Capstone Project

An online retail store is trying to understand the various customer purchase patterns for their firm; you are required to give enough evidence-based insights to provide the same.

Dataset: online_retail.csv

Table of Contents

Problem Statement

Analyse customer purchase behaviour to identify key segments, best-selling products, and revenue-driving countries in order to optimize marketing strategies and inventory planning. Segment customers based on their purchase recency, frequency, and monetary value to design targeted marketing campaigns and improve customer retention. Identify top-performing and underperforming products to improve inventory management and suggest product bundling or promotional strategies

Problem Objective

The primary objective of this project is to perform a comprehensive analysis of customer transactions from a UK-based online retail store to uncover actionable business insights. The dataset contains records of purchases including details such as invoice numbers, stock codes, product descriptions, quantities, unit prices, customer IDs, and countries. The goal is to leverage this data to understand customer purchasing behaviour, identify high-value customers, analyse product performance, and optimize sales strategies.

Key areas of focus include calculating total and time-based sales trends, identifying top-selling and most-returned products, estimating customer lifetime value (CLV), and evaluating customer engagement using RFM (Recency, Frequency, Monetary) analysis. Furthermore, clustering techniques will be applied to segment customers more effectively, enabling targeted marketing and personalized offers. Pricing sensitivity will also be analysed to understand how unit price impacts purchase volumes. The analysis extends to country-level insights, helping the business identify top revenue-generating regions and popular products across different markets.

Ultimately, the insights derived from this analysis aim to support data-driven decision-making in areas like customer retention, inventory management, and revenue optimization, thereby driving overall business growth and operational efficiency.

Data Description

The dataset represents transactional data from a UK-based online retailer between 01/12/2010 and 09/12/2011, capturing purchases made by various customers across different countries. Each row in the dataset represents an individual item-level transaction in a specific invoice.

The various columns of the dataset can be described as follows:

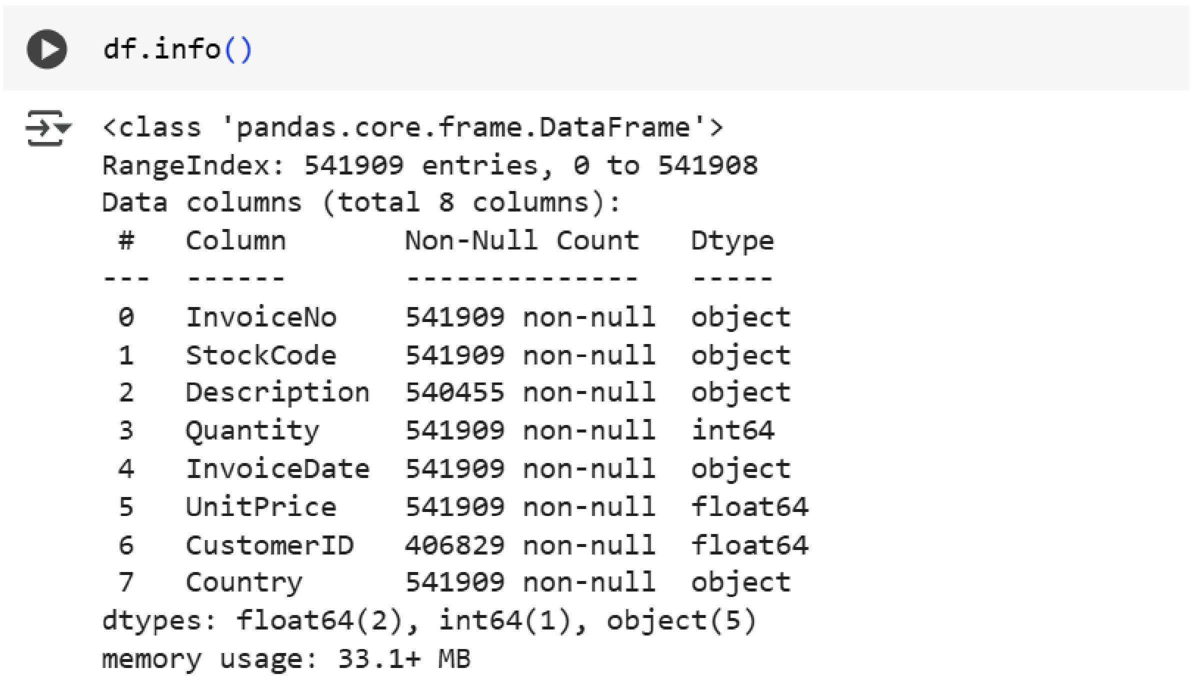
- <u>InvoiceNo:</u> A unique 6-digit identifier for each transaction (invoice). If it starts with 'C', it indicates a cancellation.
- <u>StockCode</u>: A unique identifier (product code) for each distinct product.
- <u>Description</u>: Text description of the product (e.g., "WHITE HANGING HEART T-LIGHT HOLDER").
- <u>Quantity</u>: The number of units purchased (can be negative for returns).
- <u>InvoiceDate</u>: Date and time when the transaction was generated.
- <u>UnitPrice</u>: Price per unit of the product (in British Pounds).
- <u>CustomerID</u>: Unique identifier for each customer. Missing values indicate unidentified customers.
- <u>Country</u>: The country where the customer resides or where the order was shipped.

The dataset contains missing values, especially in CustomerID. Negative Quantity values represent returns or order cancellations. There are multiple countries, so international analysis is possible. The data is transactional and suited for customer segmentation, sales trend analysis, and product performance evaluation.

Data Pre-Processing

Data preprocessing is a critical step in extracting meaningful insights from the Online Retail dataset. Below is a structured breakdown of the data preprocessing steps we should consider:

1. Type conversions:



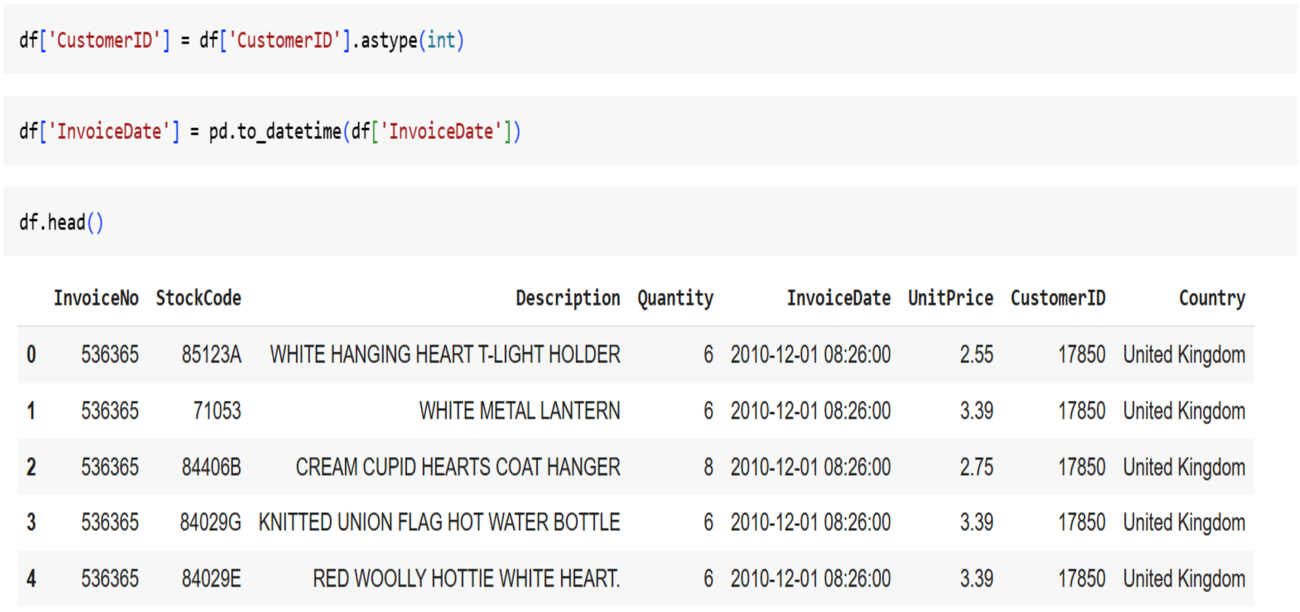
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   InvoiceNo    541909 non-null  object 
 1   StockCode    541909 non-null  object 
 2   Description  540455 non-null  object 
 3   Quantity     541909 non-null  int64  
 4   InvoiceDate  541909 non-null  object 
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object 
dtypes: float64(2), int64(1), object(5)
memory usage: 33.1+ MB
```

The dataset consists of 541,909 entries and 8 columns, capturing detailed transactional data for an online retail store.

- InvoiceNo column is of object type and contains no missing values.
- StockCode column, also an object type, represents the unique product code for each item sold.
- Description column, which provides textual information about the product; however, it has some missing values, with 540455 non-null entries.
- Quantity column is an integer type indicating the number of items purchased per transaction, which can include negative values representing returns.

- **InvoiceDate** column records the timestamp of each transaction but is currently stored as an object and should ideally be converted to a datetime format
- **UnitPrice** column is of float type and reflects the price per unit of the product.
- **CustomerID** column is also a float type and identifies the customer associated with each transaction, but it contains a significant number of missing values—only 406,829 entries are non-null.
- **Country** column, stored as an object, indicates the country where the transaction was made.



```
df['CustomerID'] = df['CustomerID'].astype(int)
```

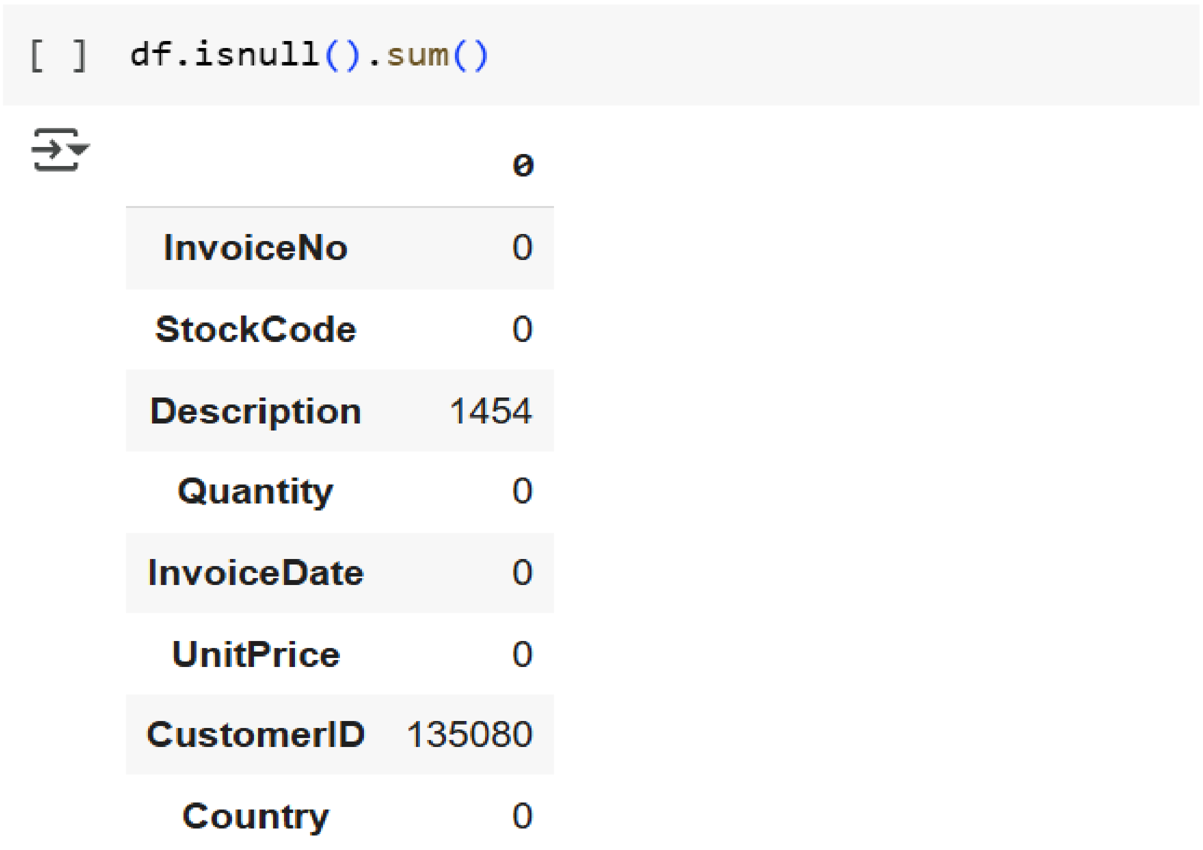
```
df['InvoiceDate'] = pd.to_datetime(df['InvoiceDate'])
```

```
df.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850	United Kingdom

- Converts the CustomerID column from float to integer type.
- Converts the InvoiceDate column from object/string to a proper datetime format.

