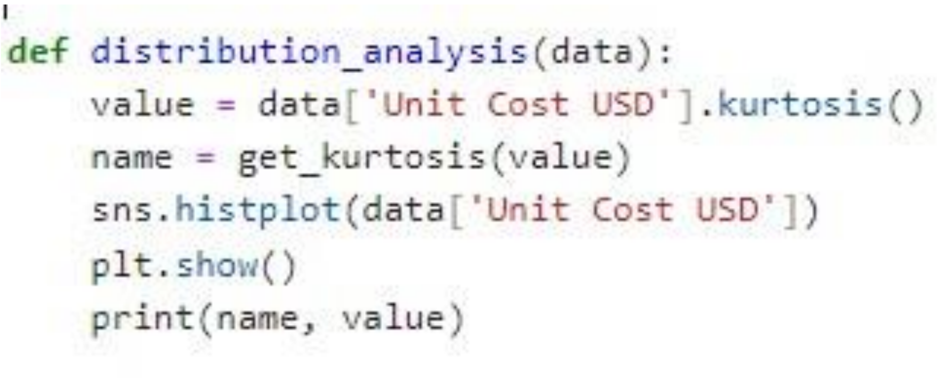
- The get_kurtosis function classifies the kurtosis of a distribution based on the given value.
- Kurtosis is a statistical measure that describes the shape of a distribution's tails in relation to its overall shape.



```
def get_kurtosis(value):
    if value > 3:
        return "Leptokurtic"
    elif value < 3:
        return "Platykurtic"
    else:
        return "Mesokurtic"
```

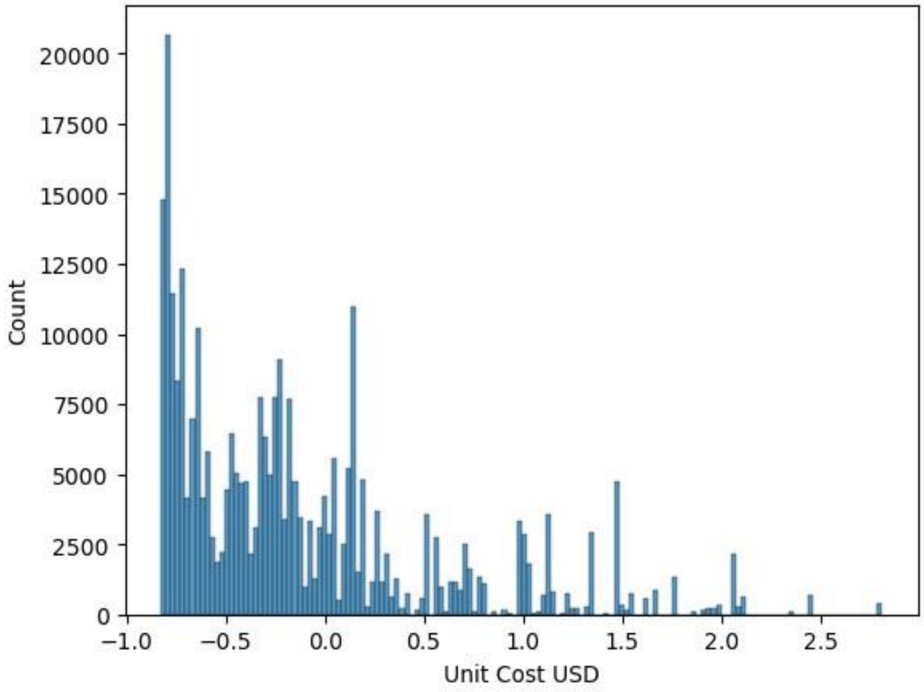
Distribution_analysis Function Explanation:

- The distribution_analysis function performs a kurtosis analysis on the 'Unit Cost USD' column of a given DataFrame.
- It then classifies the distribution based on the kurtosis value, visualizes the distribution using a histogram, and prints the kurtosis type and value.



```
def distribution_analysis(data):
    value = data['Unit Cost USD'].kurtosis()
    name = get_kurtosis(value)
    sns.histplot(data['Unit Cost USD'])
    plt.show()
    print(name, value)
```

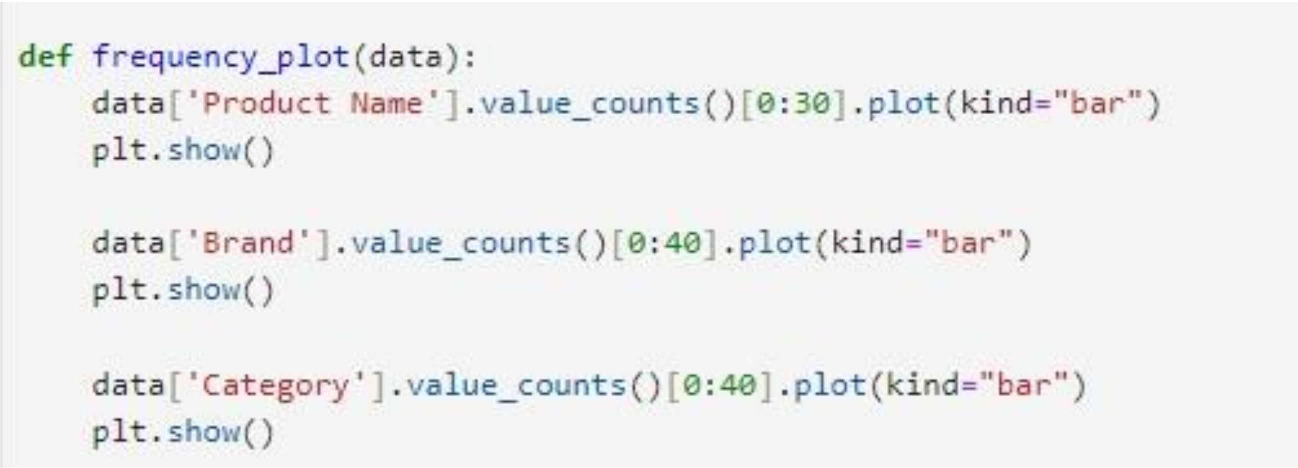
Hist plot:



```
Platykurtic 1.6799636851430257
```

Frequency_plot Function Explanation:

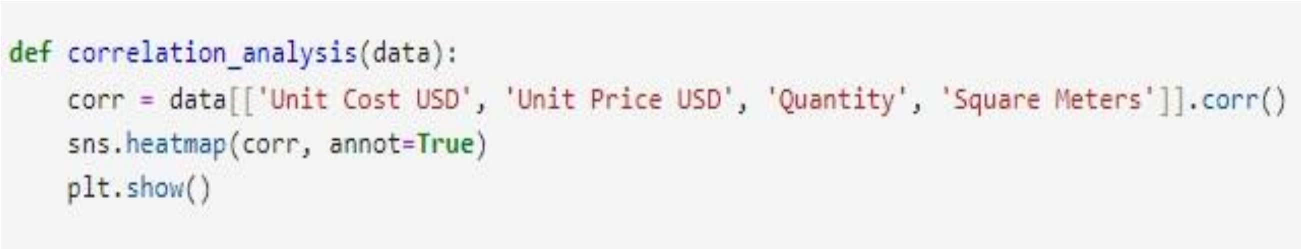
- The frequency_plot function generates bar plots showing the frequency distribution of product-related attributes from a given Data Frame.
- It visualizes the top 30 products, top 40 brands, and top 40 categories based on their occurrence in the dataset.



```
def frequency_plot(data):
    data['Product Name'].value_counts()[0:30].plot(kind="bar")
    plt.show()

    data['Brand'].value_counts()[0:40].plot(kind="bar")
    plt.show()

    data['Category'].value_counts()[0:40].plot(kind="bar")
    plt.show()
```

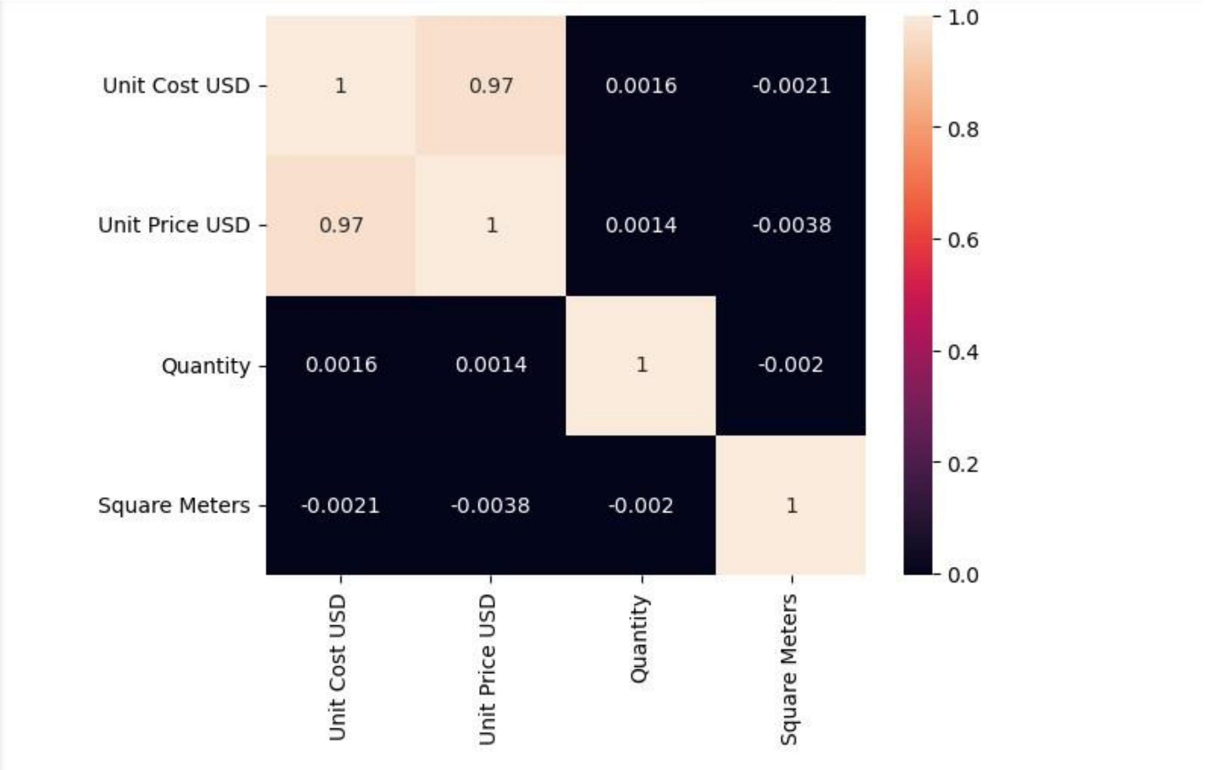
Correlation_analysis Function Explanation:

- The correlation_analysis function performs a correlation analysis on a subset of numerical columns from a DataFrame and visualizes the results using a heatmap.