2. Handling Null values:



```
[ ]  df.isnull().sum()
```

	0
InvoiceNo	0
StockCode	0
Description	1454
Quantity	0
InvoiceDate	0
UnitPrice	0
CustomerID	135080
Country	0

Displays the result of checking missing values in the dataset using df.isnull().sum(). We find out that the columns Description & CustomerID contains missing values.

- Description: 1,454 missing values
  Some products are missing their descriptions. These may correspond to cancellations, errors, or special cases.
- CustomerID: 135,080 missing values
  A significant portion of entries don't have a CustomerID.

```
df.dropna(inplace=True)
```

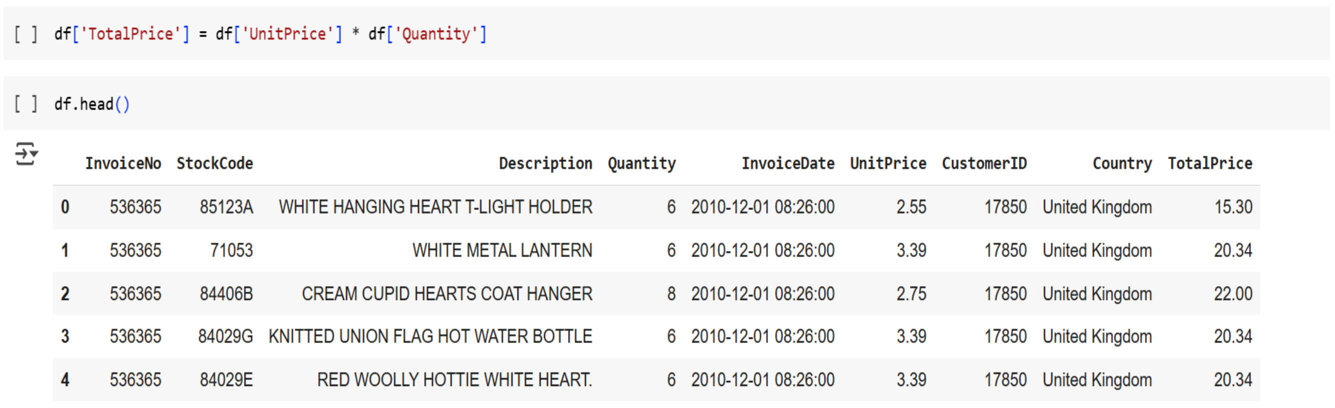
```
df.shape
```

```
(406829, 8)
```

- df.dropna() this command removes all rows with any missing values from the DataFrame df.
- inplace=True means the changes are made directly to the original dataframe, so we no need to assign it back.
- In our dataset, this would have removed rows where Description or CustomerID was missing (as identified earlier).
- df.shape shows the new shape (dimensions) of the DataFrame after dropping missing values.

The dataset now has **406,829 rows** and **8 columns**. This means over **135,000** rows were dropped (from the original 541,909 rows). The DataFrame is now cleaned of missing data, making it ready for accurate feature engineering, modelling, or clustering.

3. **Create TotalPrice Column:**



```
[ ] df['TotalPrice'] = df['UnitPrice'] * df['Quantity']
```

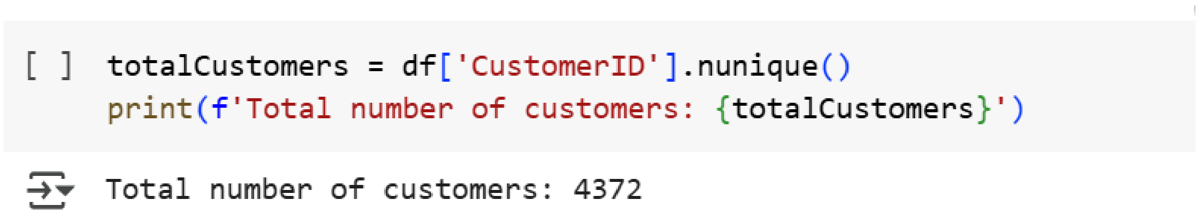
```
[ ] df.head()
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	TotalPrice
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850	United Kingdom	15.30
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	20.34
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850	United Kingdom	22.00
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	20.34
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850	United Kingdom	20.34

The image shows the creation of a new column named **TotalPrice** in a retail transaction dataset. It is calculated by multiplying the unit price of a product (**UnitPrice**) with the quantity purchased (**Quantity**). This represents the total revenue for that product line in a single invoice row.

Data Analysis and Insights

1. Total number of customers:



```
[ ] totalCustomers = df['CustomerID'].nunique()
    print(f'Total number of customers: {totalCustomers}')
```

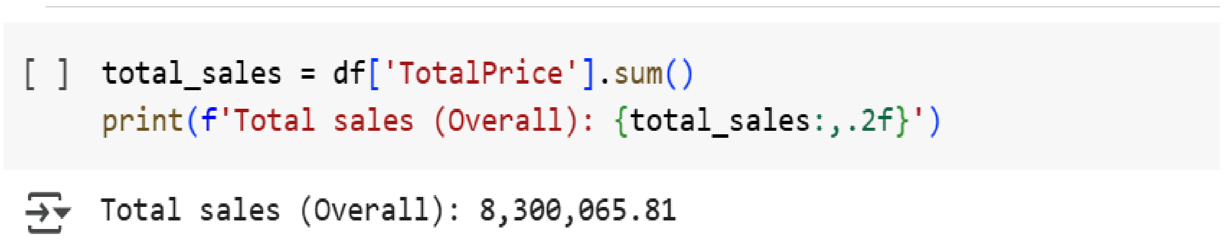
```
Total number of customers: 4372
```

The code in the image calculates the total number of unique customers in the dataset based on the CustomerID column.

- **df['CustomerID']:** Accesses the CustomerID column from the DataFrame df.
- **nunique():** Counts the number of distinct (unique) values in that column.

This gives us the total number of individual customers, regardless of how many purchases each made i.e. The dataset contains **4,372** unique customers.

2. Total Cumulative Sales:



```
[ ] total_sales = df['TotalPrice'].sum()
    print(f'Total sales (Overall): {total_sales:,.2f}')
```

```
Total sales (Overall): 8,300,065.81
```

The code in the image calculates the total overall sales revenue based on the TotalPrice column of the DataFrame.

- **df['TotalPrice']:** Accesses the column that likely contains the sales amount for each transaction (e.g. unit price × quantity).
- **sum():** Adds up all values in the TotalPrice column.
- Result is stored in the variable **total_sales.**

The **total revenue** generated from all transactions in the dataset is **8,300,065.81** (likely pounds).

3. Total Sales on Monthly basis:

```
data.set_index('InvoiceDate', inplace=True)
```

```
monthly_sales = data.resample('M')['TotalPrice'].sum()
print(f'Monthly Sales :{monthly_sales}')
```

```
Monthly Sales :InvoiceDate
2010-12-31     554604.020
2011-01-31     475074.380
2011-02-28     436546.150
2011-03-31     579964.610
2011-04-30     426047.851
2011-05-31     648251.080
2011-06-30     608013.160
2011-07-31     574238.481
2011-08-31     616368.000
2011-09-30     931440.372
2011-10-31     974603.590
2011-11-30    1132407.740
2011-12-31     342506.380
```

The code in the image calculates **monthly sales revenue** using a DataFrame containing a **TotalPrice** column and an InvoiceDate column.

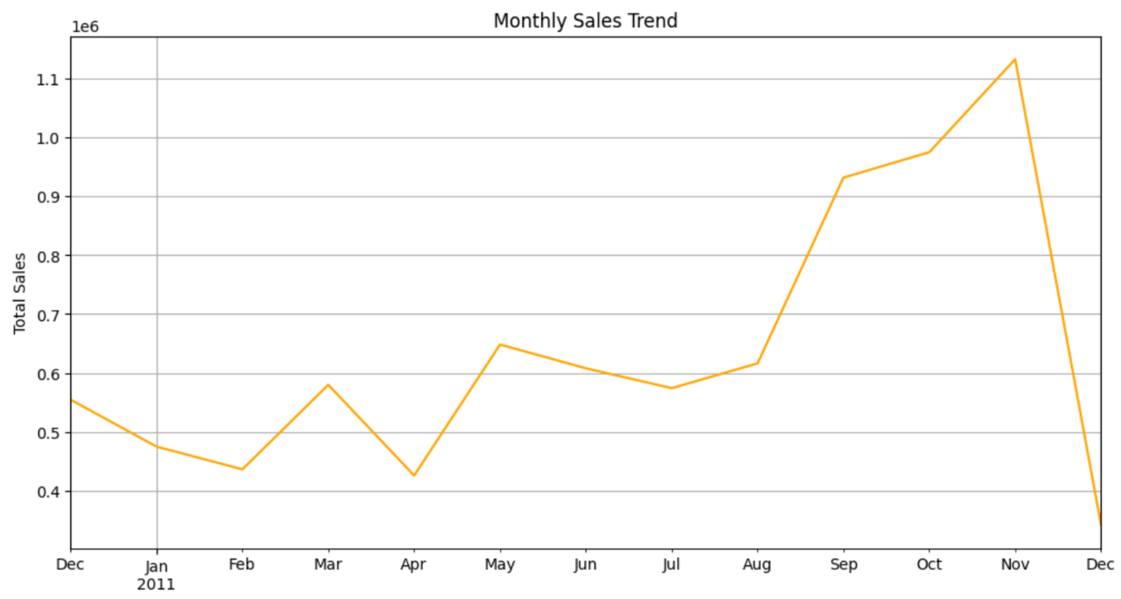
- It sets the **InvoiceDate** column as the **index** — this is required for time-based operations like resample.
- **resample('M'):** Groups data by month-end frequency.

- **'M'** stands for **month-end**, so 2010-12-31, 2011-01-31, etc.
- **['TotalPrice']:** Selects the column that contains transaction totals.
- **sum():** Aggregates the total sales for each month.

Finally, we get the total sales based on each month.

Plotting Monthly Sales

```
[ ]  plt.figure(figsize=(12, 6))
     monthly_sales.plot(color='orange')
     plt.title('Monthly Sales Trend')
     plt.xlabel('Month')
     plt.ylabel('Total Sales')
     plt.grid(True)
```



Trend Analysis

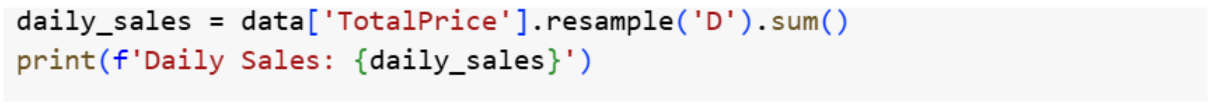
- **Early 2011 (Jan–Apr)**: Sales fluctuate slightly and stay below 600,000, even dipping to around 430,000 in April.
- **May to July**: Sales begin to rise steadily, reaching over 600,000.
- **August to November**: There's a significant spike:
  - September: ~931,000
  - October: ~974,000

        - November: Peaks at ~1.13 million — the highest month of the year
            - This is likely due to holiday season or promotional sales .
- **December 2011**: Sales drop sharply to the lowest point (~340,000), possibly due to missing data or year-end business shutdowns.

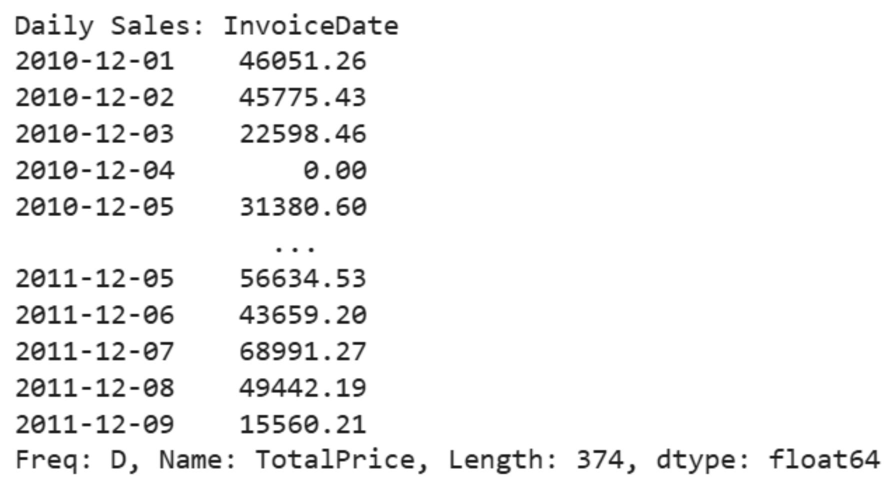
Business Insights

- **Strongest months**: September to November — ideal for marketing pushes and stock optimization.
- **Weakest month**: December requires investigation (e.g., data issues or natural drop).
- **Planning**: Use this trend for inventory, staffing, and promotional planning.

4. Total Sales on Daily basis:



```python
daily_sales = data['TotalPrice'].resample('D').sum()
print(f'Daily Sales: {daily_sales}')
```



```
Daily Sales: InvoiceDate
2010-12-01    46051.26
2010-12-02    45775.43
2010-12-03    22598.46
2010-12-04        0.00
2010-12-05    31380.60
                ...
2011-12-05    56634.53
2011-12-06    43659.20
2011-12-07    68991.27
2011-12-08    49442.19
2011-12-09    15560.21
Freq: D, Name: TotalPrice, Length: 374, dtype: float64
```

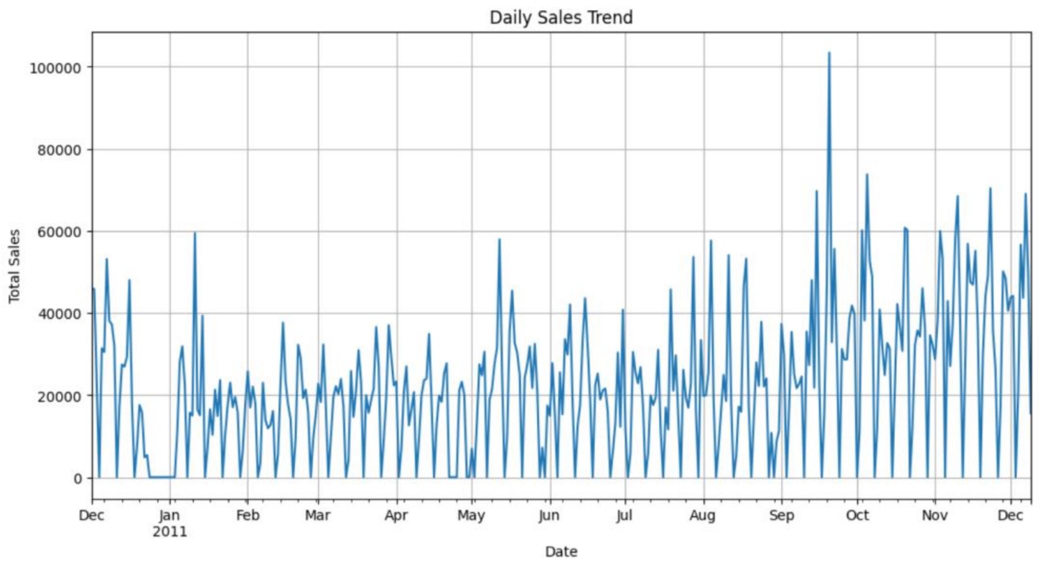
The code in the image calculates **monthly sales revenue** using a DataFrame containing a **TotalPrice** column and an InvoiceDate column.

- **data['TotalPrice']:** Selects the column containing the total price (sales amount).

- **resample('D'):** Resamples the data to daily frequency ('D' = day), using the datetime index ('InvoiceDate').
- **sum():** Aggregates the total sales for each day by summing all transactions on that day.
- **Result**: daily_sales is a time series of daily sales totals.

Plotting Daily Sales

```
plt.figure(figsize=(12, 6))
daily_sales.plot()
plt.title('Daily Sales Trend')
plt.xlabel('Date')
plt.ylabel('Total Sales')
plt.grid(True)
```



- **Late 2010/Early 2011 (Dec- Jan):** Sales are present but exhibit some periods of zero or very low sales, perhaps indicating non-operating days or very slow periods. There's a notable dip to zero sales right at the beginning of January 2011.
- **February- August 2011:** Sales show consistent daily fluctuations, generally ranging from 0 to around 40,000-60,000. There are regular periods of very low or zero sales, possibly weekends or

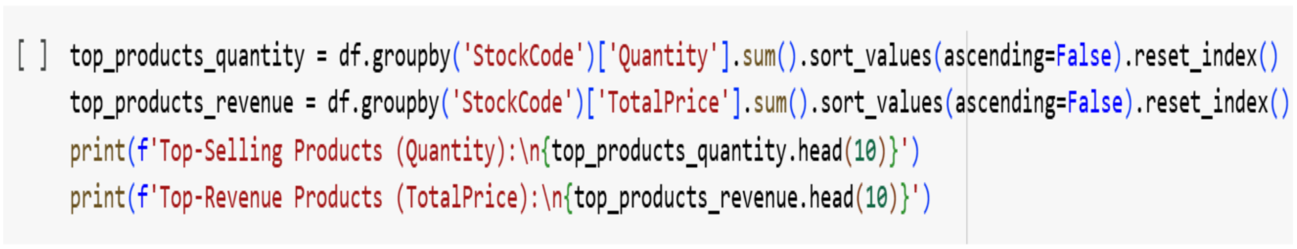
holidays. We can see some peaks, for example, in late May/early June, reaching close to 60,000.

- Late August/Early September- November 2011: This period shows a significant increase in both the average daily sales and the peak sales. Several days in September and October record sales exceeding 80,000, with an all-time high peak reaching over 100,000 sales in late September/early October. The overall sales volume appears much higher during these months compared to the earlier part of the year.
- December 2011: Sales remain relatively high, similar to the November period, often reaching peaks of 60,000 to 70,000, although not consistently as high as the peak in September/October.

Key Observations:

- Fluctuating Daily Sales: There is considerable day-to-day variability in sales, with frequent drops to zero or near-zero, suggesting either non-business days (weekends, holidays) or periods of extremely low demand.
- Growth in Second Half of the Year: The most prominent feature is the substantial increase in sales volume from late summer through the autumn, with a clear peak in late September/early October. This could indicate seasonal demand, marketing campaigns, or a general growth in business during this period.
- Peak Sales Period: The period from late September to early October marks the highest sales figures observed throughout the year.

5. Top 10 Selling Products by quantity and revenue:



```
[ ] top_products_quantity = df.groupby('StockCode')['Quantity'].sum().sort_values(ascending=False).reset_index()
    top_products_revenue = df.groupby('StockCode')['TotalPrice'].sum().sort_values(ascending=False).reset_index()
    print(f'Top-Selling Products (Quantity):\n{top_products_quantity.head(10)}')
    print(f'Top-Revenue Products (TotalPrice):\n{top_products_revenue.head(10)}')
```

- df.groupby('StockCode'): This groups the DataFrame df by the unique values in the 'StockCode' column. 'StockCode' likely represents a unique identifier for each product.
- ['Quantity'].sum(): After grouping by 'StockCode', this selects the 'Quantity' column for each group and calculates the sum of 'Quantity' for all entries belonging to that specific 'StockCode'. This effectively gives the total quantity sold for each product.
- sort_values(ascending=False): This sorts the resulting sums of quantities in descending order (from highest to lowest). So, the products with the highest total quantity sold will appear first.
- reset_index(): After grouping and sorting, 'StockCode' would typically become the index of the resulting Series. reset_index() converts this index back into a regular column, making 'StockCode' a column again, and creates a default integer index.

In summary, this line calculates the total quantity sold for each product (identified by 'StockCode') and stores them in a new DataFrame called top_products_quantity, sorted from the most to least sold by quantity.

Top products by revenue is very similar to the first one, but instead of 'Quantity', it operates on the 'TotalPrice' column.

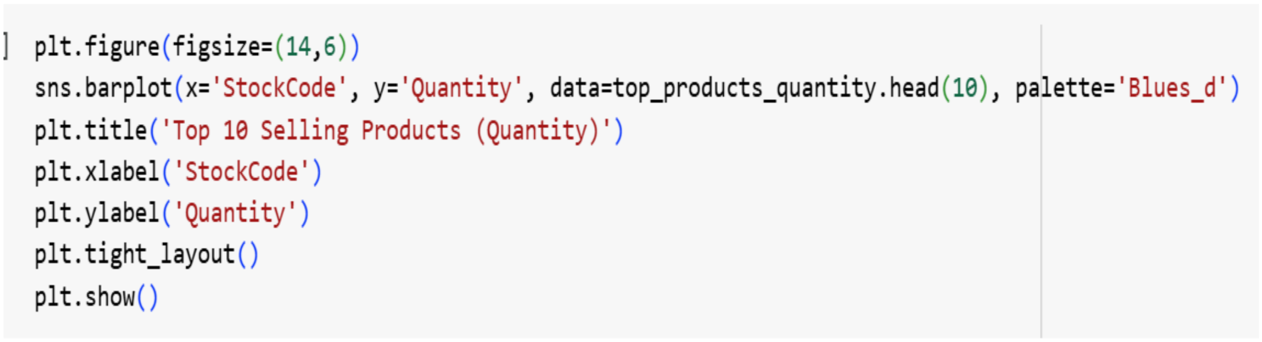
- df.groupby('StockCode'): Groups the data by product.
- ['TotalPrice'].sum(): Calculates the sum of 'TotalPrice' for each product. 'TotalPrice' likely represents the revenue generated by each line item in the sales data. Summing it up for each

'StockCode' gives the total revenue generated by that specific product.

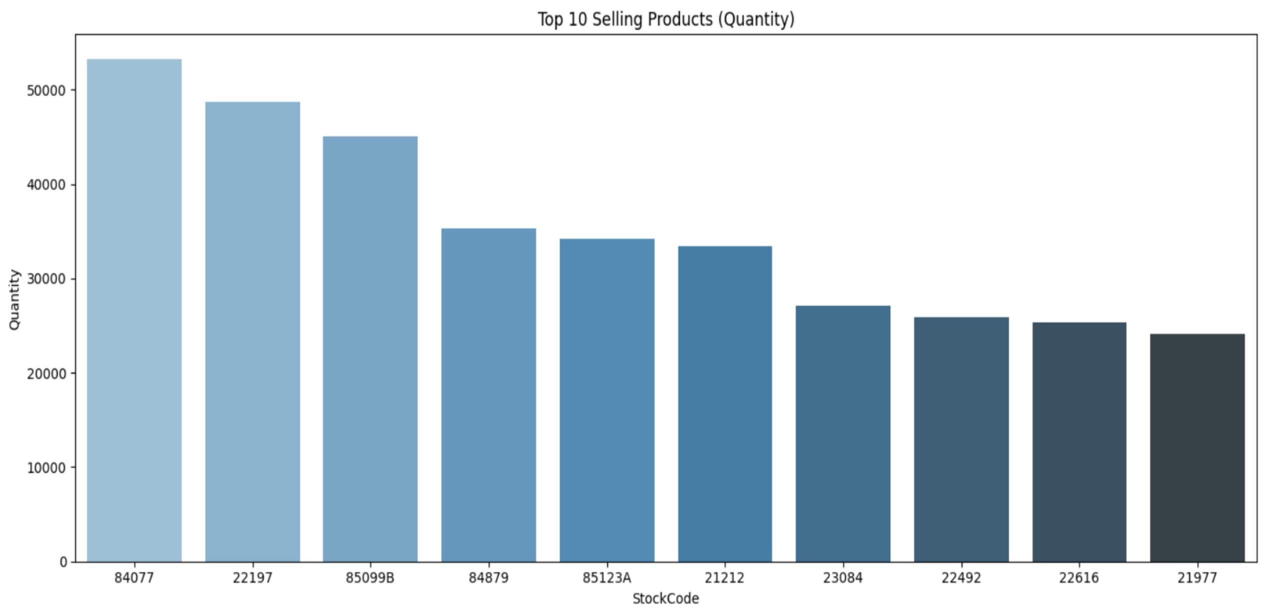
- **sort_values(ascending=False):** Sorts the total revenues in descending order, placing the products that generated the most revenue at the top.
- **reset_index():** Converts 'StockCode' from an index back to a column.

In summary, this line calculates the total revenue generated by each product (identified by 'StockCode') and stores them in a new DataFrame called top_products_revenue, sorted from the highest to lowest revenue.

Plotting Top 10 Selling Products by quantity:



```
] plt.figure(figsize=(14,6))
  sns.barplot(x='StockCode', y='Quantity', data=top_products_quantity.head(10), palette='Blues_d')
  plt.title('Top 10 Selling Products (Quantity)')
  plt.xlabel('StockCode')
  plt.ylabel('Quantity')
  plt.tight_layout()
  plt.show()
```

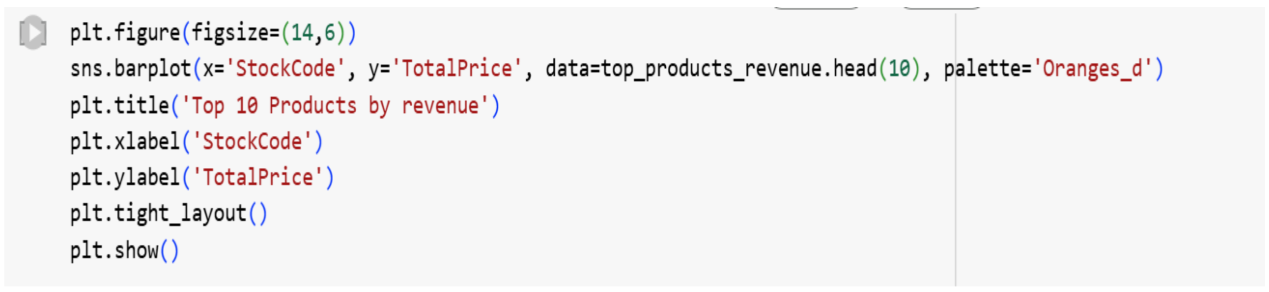


Bars and their significance: Each bar corresponds to a specific StockCode and its height indicates the total Quantity sold for that product. The bars are arranged in descending order of quantity, from left to right.

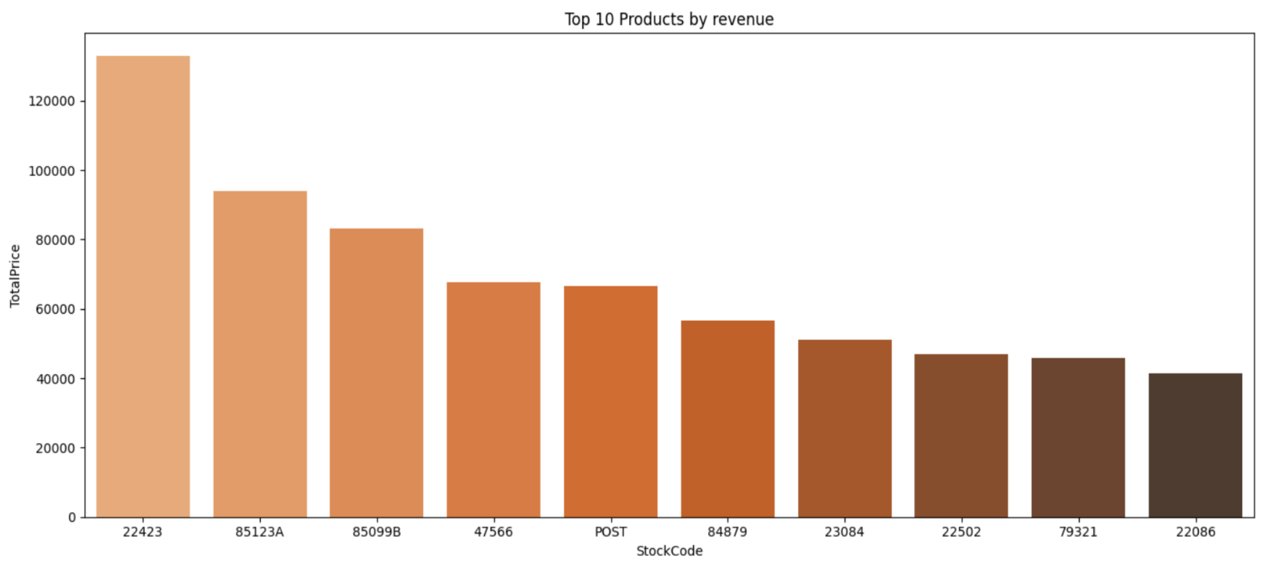
- The tallest bar on the left (StockCode **84077**) shows the highest quantity sold, slightly above 50,000 units.
- The second tallest bar (StockCode **22197**) sold nearly 50,000 units.
- The quantities gradually decrease for subsequent products.
- The shortest bar on the far right (StockCode **21977**) represents the 10th top-selling product, with a quantity just below 25,000 units.

It clearly highlights that product **84077** is the best-selling product by quantity, followed closely by **22197** and **85099B**.

Plotting Top 10 Selling Products by revenue:



```
plt.figure(figsize=(14,6))
sns.barplot(x='StockCode', y='TotalPrice', data=top_products_revenue.head(10), palette='Oranges_d')
plt.title('Top 10 Products by revenue')
plt.xlabel('StockCode')
plt.ylabel('TotalPrice')
plt.tight_layout()
plt.show()
```

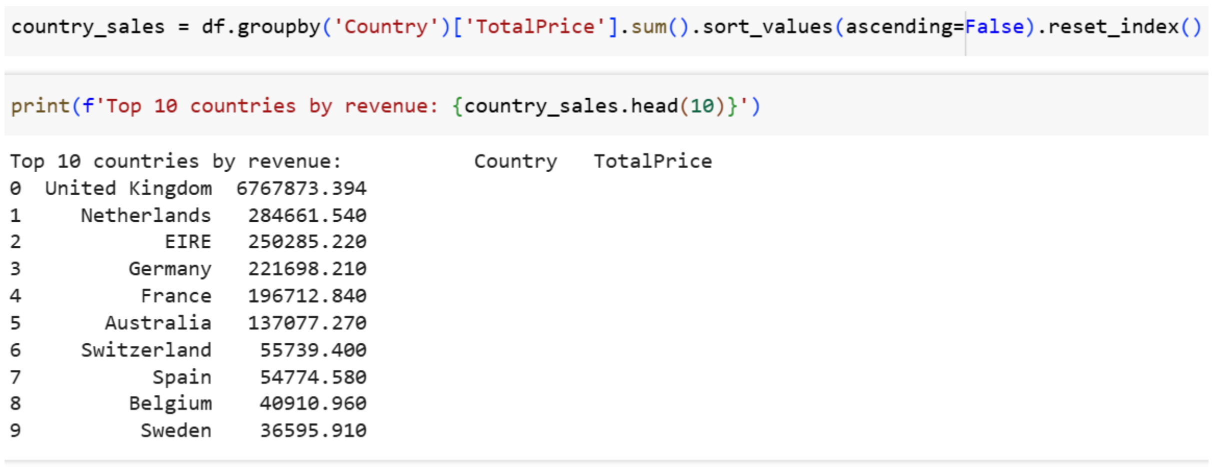


Bars and their meaning: Each bar corresponds to a specific StockCode, and its height represents the total revenue generated by that product. The bars are arranged in descending order, from the highest revenue on the left to the lowest on the right.

- The tallest bar on the left, corresponding to **StockCode 22423**, shows the highest revenue, significantly exceeding **120,000**.
- The second highest revenue is generated by **StockCode 85123A**, which is around **95,000**.
- **StockCode 85099B** is the third highest, generating over **80,000** in revenue.
- The revenue gradually decreases for the subsequent products, with the 10th product (**22086**) generating just over **40,000**.

It's clear that **StockCode 22423** is the top revenue generator by a significant margin, followed by **85123A** and **85099B**.

6. Top 10 countries by revenue:



```
country_sales = df.groupby('Country')['TotalPrice'].sum().sort_values(ascending=False).reset_index()
```

```
print(f'Top 10 countries by revenue: {country_sales.head(10)}')
```

```
Top 10 countries by revenue:          Country  TotalPrice
0  United Kingdom  6767873.394
1     Netherlands   284661.540
2            EIRE   250285.220
3         Germany   221698.210
4          France   196712.840
5       Australia   137077.270
6     Switzerland    55739.400
7           Spain    54774.580
8         Belgium    40910.960
9          Sweden    36595.910
```

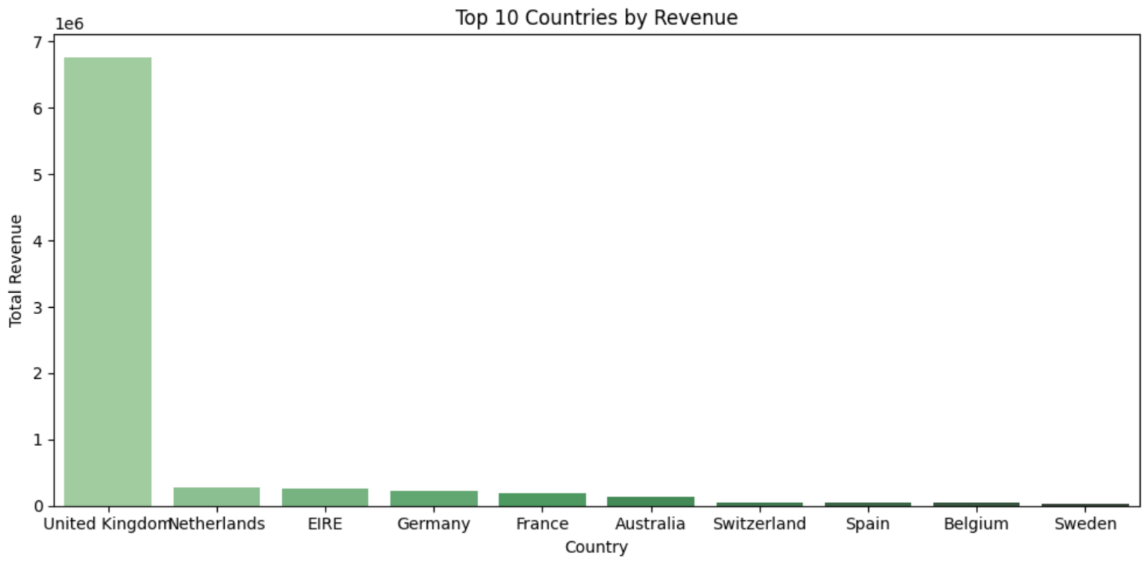
- **df.groupby('Country'):** This operation groups the DataFrame df by unique values in the 'Country' column. This means that all rows belonging to the same country will be grouped together.

- **['TotalPrice'].sum():** After grouping by 'Country', this selects the 'TotalPrice' column for each group and calculates the sum of 'TotalPrice' for all entries within that specific country. 'TotalPrice' likely represents the revenue for each transaction or line item. Summing it up for each country effectively gives the total revenue generated from that country.
- **sort_values(ascending=False):** This sorts the resulting sums of 'TotalPrice' in descending order. This means that countries with the highest total revenue will appear first.
- **reset_index():** After grouping and sorting, 'Country' would typically become the index of the resulting pandas Series. reset_index() converts this index back into a regular column, making 'Country' a column again, and adds a default integer index (0, 1, 2, ...).

In summary, this line calculates the total revenue generated from each country and stores the results in a new DataFrame called country_sales, sorted from the highest to the lowest revenue.

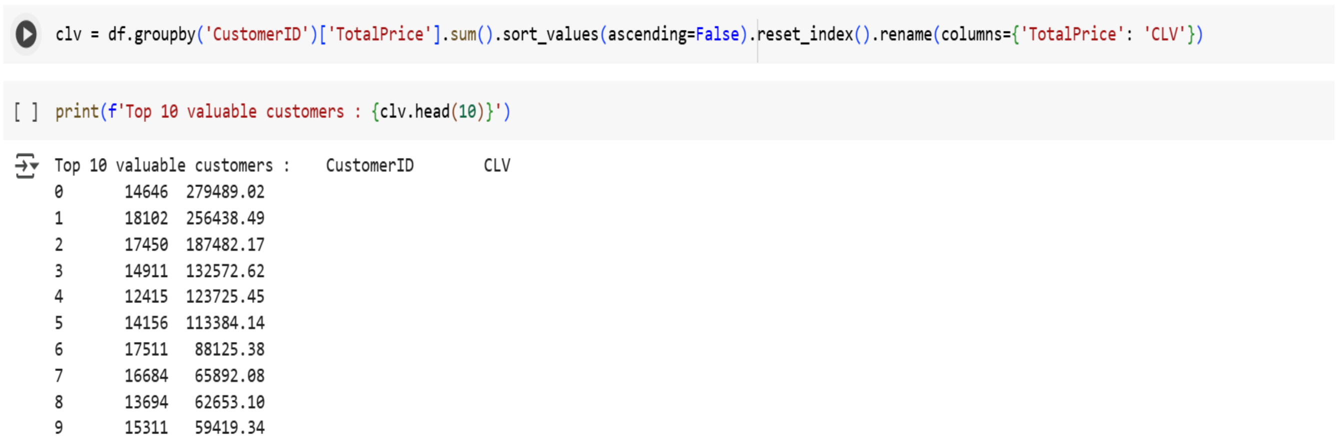
Plotting top 10 countries by revenue