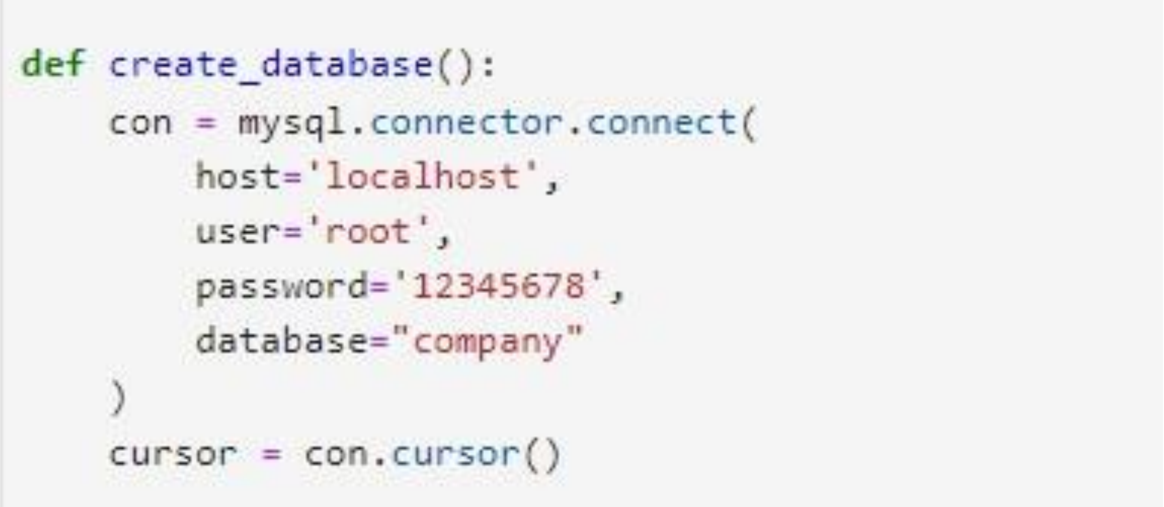
```python
def correlation_analysis(data):
    corr = data[['Unit Cost USD', 'Unit Price USD', 'Quantity', 'Square Meters']].corr()
    sns.heatmap(corr, annot=True)
    plt.show()
```

Heat map:



Database Connection

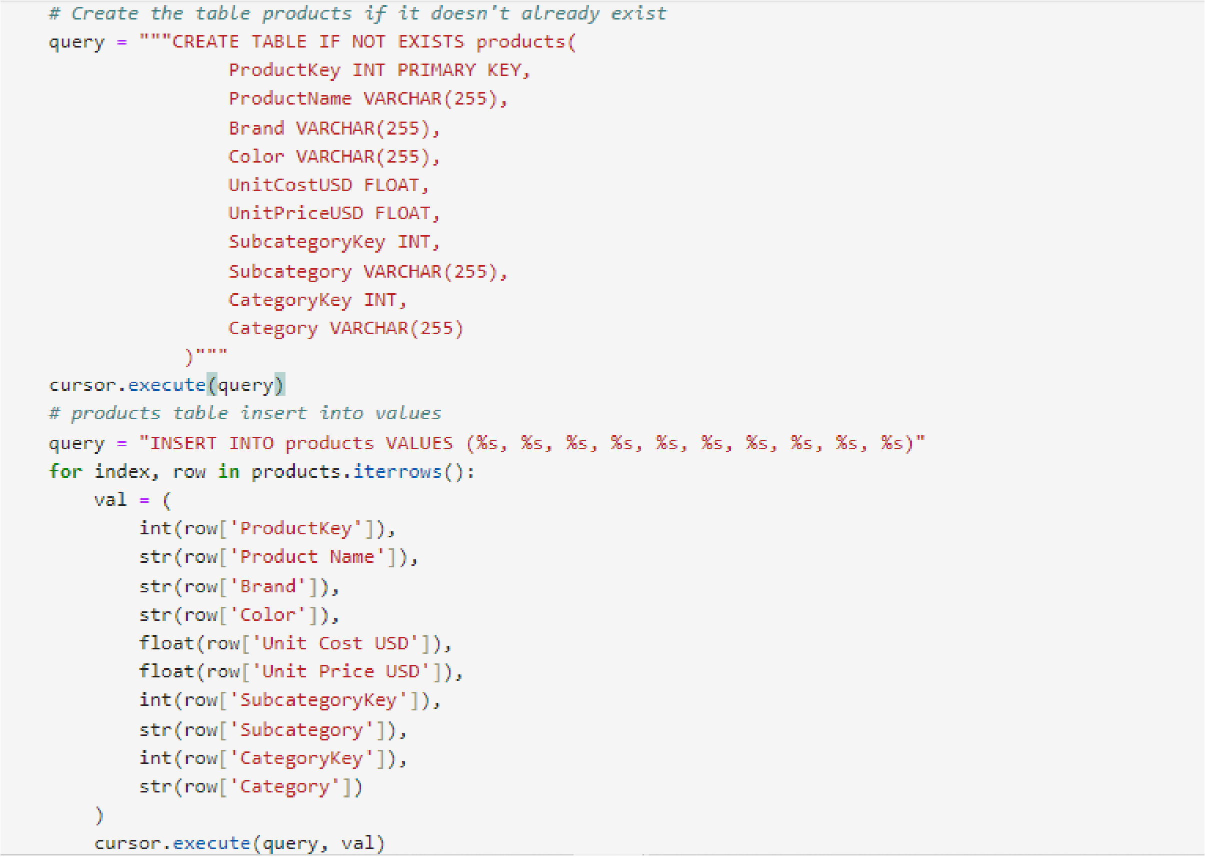
- The create_database function connects to a MySQL database and prepares to interact with it.
- mysql. connector.connect() establishes a connection to a MySQL database.



```
def create_database():
    con = mysql.connector.connect(
        host='localhost',
        user='root',
        password='12345678',
        database="company"
    )
    cursor = con.cursor()
```

Create the products Table:

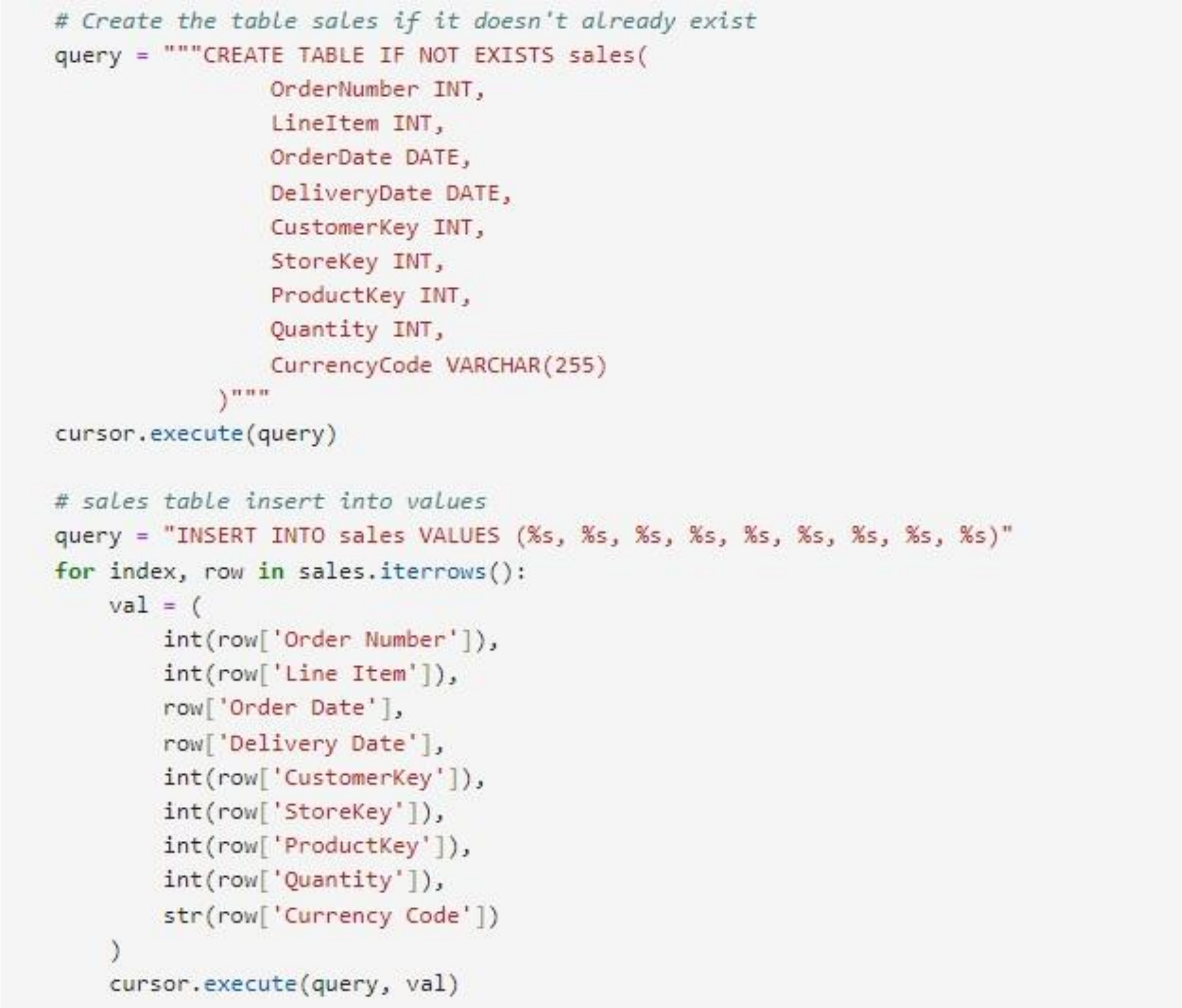
- create a new table named products if it does not already exist.
- inserts data from a DataFrame into the products table, converted the data to the correct types and handling the insertion in a loop.



```
# Create the table products if it doesn't already exist
query = """CREATE TABLE IF NOT EXISTS products(
                ProductKey INT PRIMARY KEY,
                ProductName VARCHAR(255),
                Brand VARCHAR(255),
                Color VARCHAR(255),
                UnitCostUSD FLOAT,
                UnitPriceUSD FLOAT,
                SubcategoryKey INT,
                Subcategory VARCHAR(255),
                CategoryKey INT,
                Category VARCHAR(255)
            )"""
cursor.execute(query)
# products table insert into values
query = "INSERT INTO products VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)"
for index, row in products.iterrows():
    val = (
        int(row['ProductKey']),
        str(row['Product Name']),
        str(row['Brand']),
        str(row['Color']),
        float(row['Unit Cost USD']),
        float(row['Unit Price USD']),
        int(row['SubcategoryKey']),
        str(row['Subcategory']),
        int(row['CategoryKey']),
        str(row['Category'])
    )
    cursor.execute(query, val)
```

Create the sales Table:

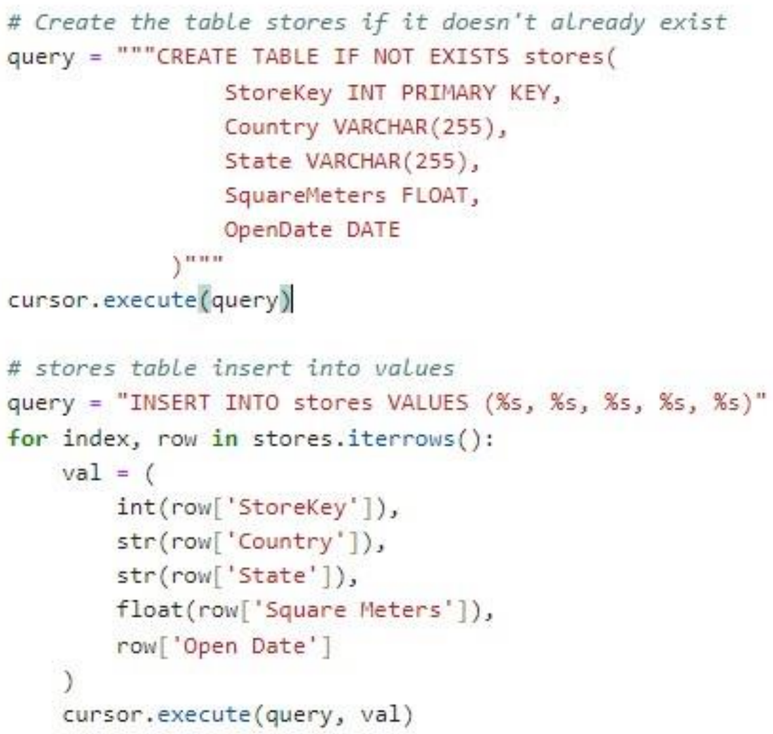
- creates a new table named sales if it does not already exist.
- Inserts data from a DataFrame into the sales table, ensuring data is converted to the correct types and inserted row by row.



```
# Create the table sales if it doesn't already exist
query = """CREATE TABLE IF NOT EXISTS sales(
                OrderNumber INT,
                LineItem INT,
                OrderDate DATE,
                DeliveryDate DATE,
                CustomerKey INT,
                StoreKey INT,
                ProductKey INT,
                Quantity INT,
                CurrencyCode VARCHAR(255)
            )"""
cursor.execute(query)

# sales table insert into values
query = "INSERT INTO sales VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)"
for index, row in sales.iterrows():
    val = (
        int(row['Order Number']),
        int(row['Line Item']),
        row['Order Date'],
        row['Delivery Date'],
        int(row['CustomerKey']),
        int(row['StoreKey']),
        int(row['ProductKey']),
        int(row['Quantity']),
        str(row['Currency Code'])
    )
    cursor.execute(query, val)
```

Create the stores Table:

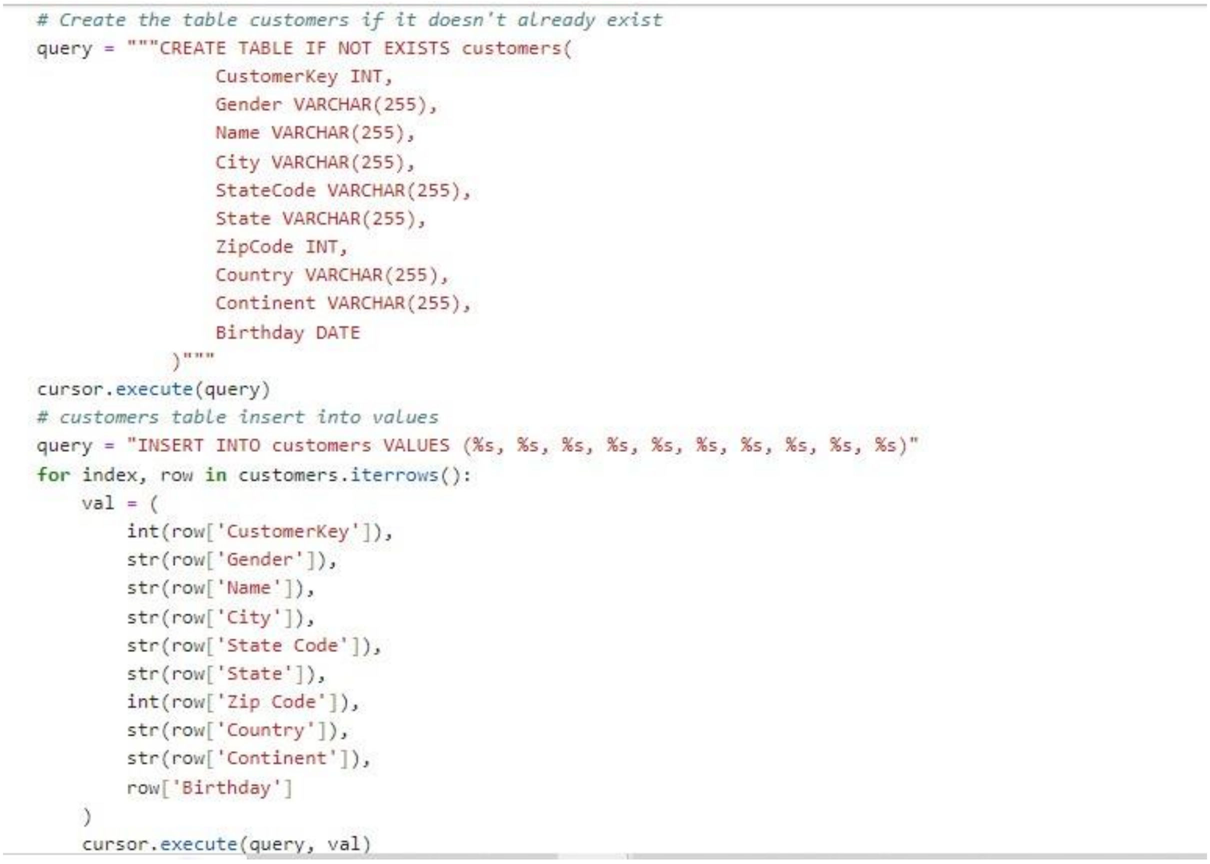
- creates a new table named stores if it does not already exist.
- Inserts data from a DataFrame into the stores table, ensuring data is converted to the correct types and inserted row by row.



```
# Create the table stores if it doesn't already exist
query = """CREATE TABLE IF NOT EXISTS stores(
                StoreKey INT PRIMARY KEY,
                Country VARCHAR(255),
                State VARCHAR(255),
                SquareMeters FLOAT,
                OpenDate DATE
            )"""
cursor.execute(query)

# stores table insert into values
query = "INSERT INTO stores VALUES (%s, %s, %s, %s, %s)"
for index, row in stores.iterrows():
    val = (
        int(row['StoreKey']),
        str(row['Country']),
        str(row['State']),
        float(row['Square Meters']),
        row['Open Date']
    )
    cursor.execute(query, val)
```

Create the customers Table:

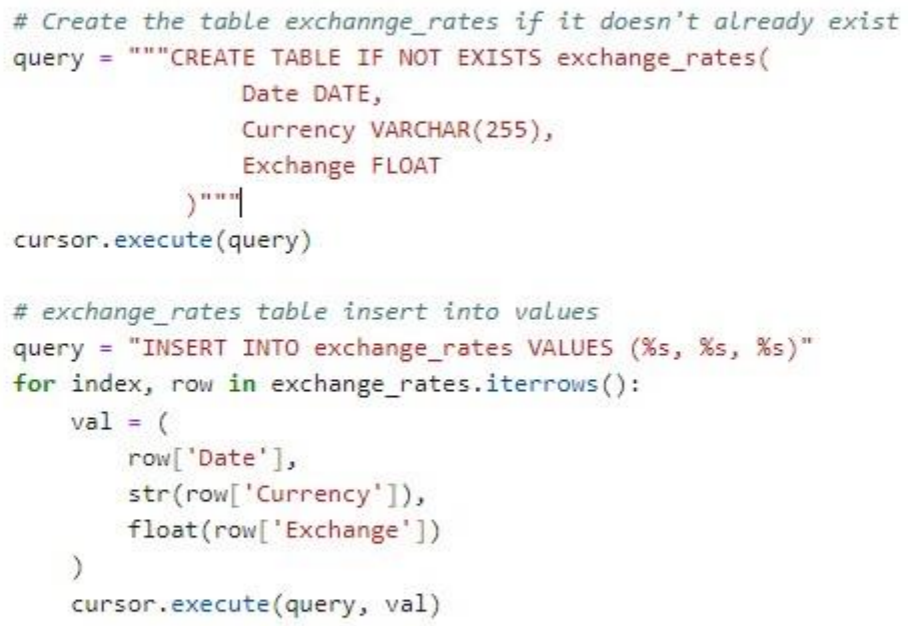
- creates a new table named customers if it does not already exist.
- Inserts data from the DataFrame into the customers table, ensuring that each column's data is converted to the correct type and inserted row by row.



```
# Create the table customers if it doesn't already exist
query = """CREATE TABLE IF NOT EXISTS customers(
                CustomerKey INT,
                Gender VARCHAR(255),
                Name VARCHAR(255),
                City VARCHAR(255),
                StateCode VARCHAR(255),
                State VARCHAR(255),
                ZipCode INT,
                Country VARCHAR(255),
                Continent VARCHAR(255),
                Birthday DATE
            )"""
cursor.execute(query)
# customers table insert into values
query = "INSERT INTO customers VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)"
for index, row in customers.iterrows():
    val = (
        int(row['CustomerKey']),
        str(row['Gender']),
        str(row['Name']),
        str(row['City']),
        str(row['State Code']),
        str(row['State']),
        int(row['Zip Code']),
        str(row['Country']),
        str(row['Continent']),
        row['Birthday']
    )
    cursor.execute(query, val)
```

Create the exchannge_rates Table:

- creates a new table named exchange_rates if it does not already exist.
- Inserts data from the DataFrame into the exchange_rates table, converting each piece of data to the appropriate type and inserting it row by row.



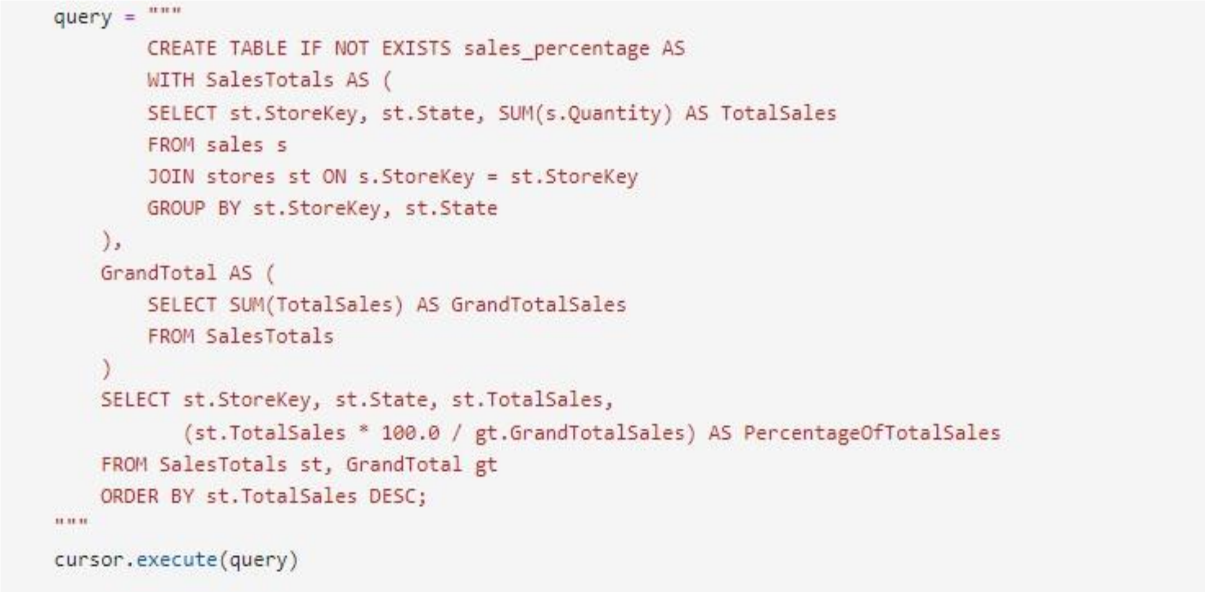
```
# Create the table exchannge_rates if it doesn't already exist
query = """CREATE TABLE IF NOT EXISTS exchange_rates(
                Date DATE,
                Currency VARCHAR(255),
                Exchange FLOAT
            )"""
cursor.execute(query)

# exchange_rates table insert into values
query = "INSERT INTO exchange_rates VALUES (%s, %s, %s)"
for index, row in exchange_rates.iterrows():
    val = (
        row['Date'],
        str(row['Currency']),
        float(row['Exchange'])
    )
    cursor.execute(query, val)
```

Visualizations process:

1. **Creating the sales_percentage Table**:

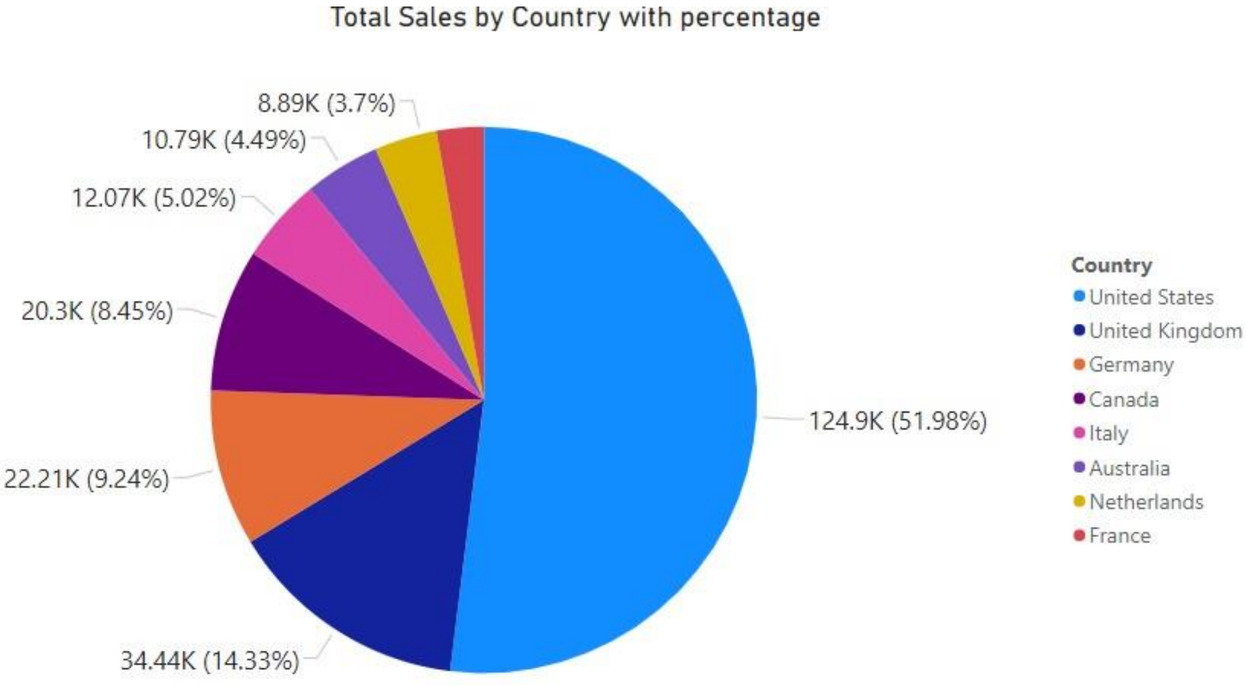
   The function creates a sales_percentage table that shows the percentage contribution of each store's sales to the total sales.

- Calculating total sales per store.
- Computing the grand total of sales across all stores.
- Determining each store's percentage of the grand total and ordering the results by total sales.



```
query = """
        CREATE TABLE IF NOT EXISTS sales_percentage AS
        WITH SalesTotals AS (
        SELECT st.StoreKey, st.State, SUM(s.Quantity) AS TotalSales
        FROM sales s
        JOIN stores st ON s.StoreKey = st.StoreKey
        GROUP BY st.StoreKey, st.State
    ),
    GrandTotal AS (
        SELECT SUM(TotalSales) AS GrandTotalSales
        FROM SalesTotals
    )
    SELECT st.StoreKey, st.State, st.TotalSales,
           (st.TotalSales * 100.0 / gt.GrandTotalSales) AS PercentageOfTotalSales
    FROM SalesTotals st, GrandTotal gt
    ORDER BY st.TotalSales DESC;
"""
cursor.execute(query)
```

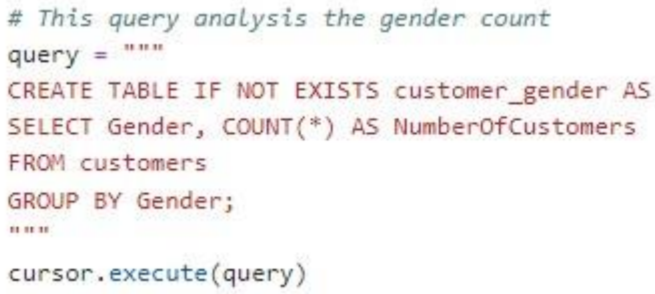
Power BI Visualizations:



2) **Creating the customer_gender Table**:

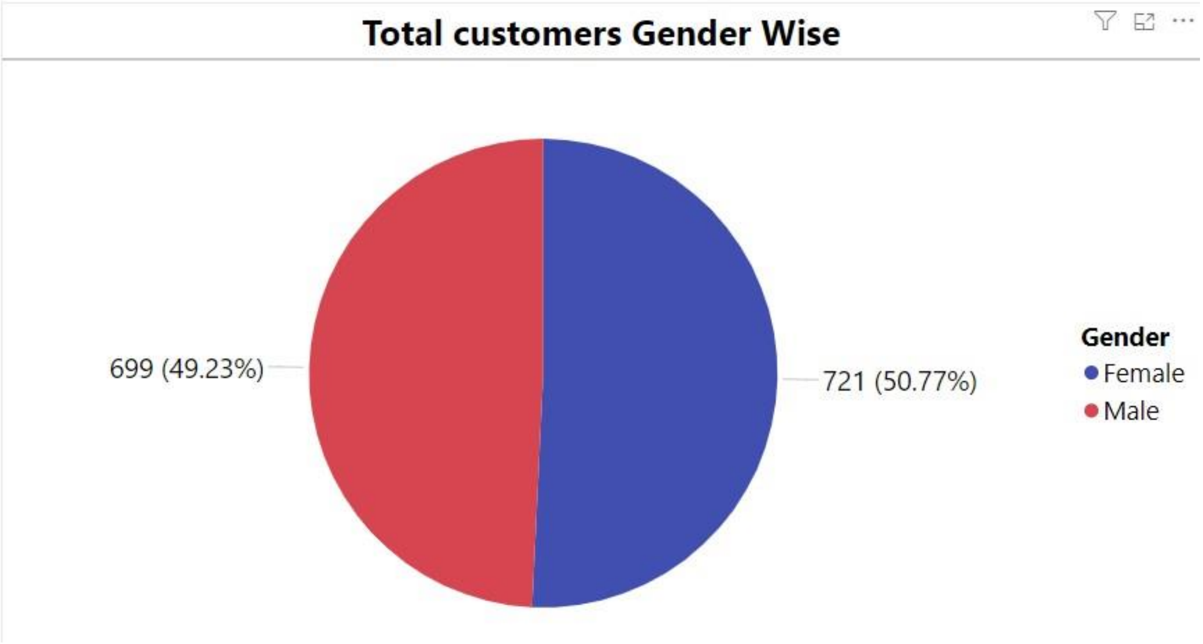
The query creates a table called customer_gender that contains:

- Gender: The gender of customers.
- NumberOfCustomers: The count of customers for each gender.
- This table contains how many customers fall into each gender category, which can be useful for understanding the gender distribution among the customer base.



```
# This query analysis the gender count
query = """
CREATE TABLE IF NOT EXISTS customer_gender AS
SELECT Gender, COUNT(*) AS NumberOfCustomers
FROM customers
GROUP BY Gender;
"""
cursor.execute(query)
```

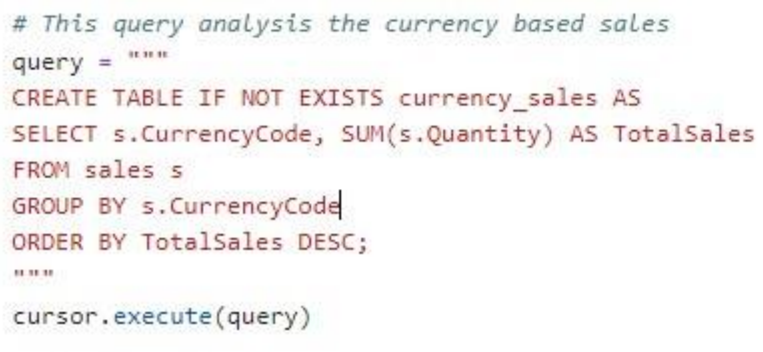
Power BI Visualizations:



3) **Creating the currency_sales Table**:

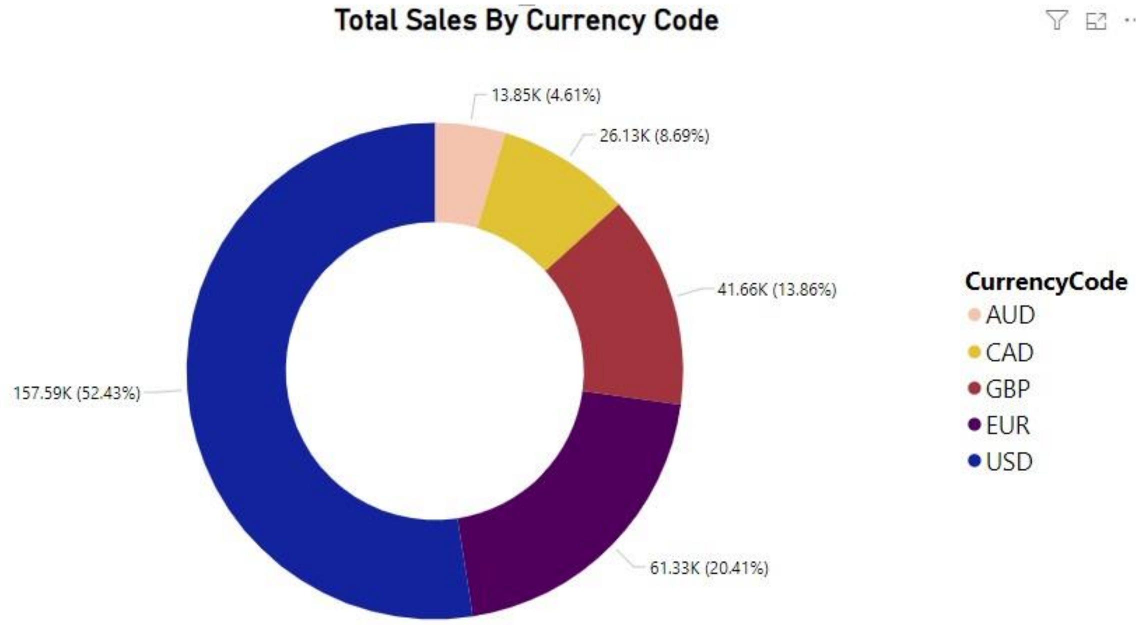
The query creates a table called currency_sales that contains:

- Currency Code: The currency code used in the sales transactions.
- Total Sales: The total quantity of items sold for each currency code.
- This table helps analyse sales performance based on different currencies, providing insights into which currencies are associated with higher sales quantities.



```
# This query analysis the currency based sales
query = """
CREATE TABLE IF NOT EXISTS currency_sales AS
SELECT s.CurrencyCode, SUM(s.Quantity) AS TotalSales
FROM sales s
GROUP BY s.CurrencyCode
ORDER BY TotalSales DESC;
"""
cursor.execute(query)
```

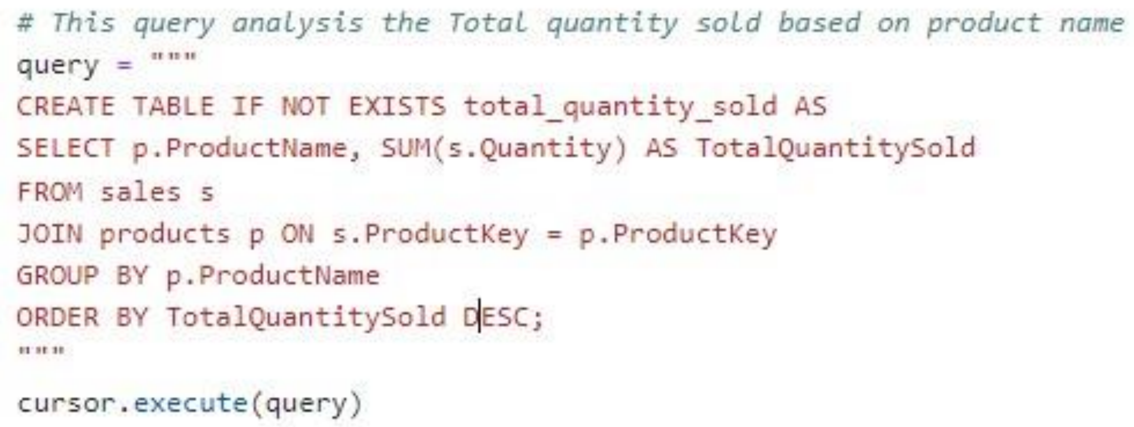
Power BI Visualizations:



4) **Creating the total_quantity_soldTable**:

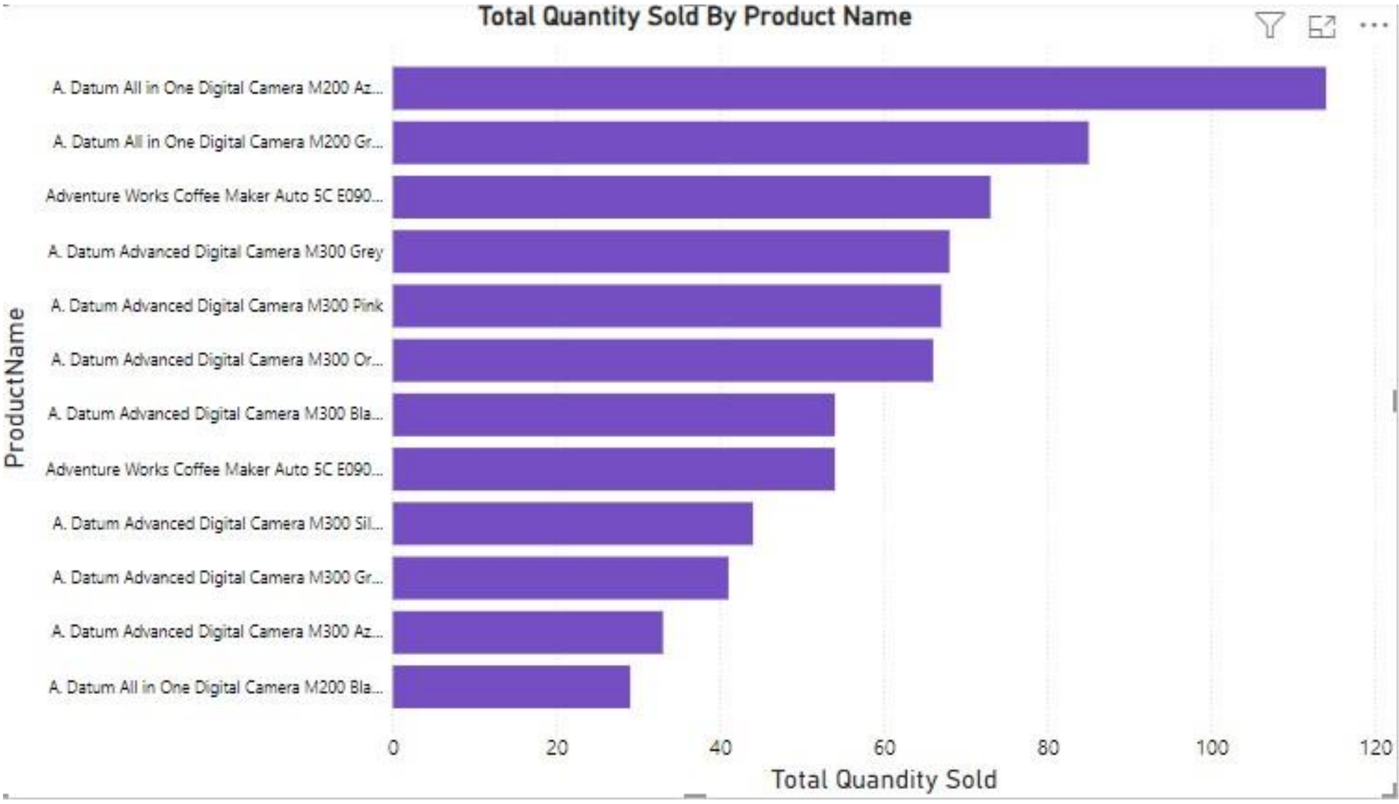
The query creates a table called total_quantity_sold that contains:

- ProductName: The name of each product.
- TotalQuantitySold: The total quantity of each product sold.
- This table provides insight into which products are selling the most, helping in inventory management, sales strategies, and identifying popular products.



```
# This query analysis the Total quantity sold based on product name
query = """
CREATE TABLE IF NOT EXISTS total_quantity_sold AS
SELECT p.ProductName, SUM(s.Quantity) AS TotalQuantitySold
FROM sales s
JOIN products p ON s.ProductKey = p.ProductKey
GROUP BY p.ProductName
ORDER BY TotalQuantitySold DESC;
"""
cursor.execute(query)
```

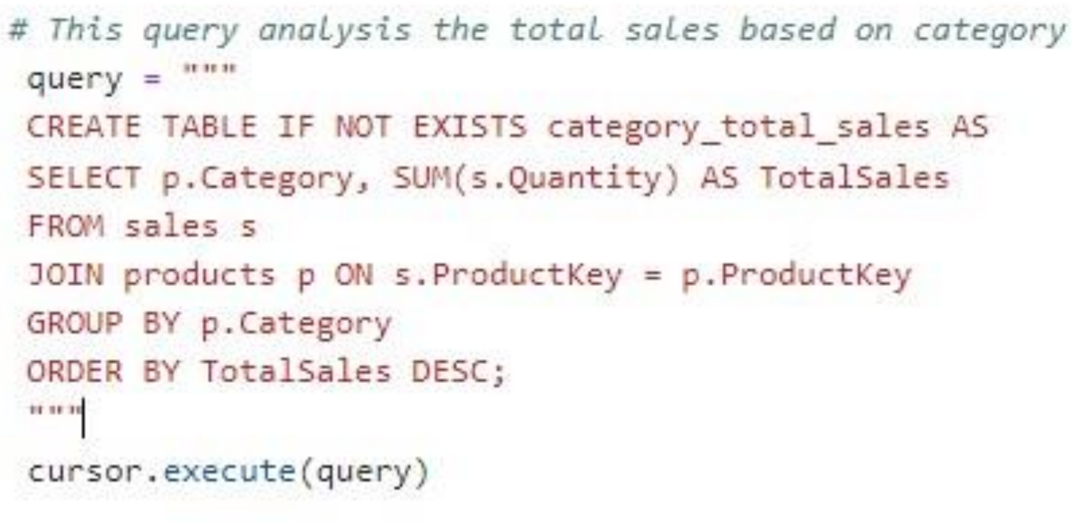
Power BI Visualizations:



5) **Creating the total_quantity_soldTable**:

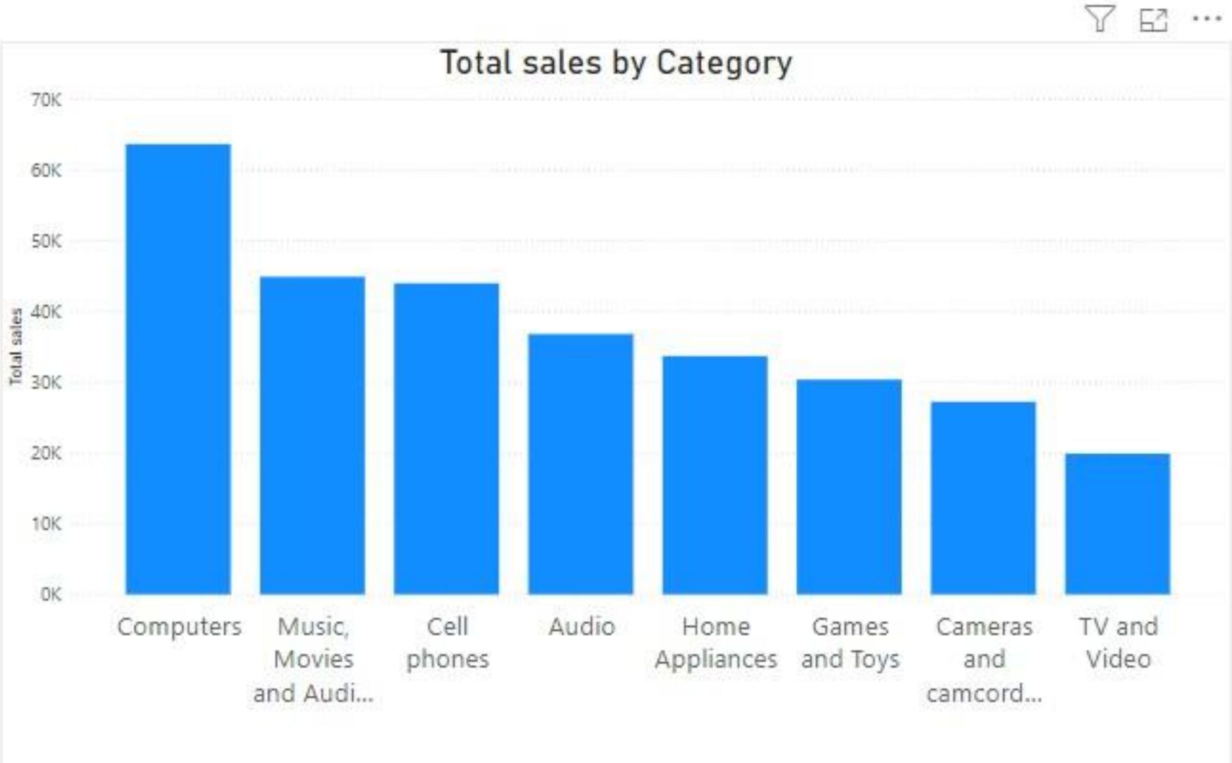
The query creates a table called category_total_sales that contains:

- Category: The category of each product.
- Total Sales: The total quantity of products sold in each category.
- This table provides insight into which product categories are selling the most, helping in category management, marketing strategies, and identifying popular product categories.



```
# This query analysis the total sales based on category
 query = """
 CREATE TABLE IF NOT EXISTS category_total_sales AS
 SELECT p.Category, SUM(s.Quantity) AS TotalSales
 FROM sales s
 JOIN products p ON s.ProductKey = p.ProductKey
 GROUP BY p.Category
 ORDER BY TotalSales DESC;
 """
 cursor.execute(query)
```

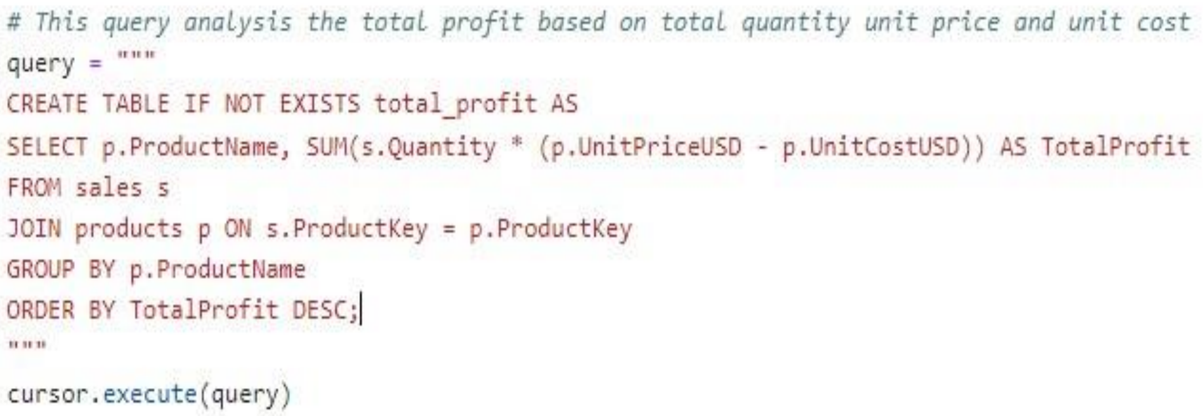
Power BI Visualizations:



6) **Creating the total_profit Table**:

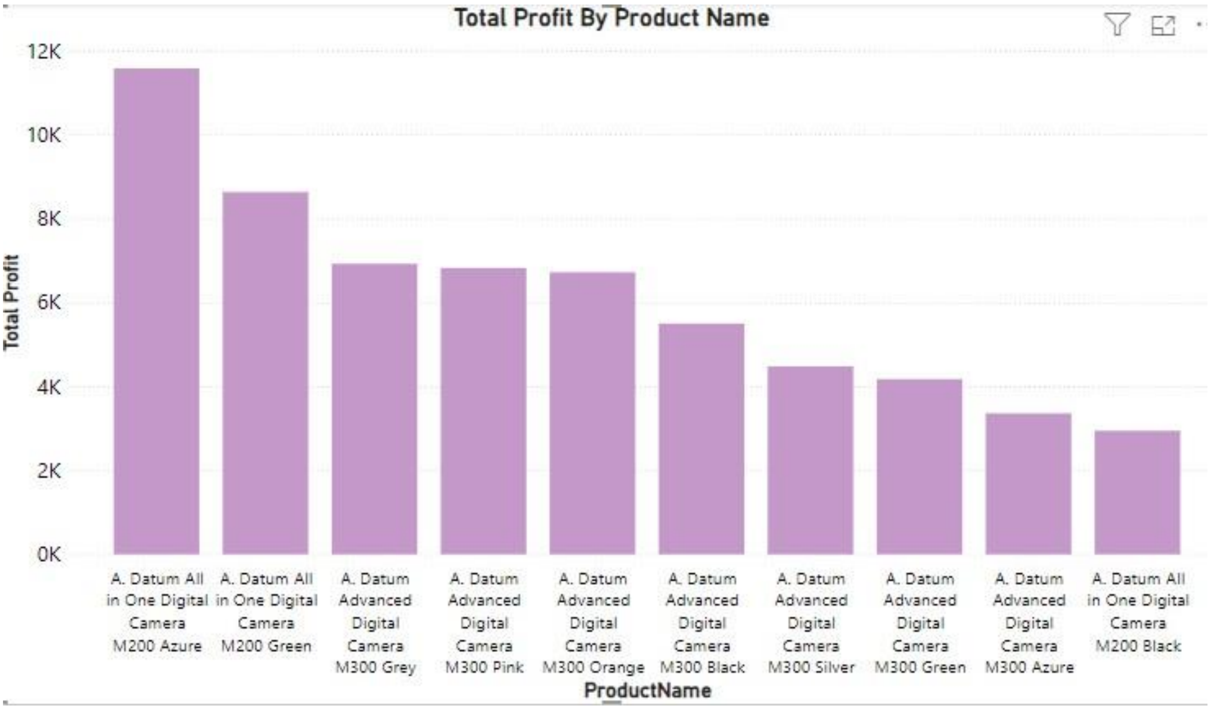
The query creates a table called total_profit that contains:

- ProductName: The name of each product.
- TotalProfit: The total profit generated by each product.
- This table provides insight into the profitability of each product, helping to identify which products contribute the most to the overall profit. This information can be used for strategic decision-making in pricing, marketing, and inventory management.



```
# This query analysis the total profit based on total quantity unit price and unit cost
query = """
CREATE TABLE IF NOT EXISTS total_profit AS
SELECT p.ProductName, SUM(s.Quantity * (p.UnitPriceUSD - p.UnitCostUSD)) AS TotalProfit
FROM sales s
JOIN products p ON s.ProductKey = p.ProductKey
GROUP BY p.ProductName
ORDER BY TotalProfit DESC;
"""
cursor.execute(query)
```

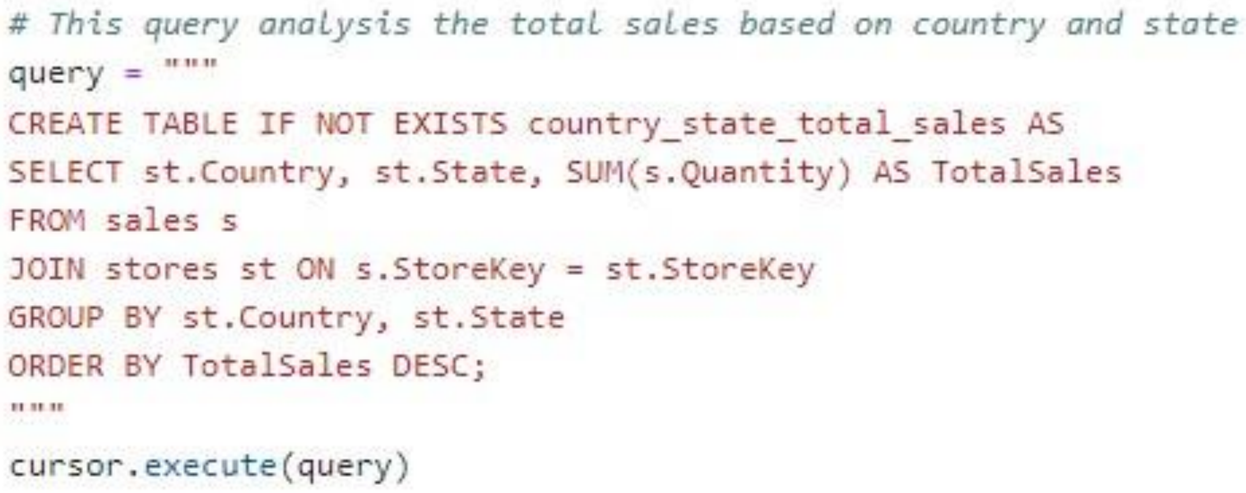
Power BI Visualizations:



7) **Creating the country_state_total_sales Table**:

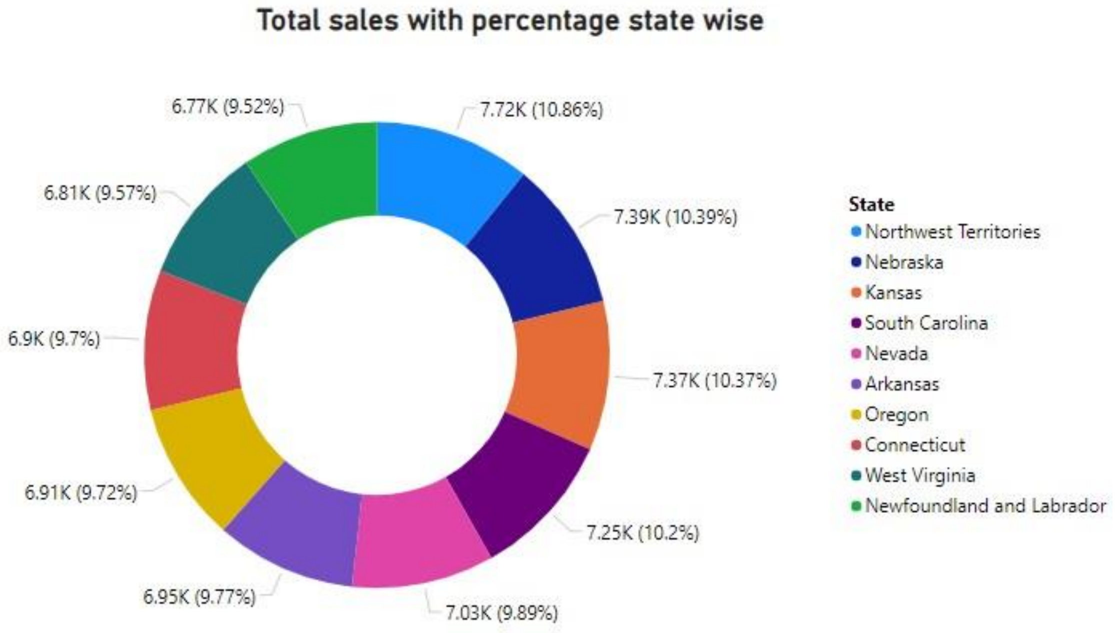
The code creates a new table called country_state_total_sales, which includes:

- Country: The country where the sales occurred.
- State: The state where the sales occurred.
- Total Sales: The total sales (quantity sold) for each combination of country and state.
- This table provides valuable insights into the geographical distribution of sales, aiding in strategic decision-making for market expansion, regional marketing strategies, and inventory allocation.



```
# This query analysis the total sales based on country and state
query = """
CREATE TABLE IF NOT EXISTS country_state_total_sales AS
SELECT st.Country, st.State, SUM(s.Quantity) AS TotalSales
FROM sales s
JOIN stores st ON s.StoreKey = st.StoreKey
GROUP BY st.Country, st.State
ORDER BY TotalSales DESC;
"""
cursor.execute(query)
```

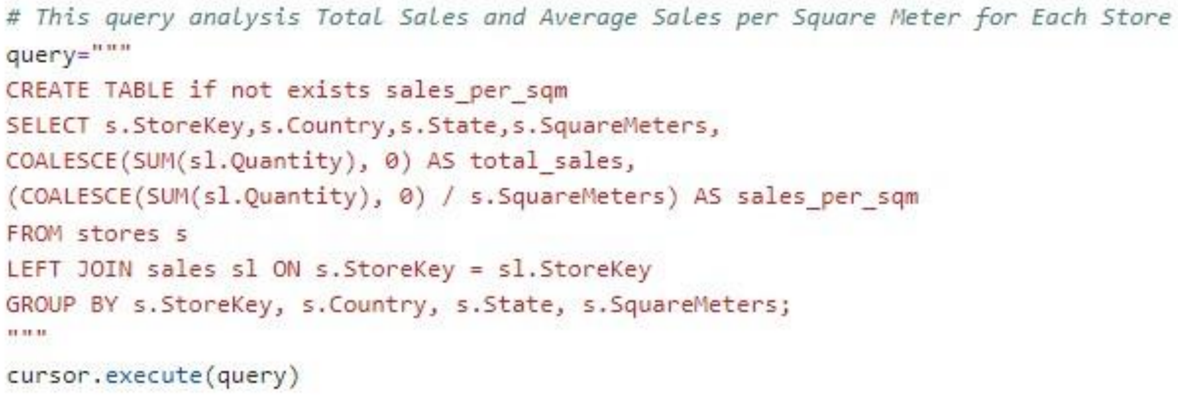
Power BI Visualizations:



8) **Creating the sales_per_sqm Table**:

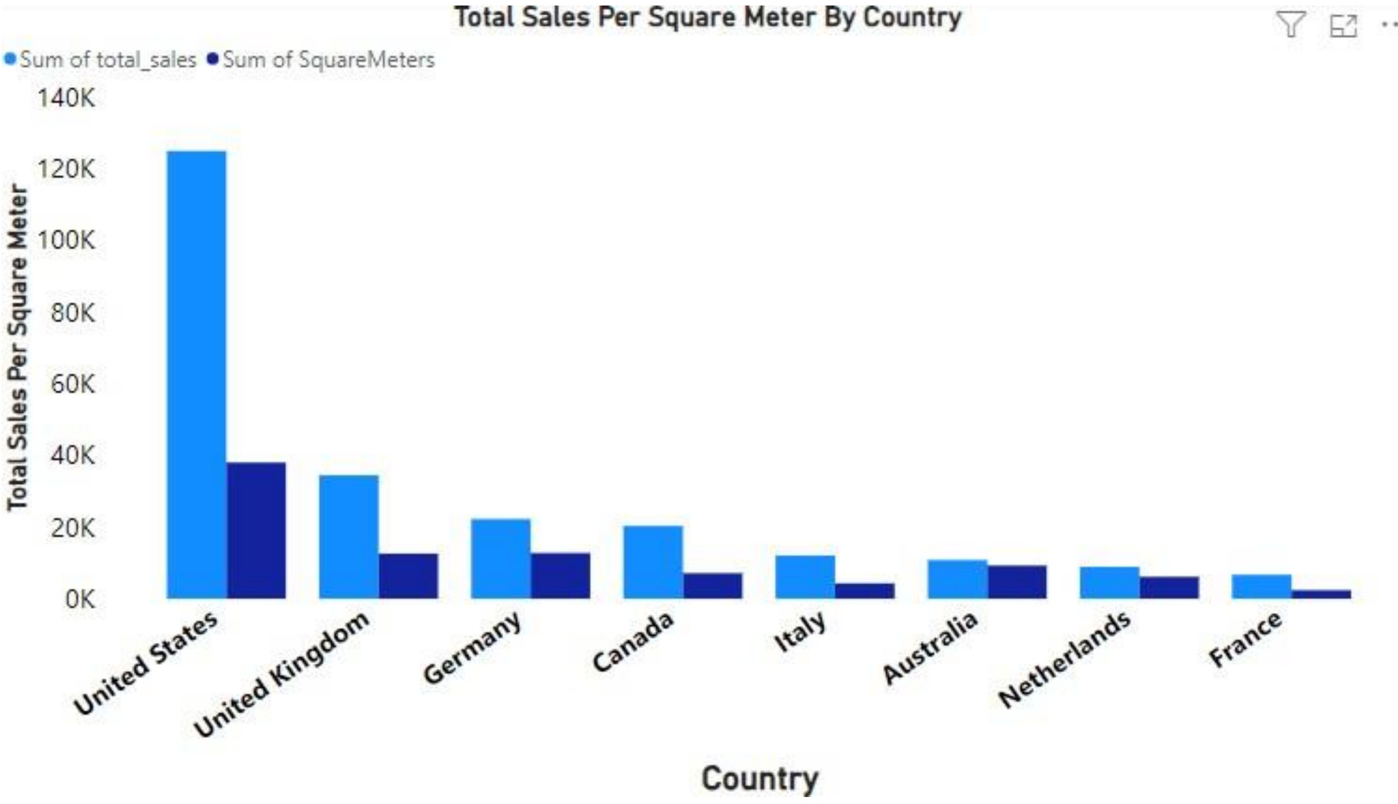
create a new table called sales_per_sqm if it doesn't already exist.

- The query creates a table that provides insights into the sales performance of each store by calculating both the total sales and the sales per square meter for each store.
- It ensures that all stores are included in the analysis, even those with no sales, by using a left join and the COALESCE function. This information can be useful for evaluating the efficiency and productivity of each store based on its size.



```
# This query analysis Total Sales and Average Sales per Square Meter for Each Store
query="""
CREATE TABLE if not exists sales_per_sqm
SELECT s.StoreKey,s.Country,s.State,s.SquareMeters,
COALESCE(SUM(sl.Quantity), 0) AS total_sales,
(COALESCE(SUM(sl.Quantity), 0) / s.SquareMeters) AS sales_per_sqm
FROM stores s
LEFT JOIN sales sl ON s.StoreKey = sl.StoreKey
GROUP BY s.StoreKey, s.Country, s.State, s.SquareMeters;
"""
cursor.execute(query)
```

Power BI Visualizations:

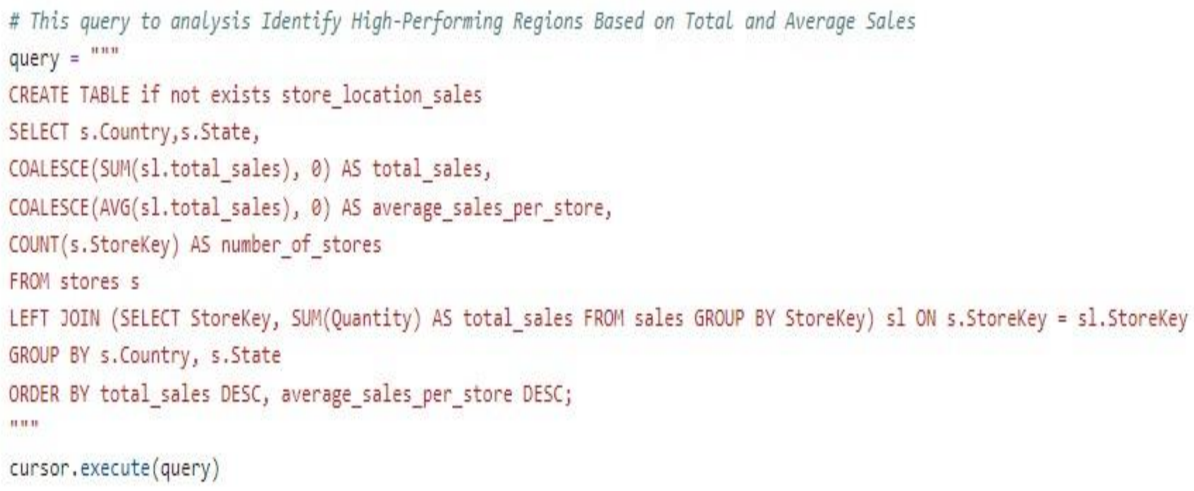


9) **Creating the store_location_sales Table**:

create a new table called store_location_sales if it doesn't already exist.