```
plt.figure(figsize=(10,5))
sns.barplot(x='Country', y='TotalPrice', data=country_sales.head(10), palette='Greens_d')
plt.title('Top 10 Countries by Revenue')
plt.xlabel('Country')
plt.ylabel('Total Revenue')
plt.tight_layout()
plt.show()
```



- **1 United Kingdom 6767873.394**: The United Kingdom is the top country by revenue, generating over 6.7 million (currency implied, likely GBP if sales data is from UK).
- **2 Netherlands 284661.540**: Netherlands is second, with revenue significantly lower than the UK but still substantial at over 284 thousand.
- **3 EIRE 250285.220**: EIRE (Republic of Ireland) follows closely with over 250 thousand in revenue.
- **4 Germany 221698.210**: Germany is next with over 221 thousand.
- **5 France 196712.840**: France generated over 196 thousand.
- **6 Australia 137077.270**: Australia with over 137 thousand.
- **7 Switzerland 55739.400**: Switzerland with over 55 thousand.
- **8 Spain 54774.580**: Spain with over 54 thousand.
- **9 Belgium 40910.960**: Belgium with over 40 thousand.
- **10 Sweden 36595.910**: Sweden with over 36 thousand.

7. Top 10 Customers:


```
clv = df.groupby('CustomerID')['TotalPrice'].sum().sort_values(ascending=False).reset_index().rename(columns={'TotalPrice': 'CLV'})
```

```
[ ] print(f'Top 10 valuable customers : {clv.head(10)}')
```

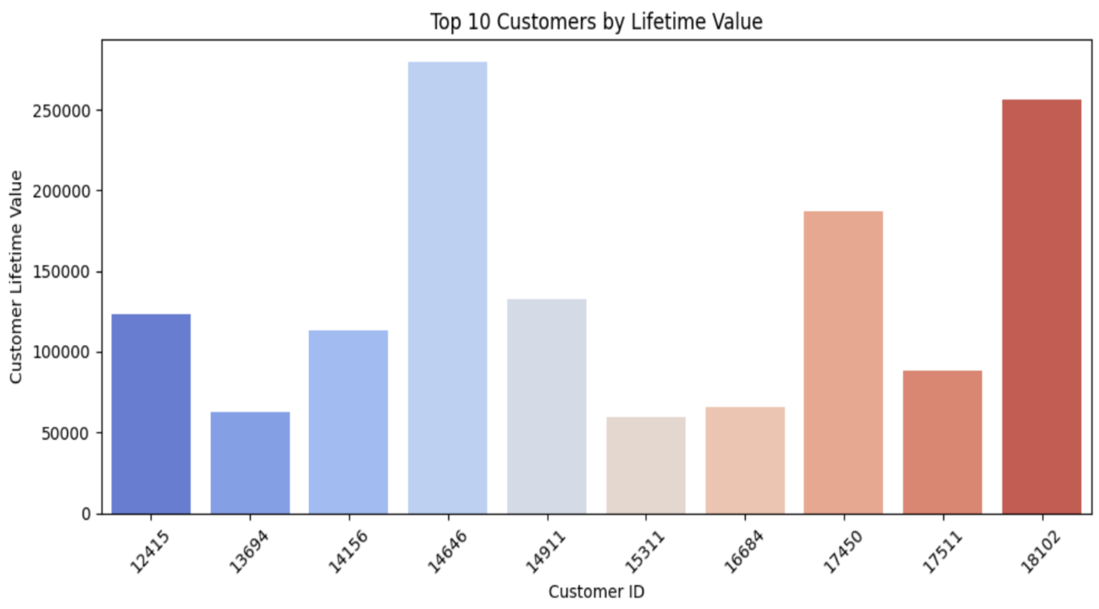
```
Top 10 valuable customers :    CustomerID        CLV
0       14646  279489.02
1       18102  256438.49
2       17450  187482.17
3       14911  132572.62
4       12415  123725.45
5       14156  113384.14
6       17511   88125.38
7       16684   65892.08
8       13694   62653.10
9       15311   59419.34
```

- **df.groupby('CustomerID'):** This groups the DataFrame df by the unique values in the 'CustomerID' column. This means all transactions from the same customer will be grouped together.
- **['TotalPrice'].sum()**: For each customer group, this calculates the sum of the 'TotalPrice' column. 'TotalPrice' likely represents the revenue from a single transaction or line item. Summing it up gives the total revenue generated by each customer.
- **sort_values(ascending=False):** This sorts the resulting total revenues in descending order, placing the customers who generated the most revenue at the top.
- **reset_index():** This converts the 'CustomerID' (which became the index during the grouping operation) back into a regular column and creates a new default integer index (0, 1, 2, ...).
- **rename(columns={'TotalPrice': 'CLV'})**: This is a crucial step. It renames the 'TotalPrice' column to 'CLV'. 'CLV' stands for Customer Lifetime Value, which in this simplified context, is being calculated as the total revenue a customer has generated over the period covered by the data.

In summary, we calculate the total revenue for each customer, sort them from highest to lowest, and store the result in a new DataFrame called **clv** with a column named 'CLV' to represent this value.

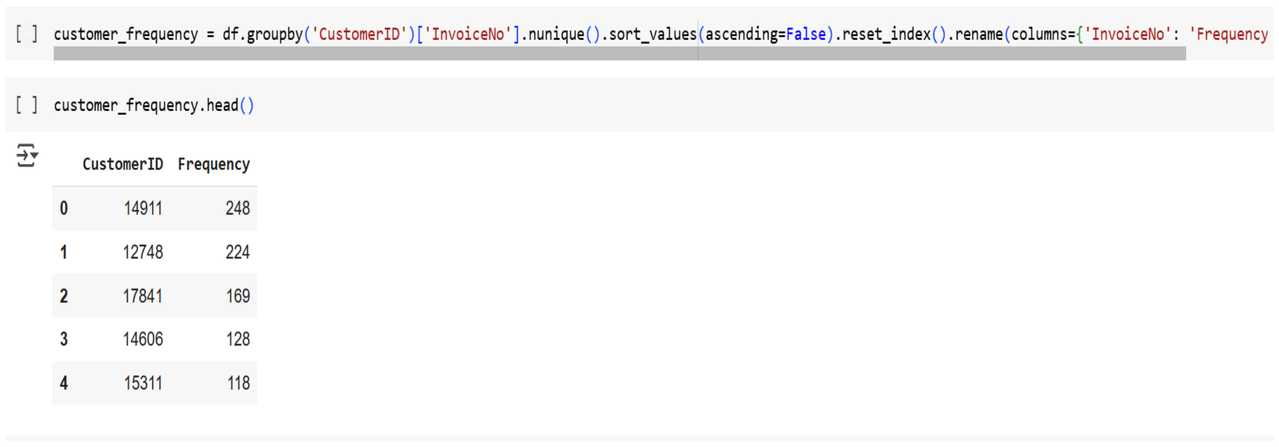
Plotting top 10 customers

```
plt.figure(figsize=(10, 5))
sns.barplot(data=clv.head(10), x='CustomerID', y='CLV', palette='coolwarm')
plt.title('Top 10 Customers by Lifetime Value')
plt.xlabel('Customer ID')
plt.ylabel('Customer Lifetime Value')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



- The tallest bar, corresponding to Customer ID **14646**, shows the highest Customer Lifetime Value, reaching approximately 280,000.
- The second tallest bar, Customer ID **18102**, has a value of approximately 250,000.
- Other bars show the lifetime values of the remaining customers in the top 10, with values ranging from just under 200,000 down to around 60,000.

8. Top 10 Customers by Frequency (repeat purchase rate):



```
[ ] customer_frequency = df.groupby('CustomerID')['InvoiceNo'].nunique().sort_values(ascending=False).reset_index().rename(columns={'InvoiceNo': 'Frequency
```

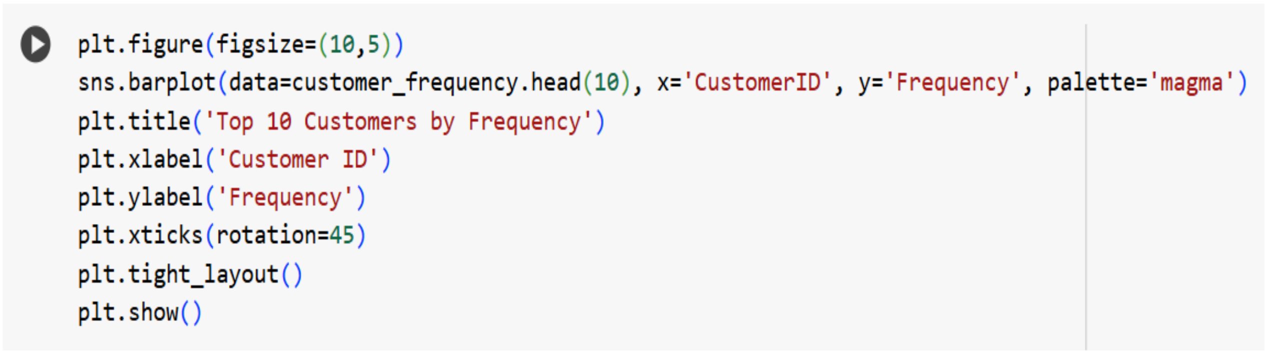
```
[ ] customer_frequency.head()
```

```
   CustomerID  Frequency
0       14911        248
1       12748        224
2       17841        169
3       14606        128
4       15311        118
```

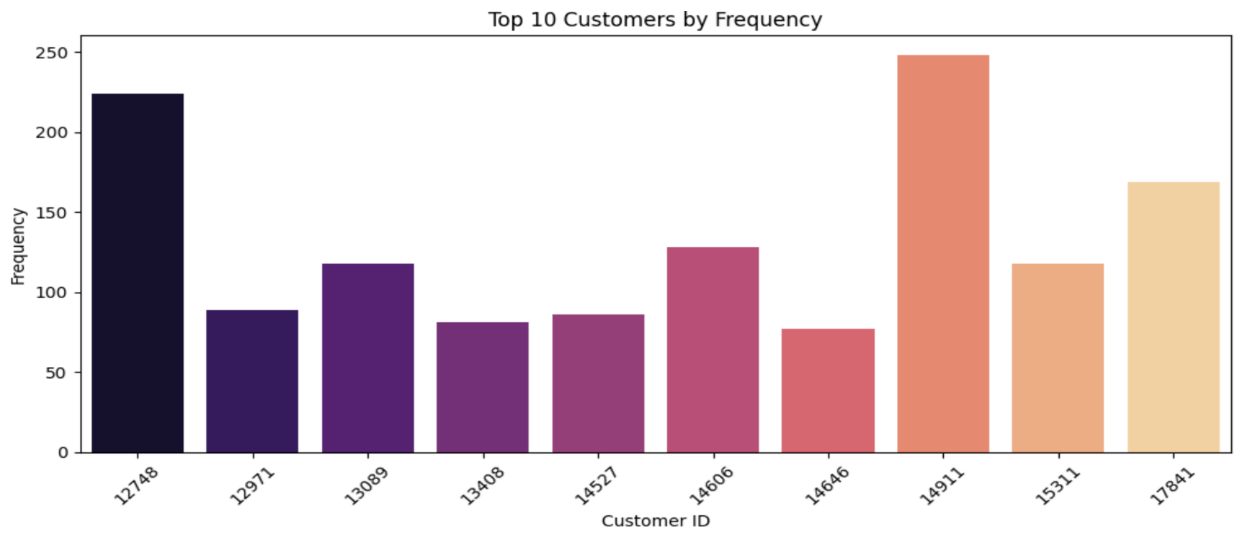
- **df.groupby('CustomerID'):** This groups the DataFrame df by the unique values in the **'CustomerID'** column. This means all transactions from the same customer will be grouped together.
- **['InvoiceNo'].nunique():** For each customer group, this selects the 'InvoiceNo' column and calculates the number of unique values (**nunique()**) in that column. InvoiceNo is a unique identifier for each purchase transaction or invoice. Counting the unique invoice numbers for each customer tells you how many separate purchases that customer made. This is a measure of their purchase frequency.
- **sort_values(ascending=False):** This sorts the resulting counts (frequencies) in descending order, placing the customers with the highest number of purchases at the top.
- **reset_index():** This converts the **'CustomerID'** (which became the index during the grouping operation) back into a regular column and creates a new default integer index.
- **rename(columns={'InvoiceNo': 'Frequency'}):** This renames the column that resulted from the **nunique()** operation (originally 'InvoiceNo') to a more descriptive name, 'Frequency'.