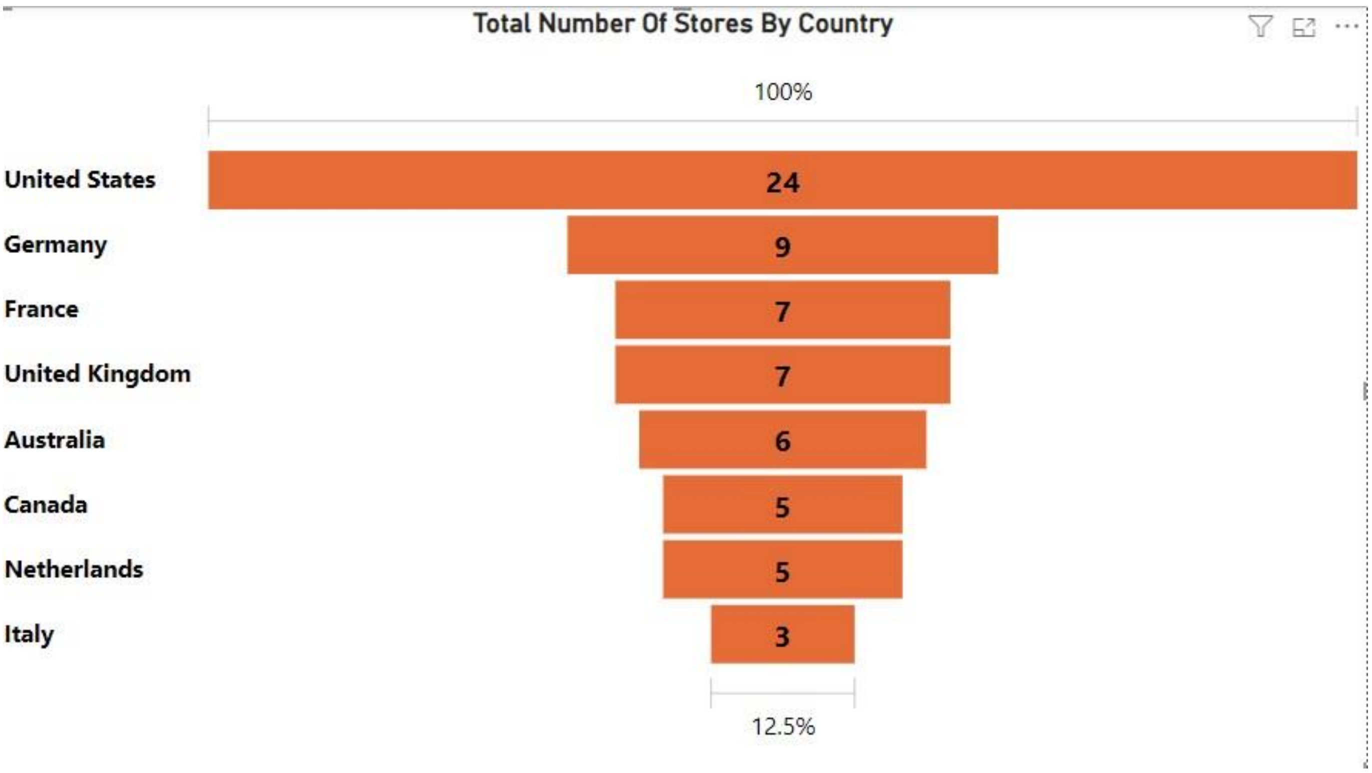
- The table aims to identify high-performing regions based on total and average sales.
- By grouping the data by country and state, summing the total sales, averaging the sales per store, and counting the number of stores, the query provides insights into which regions perform best in terms of sales.

- The results are ordered to highlight the top-performing regions, making it easier to analyze and compare their performance.



```
# This query to analysis Identify High-Performing Regions Based on Total and Average Sales
query = """
CREATE TABLE if not exists store_location_sales
SELECT s.Country,s.State,
COALESCE(SUM(sl.total_sales), 0) AS total_sales,
COALESCE(AVG(sl.total_sales), 0) AS average_sales_per_store,
COUNT(s.StoreKey) AS number_of_stores
FROM stores s
LEFT JOIN (SELECT StoreKey, SUM(Quantity) AS total_sales FROM sales GROUP BY StoreKey) sl ON s.StoreKey = sl.StoreKey
GROUP BY s.Country, s.State
ORDER BY total_sales DESC, average_sales_per_store DESC;
"""
cursor.execute(query)
```

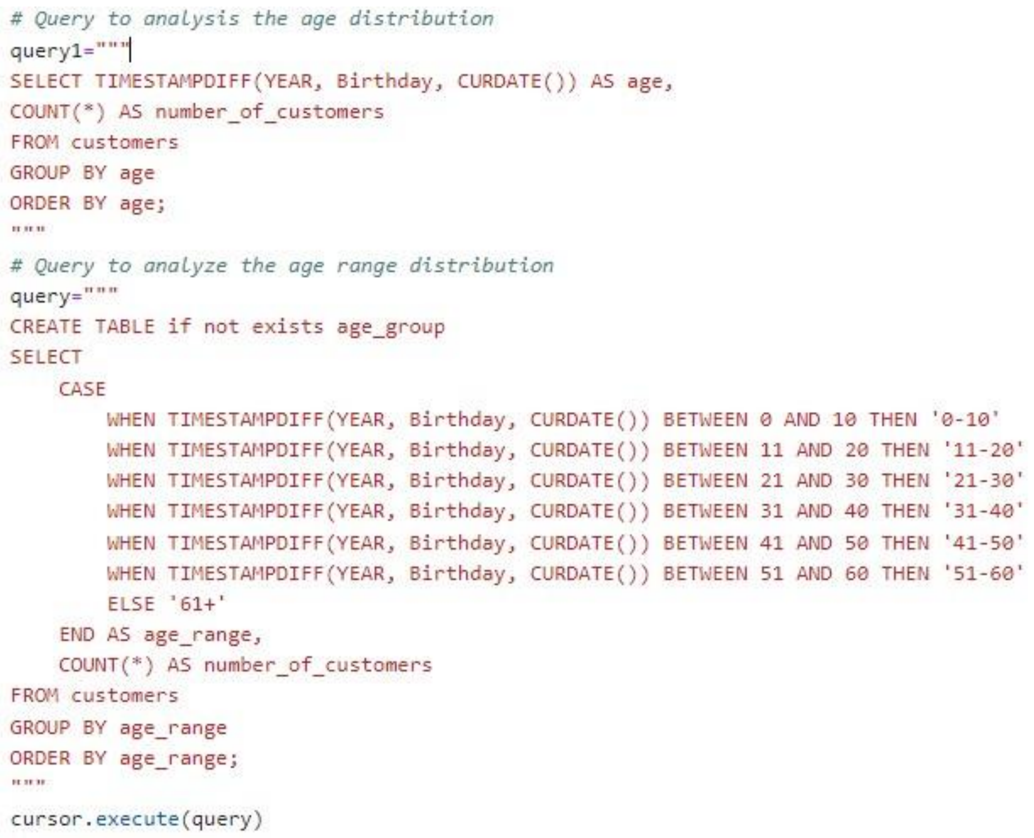
Power BI Visualizations:



10) **Creating the age_group Table**:

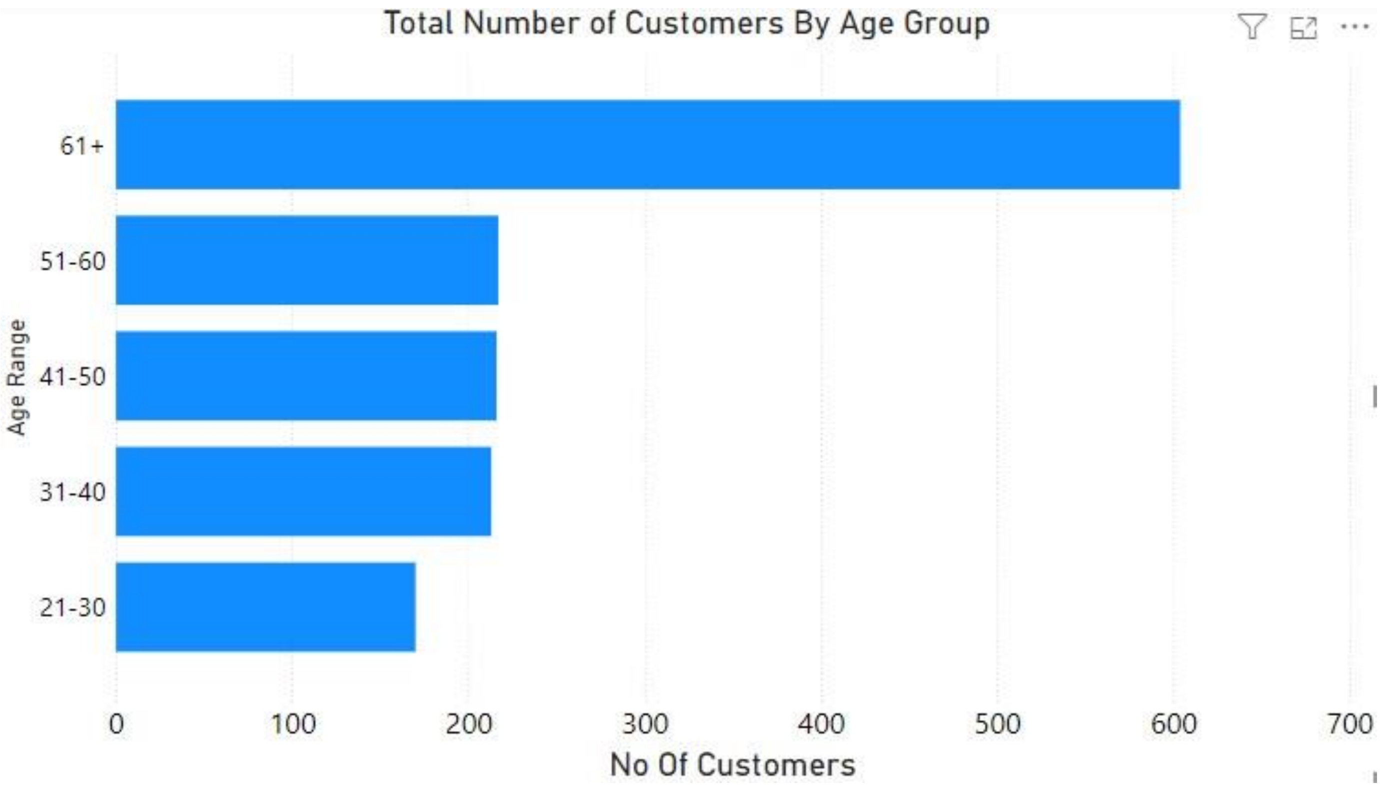
This creates a new table called age_group if it doesn't already exist.

- To calculate the exact age of customers and count the number of customers for each age.
- To categorize customers into predefined age ranges and count the number of customers in each range.



```
# Query to analysis the age distribution
query1="""
SELECT TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) AS age,
COUNT(*) AS number_of_customers
FROM customers
GROUP BY age
ORDER BY age;
"""
# Query to analyze the age range distribution
query="""
CREATE TABLE if not exists age_group
SELECT
    CASE
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 0 AND 10 THEN '0-10'
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 11 AND 20 THEN '11-20'
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 21 AND 30 THEN '21-30'
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 31 AND 40 THEN '31-40'
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 41 AND 50 THEN '41-50'
        WHEN TIMESTAMPDIFF(YEAR, Birthday, CURDATE()) BETWEEN 51 AND 60 THEN '51-60'
        ELSE '61+'
    END AS age_range,
    COUNT(*) AS number_of_customers
FROM customers
GROUP BY age_range
ORDER BY age_range;
"""
cursor.execute(query)
```

Power BI Visualizations:



Main Function:

The main function manages the entire data processing pipeline, from loading and preprocessing data to analyzing and storing the results in a MySQL database. The process includes:

1. Loading data from various sources.
2. Preprocessing the data.
3. Merging the data into a single DataFrame.
4. Detecting and handling outliers.
5. Normalizing the data.
6. Performing data analysis and visualization.
7. Connecting to a MySQL database.

8. Creating analytics tables in the database based on the processed data.

Running the Script

To run the script, executed the following command terminal:

```
python data_spark.py
```

GitHub Link:

https://github.com/vishnupriya08-hub/capstone_projects