In summary, we calculate the number of unique purchases (frequency) for each customer, sort them from most frequent to least frequent, and store the result in a new DataFrame called customer_frequency.

Plotting top 10 frequent customers

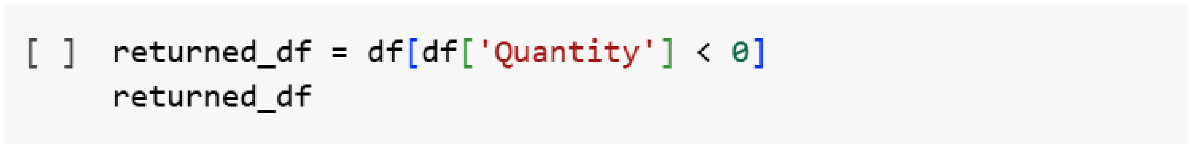


```
plt.figure(figsize=(10,5))
sns.barplot(data=customer_frequency.head(10), x='CustomerID', y='Frequency', palette='magma')
plt.title('Top 10 Customers by Frequency')
plt.xlabel('Customer ID')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



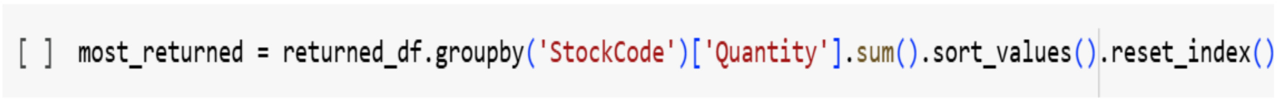
- The tallest bar, corresponding to Customer ID **14911**, represents the most frequent customer, with a frequency of nearly 250 purchases.
- The second tallest bar, Customer ID **12748**, has a frequency of over 200 purchases.
- The third tallest bar, Customer ID **17841**, has a frequency of over 150 purchases.
- The shortest bars represent customers with lower purchase frequencies, but still within the top 10.

9. Top 10 Products Returned:



```
[ ]  returned_df = df[df['Quantity'] < 0]
     returned_df
```

In summary, this line creates a new DataFrame called **returned_df** that contains only the rows from the original **df** where the value in the 'Quantity' column is negative. In a sales dataset, a negative quantity typically signifies a return or a cancelled order.



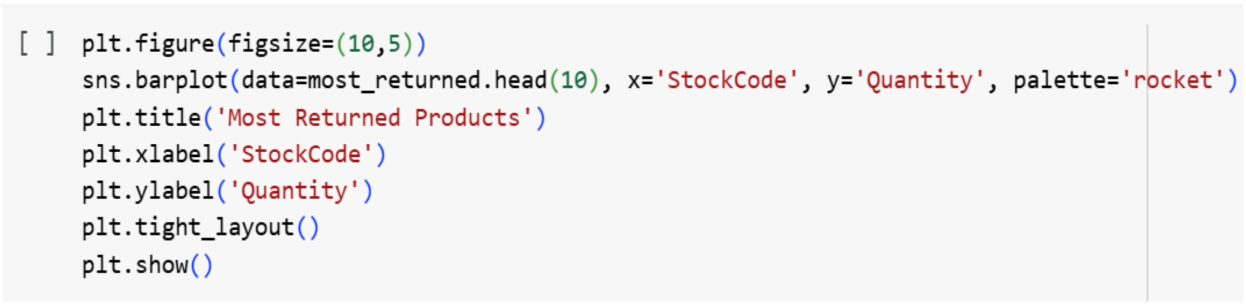
```
[ ]  most_returned = returned_df.groupby('StockCode')['Quantity'].sum().sort_values().reset_index()
```

- **returned_df:** This is the DataFrame created in a previous step (as seen in the earlier image), which contains only the rows where the Quantity is negative, representing returned items.
- **groupby('StockCode'):** This groups the returned_df by the unique values in the **'StockCode'** column. This means all returned items with the same product identifier will be grouped together.
- **['Quantity'].sum():** After grouping by **'StockCode'**, this selects the **'Quantity'** column and calculates the sum. Since the Quantity values in this DataFrame are negative, summing them up will result in a negative total. The magnitude of this negative number represents the total number of units returned for that product. For example, if a product was returned 5 times, and the quantity for each return was-1, the sum would be-5.
- **sort_values():** This sorts the resulting sums. Crucially, the ascending parameter is not specified, so it defaults to **ascending=True**. This means it will sort the values from the most negative (smallest number) to the least negative (largest

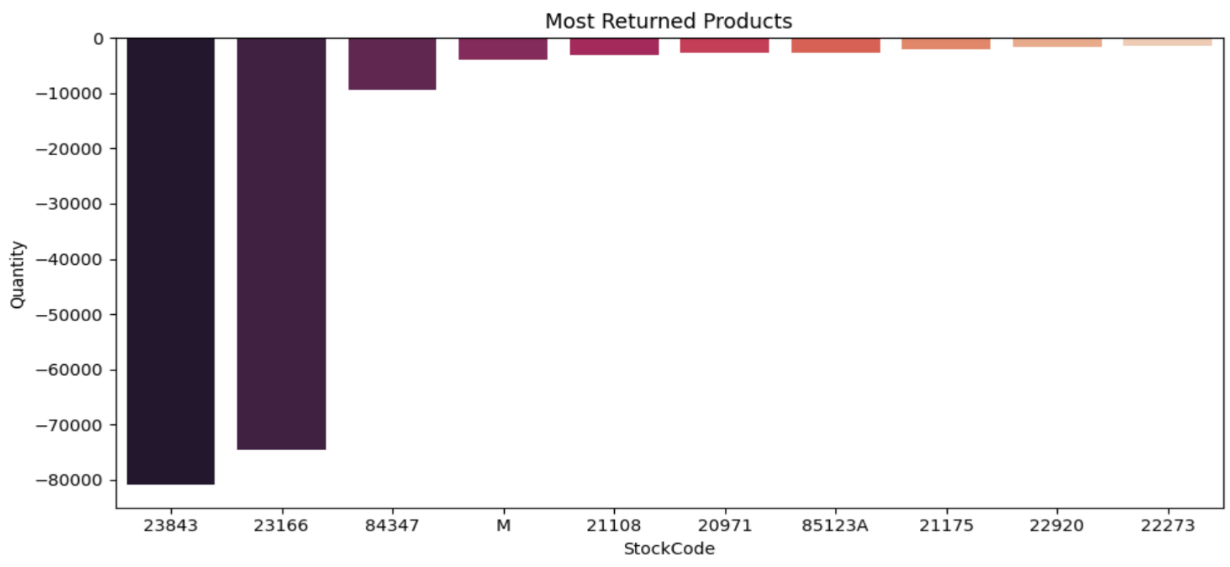
number). The most negative number corresponds to the product with the highest total number of returns.

- **reset_index():** This converts the **'StockCode'** (which became the index during the grouping) back into a regular column and creates a new default integer index.

Plotting top most returned products

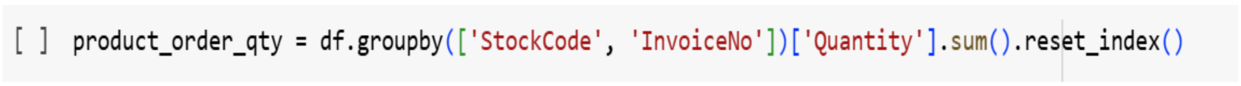


```
[ ]  plt.figure(figsize=(10,5))
     sns.barplot(data=most_returned.head(10), x='StockCode', y='Quantity', palette='rocket')
     plt.title('Most Returned Products')
     plt.xlabel('StockCode')
     plt.ylabel('Quantity')
     plt.tight_layout()
     plt.show()
```



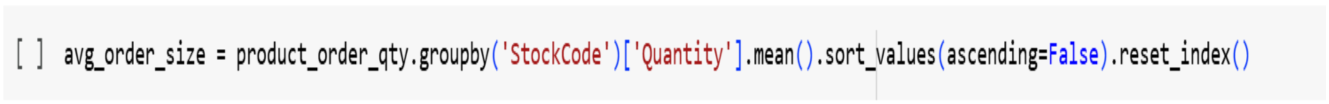
- The bar for StockCode **23843** is the tallest (most negative), showing that this product had the highest number of units returned, with a total of approximately 80,000.
- The second tallest bar, StockCode **23166**, is close behind, with a total of about 75,000 returns.
- The remaining bars show a gradual decrease in the number of units returned, with the smallest bars corresponding to products that had fewer returns.

10. Average Order Size per Product:



```
[ ]  product_order_qty = df.groupby(['StockCode', 'InvoiceNo'])['Quantity'].sum().reset_index()
```

- **df**: This refers to a pandas DataFrame, which is assumed to contain transactional or sales data.
- **groupby(['StockCode', 'InvoiceNo']):** This is a key part of the code. It groups the DataFrame df by a combination of two columns: **'StockCode'** and **'InvoiceNo'**. This means that all rows that have the same product ID (StockCode) and are part of the same transaction/invoice (InvoiceNo) will be grouped together. This is useful for dealing with datasets where an invoice might have multiple line items for the same product.
- **['Quantity'].sum():** After grouping by the combination of 'StockCode' and 'InvoiceNo', this selects the 'Quantity' column for each group and calculates the sum of all the Quantity values within that group. This consolidates the quantities for a single product within a single invoice.
- **reset_index():** After grouping, the **'StockCode'** and **'InvoiceNo'** columns become a multi-level index. reset_index() converts this index back into regular columns, creating a new DataFrame with **'StockCode'**, **'InvoiceNo'**, and the summed **'Quantity'** as columns.
- The result is stored in a new DataFrame called **product_order_qty.**

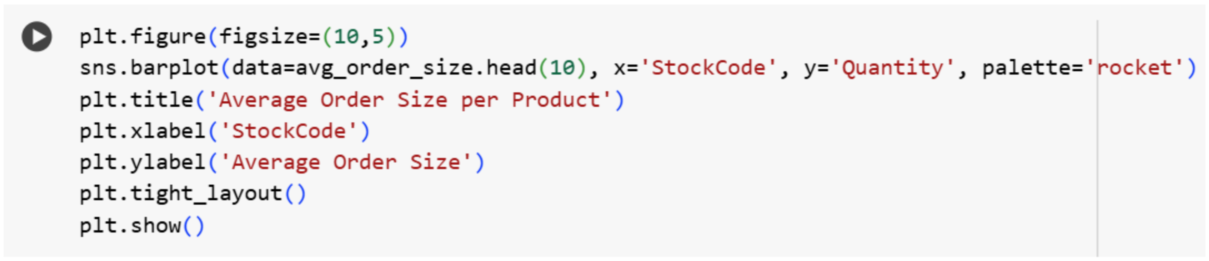


```
[ ]  avg_order_size = product_order_qty.groupby('StockCode')['Quantity'].mean().sort_values(ascending=False).reset_index()
```

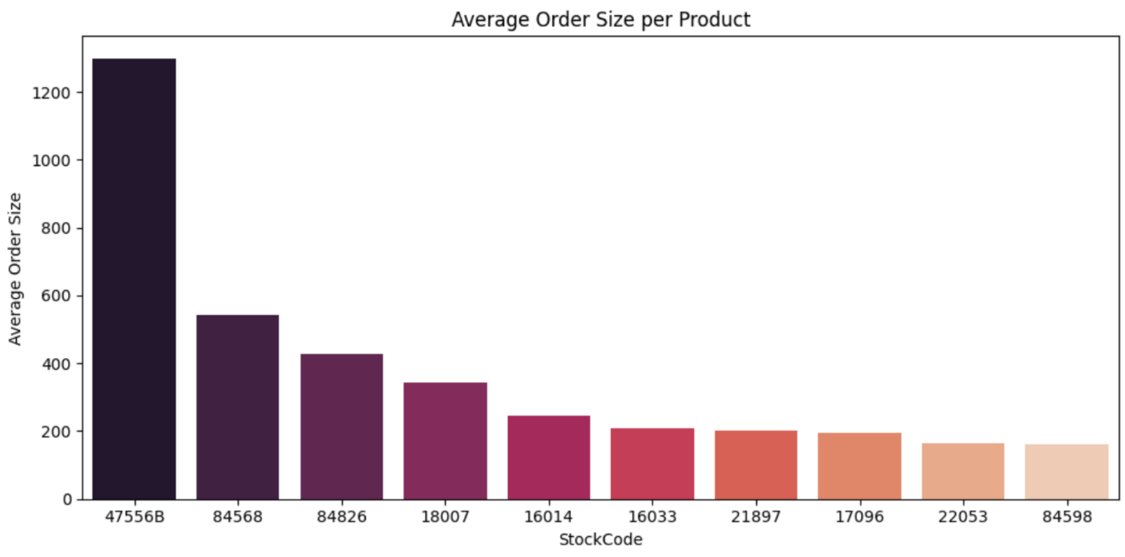
- **product_order_qty:** This is the DataFrame created in the previous step. It contains the total quantity of each product for each invoice (StockCode, InvoiceNo, Quantity).

- **groupby('StockCode'):** This groups the product_order_qty DataFrame by the unique values in the **'StockCode'** column. This means all the quantity totals for a specific product, from all different invoices, will be grouped together.
- **['Quantity'].mean():** After grouping by **'StockCode'**, this selects the **'Quantity'** column and calculates the **mean** (average) for each group. The mean() function here calculates the average number of units of a specific product sold per invoice. This is a measure of the average order size for that particular product.
- **sort_values(ascending=False):** This sorts the resulting average quantities in descending order (from highest to lowest). This allows you to easily identify which products are typically purchased in the largest quantities per order.
- **reset_index():** This converts the **'StockCode'** (which became the index during the grouping and aggregation) back into a regular column, creating a new DataFrame with **'StockCode'** and the average quantity as columns.
- In summary, this single line of code calculates the average order size for each product, sorts the results to find which products are typically bought in the largest quantities, and stores the result in a new DataFrame called **avg_order_size.**

Plotting data

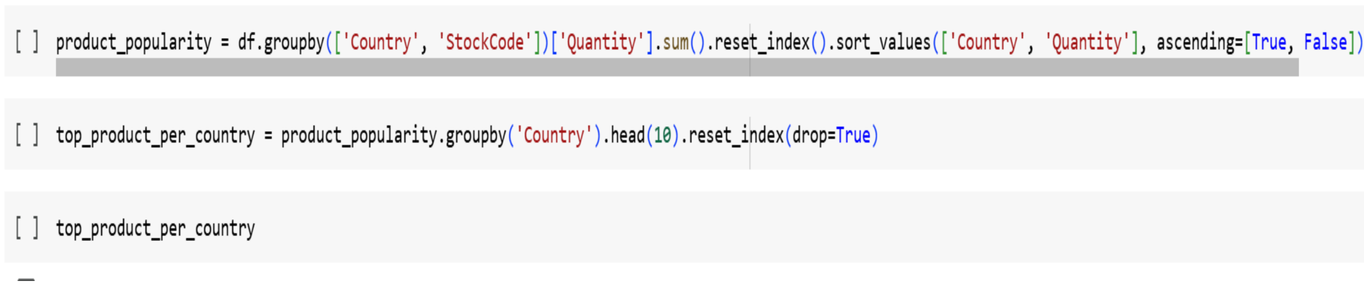


```
plt.figure(figsize=(10,5))
sns.barplot(data=avg_order_size.head(10), x='StockCode', y='Quantity', palette='rocket')
plt.title('Average Order Size per Product')
plt.xlabel('StockCode')
plt.ylabel('Average Order Size')
plt.tight_layout()
plt.show()
```



- The tallest bar on the left, corresponding to StockCode **47556B**, shows the highest average order size, with a value of over 1200 units per order. This product is clearly an outlier, sold in very large quantities when purchased.
- The second tallest bar, StockCode **84568**, has an average order size of around 550 units.
- The remaining bars show a gradual decrease in average order size, with the smallest bar on the far right (StockCode **84598**) having an average order size of just under 200.

11. Product Popularity by Country



```
[ ] product_popularity = df.groupby(['Country', 'StockCode'])['Quantity'].sum().reset_index().sort_values(['Country', 'Quantity'], ascending=[True, False])

[ ] top_product_per_country = product_popularity.groupby('Country').head(10).reset_index(drop=True)

[ ] top_product_per_country
```

- **df.groupby(['Country', 'StockCode']):** This groups the DataFrame df by a combination of two columns: **'Country'** and **'StockCode'**. This creates groups for each unique country-product pair.

- **['Quantity'].sum():** For each country-product group, this calculates the sum of the Quantity values. This gives the total quantity sold for each product in each country.
- **reset_index():** This converts the multi-level index (Country and StockCode) back into regular columns.
- **sort_values(['Country', 'Quantity'], ascending=[True, False]):** This is a crucial and powerful step. It performs a multi-level sort:

- **['Country', 'Quantity']:** The sorting will be done first by **'Country'** and then by **'Quantity'**.

- **ascending=[True, False]:** This specifies the sorting order for each of the columns. True for **'Country'**: It will sort the countries in alphabetical order (e.g., Australia, Belgium, France, etc.). False for **'Quantity'**: Within each country, it will sort the total quantity from highest to lowest.

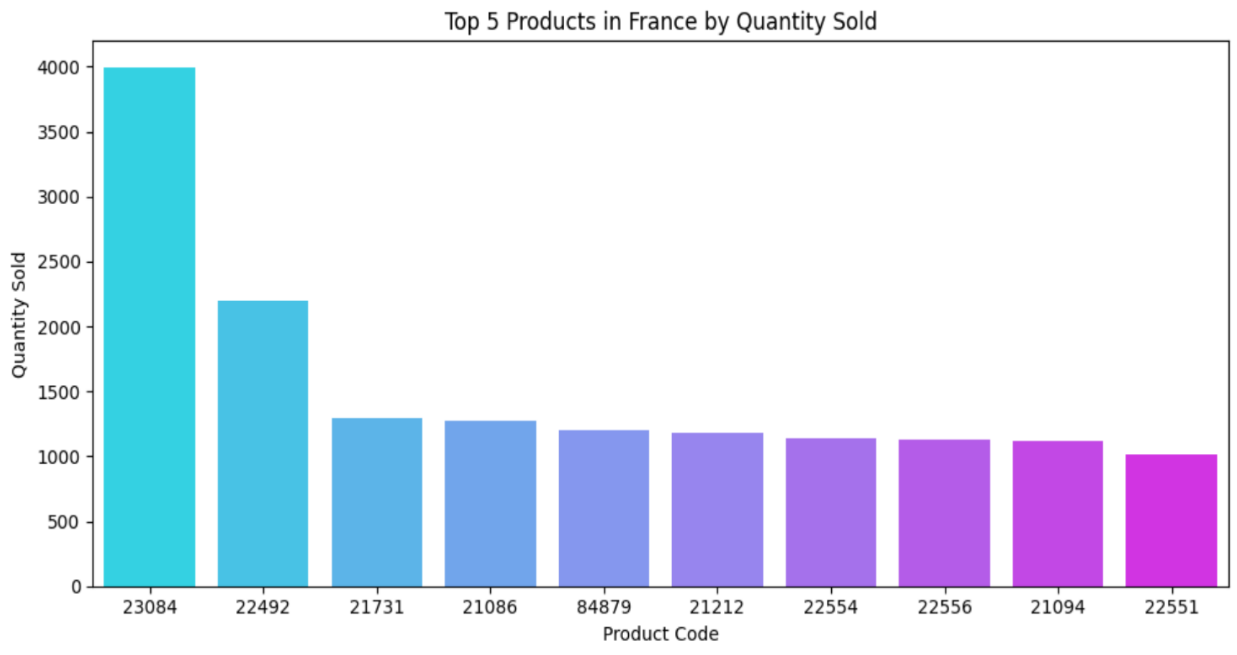
- This first line of code creates a DataFrame that lists the total quantity sold for every product in every country, and then sorts this list so that the products for each country are ranked from the best-selling to the worst-selling. The result is stored in the **product_popularity** DataFrame.

- **product_popularity.groupby('Country'):** This regroups the product_popularity DataFrame (which is already sorted) by **'Country'**.
- **head(10):** This is applied to each group. Because the groups are already sorted by quantity in descending order, **head(10)** selects the top 10 rows (the top 10 best-selling products) for each country.
- **reset_index(drop=True):** This resets the index, but drop=True prevents the old index from being added as a column. This results in a clean DataFrame with a default integer index.
- This line takes the pre-sorted **product_popularity** DataFrame and extracts the top 10 products for each country, storing the result in a new DataFrame called **top_product_per_country**.

Plotting popular products in France:

```
[ ]  top_france = top_product_per_country[top_product_per_country['Country'] == 'France']
```

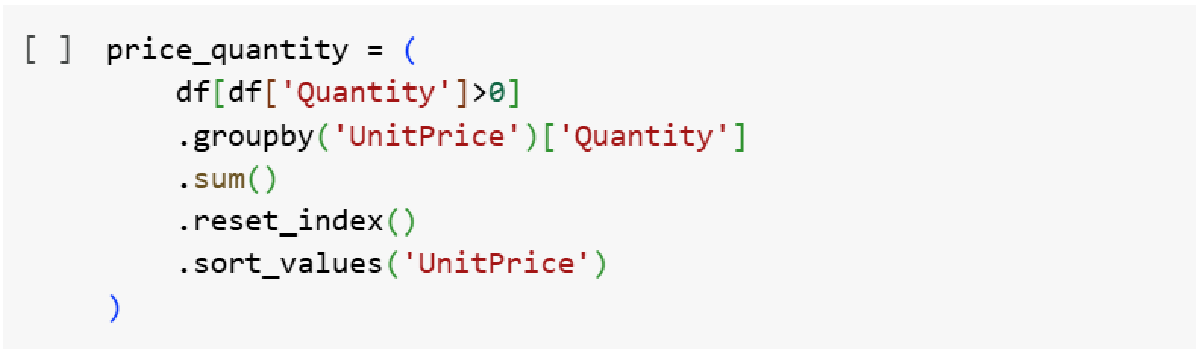
```
[ ]  plt.figure(figsize=(10, 5))
     sns.barplot(data=top_france, x='StockCode', y='Quantity', palette='cool')
     plt.title(f'Top 5 Products in France by Quantity Sold')
     plt.xlabel('Product Code')
     plt.ylabel('Quantity Sold')
     plt.tight_layout()
     plt.show()
```



- The tallest bar on the left, corresponding to Product Code **23084**, shows the highest quantity sold in France, at almost 4000 units.
- The second tallest bar, Product Code **22492**, has a quantity sold of just over 2000 units.
- The remaining bars show a gradual decrease in quantities sold, with the shortest bar on the far right (Product Code **22551**) having a quantity sold of approximately 1000 units.

It clearly identifies Product Code **23084** as the most popular product in France by a significant margin.

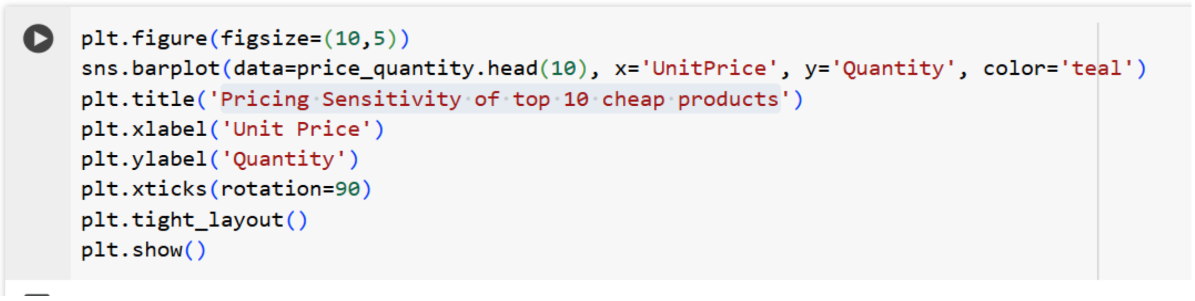
12. Price Sensitivity:



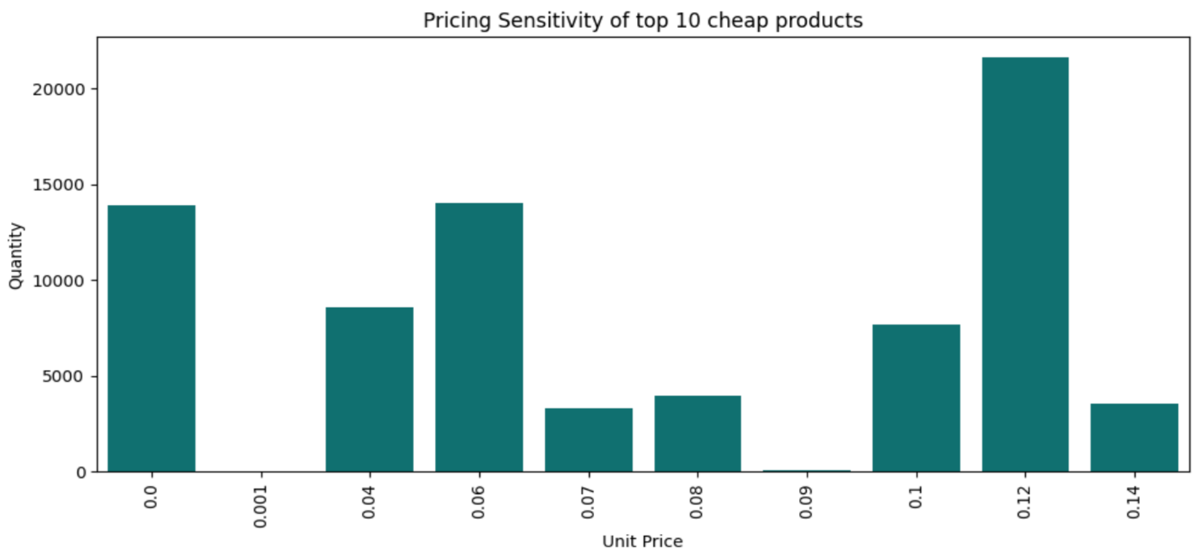
```
[ ]  price_quantity = (
         df[df['Quantity']>0]
         .groupby('UnitPrice')['Quantity']
         .sum()
         .reset_index()
         .sort_values('UnitPrice')
     )
```

- **df['Quantity']>0:** This is a Boolean condition that checks every row in the df DataFrame to see if the value in the 'Quantity' column is greater than zero. This effectively filters out all returns and cancelled orders (where quantity is typically negative or zero).
- **groupby('UnitPrice')**: After filtering for only sales, this groups the data by the unique values in the **'UnitPrice'** column. This means all sales that occurred at the same price will be grouped together.
- **['Quantity'].sum()**: For each group (each unique price), this selects the **'Quantity'** column and calculates the sum of all quantities sold at that specific price.
- **reset_index()**: After the groupby and sum operations, **'UnitPrice'** becomes the index. This command converts **'UnitPrice'** back into a regular column, creating a new DataFrame with **'UnitPrice'** and the summed **'Quantity'** as columns.
- **sort_values('UnitPrice')**: This sorts the resulting DataFrame by the **'UnitPrice'** column in ascending order. This arranges the data from the lowest price to the highest price.
- The entire operation is assigned to the variable **price_quantity**.

Plotting Pricing Sensitivity of top 10 cheap & costly products

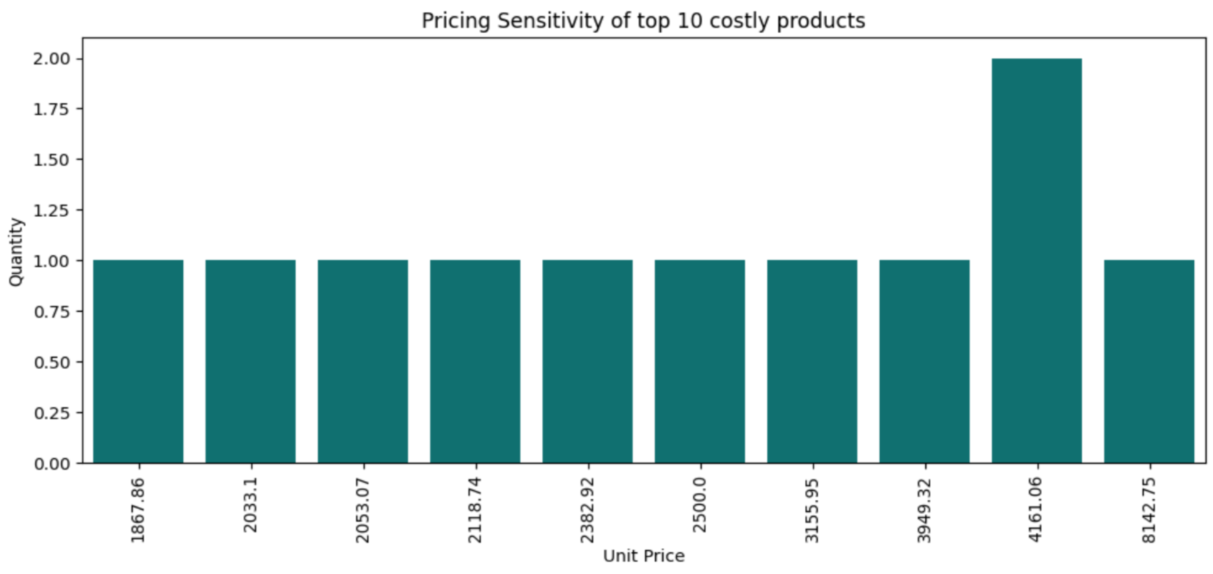


```
plt.figure(figsize=(10,5))
sns.barplot(data=price_quantity.head(10), x='UnitPrice', y='Quantity', color='teal')
plt.title('Pricing Sensitivity of top 10 cheap products')
plt.xlabel('Unit Price')
plt.ylabel('Quantity')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```



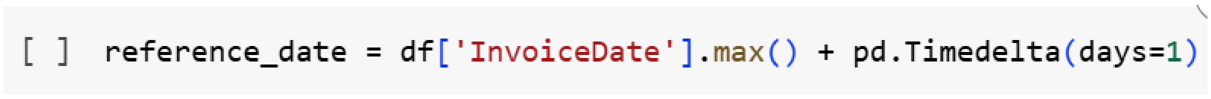
- The tallest bar is at a Unit Price of **0.12**, showing a total quantity sold of over 20,000 units. This suggests that this price point is highly effective for driving sales volume.
- The second and third tallest bars are at Unit Price **0.06** and **0.0**, both with quantities sold around 14,000 units. The Unit Price of 0.0 might represent promotional items, free samples, or data entry errors.
- The bar at Unit Price **0.04** is also significant, with a quantity sold of over 8,000 units.
- Other price points like **0.07, 0.08, and 0.09** show much lower quantities sold, indicating lower demand at those prices. The quantity sold at **0.09** is particularly low, barely visible on the chart.
- It reveals that the most effective price point for maximizing sales volume is **0.12**, followed by **0.06** and **0.0**.

```python
plt.figure(figsize=(10,5))
sns.barplot(data=price_quantity.tail(10), x='UnitPrice', y='Quantity', color='teal')
plt.title('Pricing Sensitivity of top 10 costly products')
plt.xlabel('Unit Price')
plt.ylabel('Quantity')
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```



- For most of the price points (**1867.86, 2033.1, 2053.07**, etc.), the Quantity sold is exactly **1**. This suggests that these are very high-value, low-volume items, and each purchase is typically for a single unit.
- The most notable bar is at a Unit Price of **4161.06**, where the Quantity sold is **2**. This indicates that on at least one occasion, two units of this very expensive item were sold together.
- The bar at Unit Price **8142.75** shows a quantity of **1**, which is the most expensive product in this top 10 list, sold as a single unit.
- It shows that for these "costly" items, sales volume is extremely low, with most purchases being for a single unit. The exception is the product priced at **4161.06**, which sold two units.

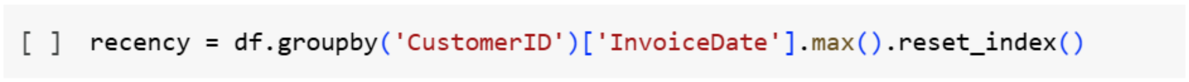
13. Recency Analysis:



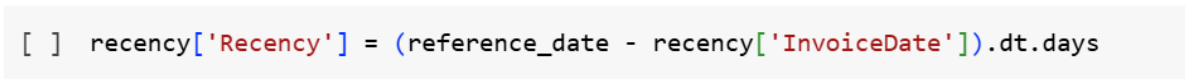
```
[ ]  reference_date = df['InvoiceDate'].max() + pd.Timedelta(days=1)
```

This single line of Python code is a common step in preparing data for an RFM (Recency, Frequency, Monetary) analysis.

- **df['InvoiceDate']**: This selects the **'InvoiceDate'** column from the pandas DataFrame **df**. This column is assumed to contain the date and time of each transaction.
- **max()**: This method finds the most recent date in the **'InvoiceDate'** column. It identifies the date of the last transaction in the entire dataset.
- **pd.Timedelta(days=1)**: This creates a pandas Timedelta object, which represents a duration of one day.
- The concat operator **(+)** adds the one-day Timedelta to the maximum date found in the dataset. This creates a date that is one day after the last transaction date.



```
[ ]  recency = df.groupby('CustomerID')['InvoiceDate'].max().reset_index()
```

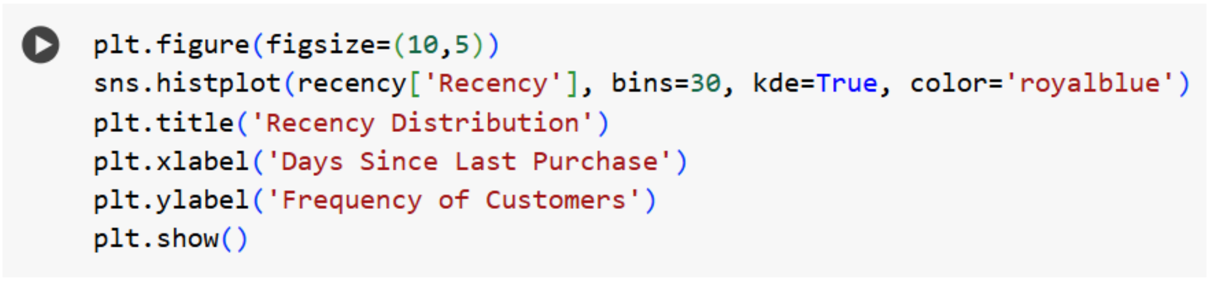


```
[ ]  recency['Recency'] = (reference_date - recency['InvoiceDate']).dt.days
```

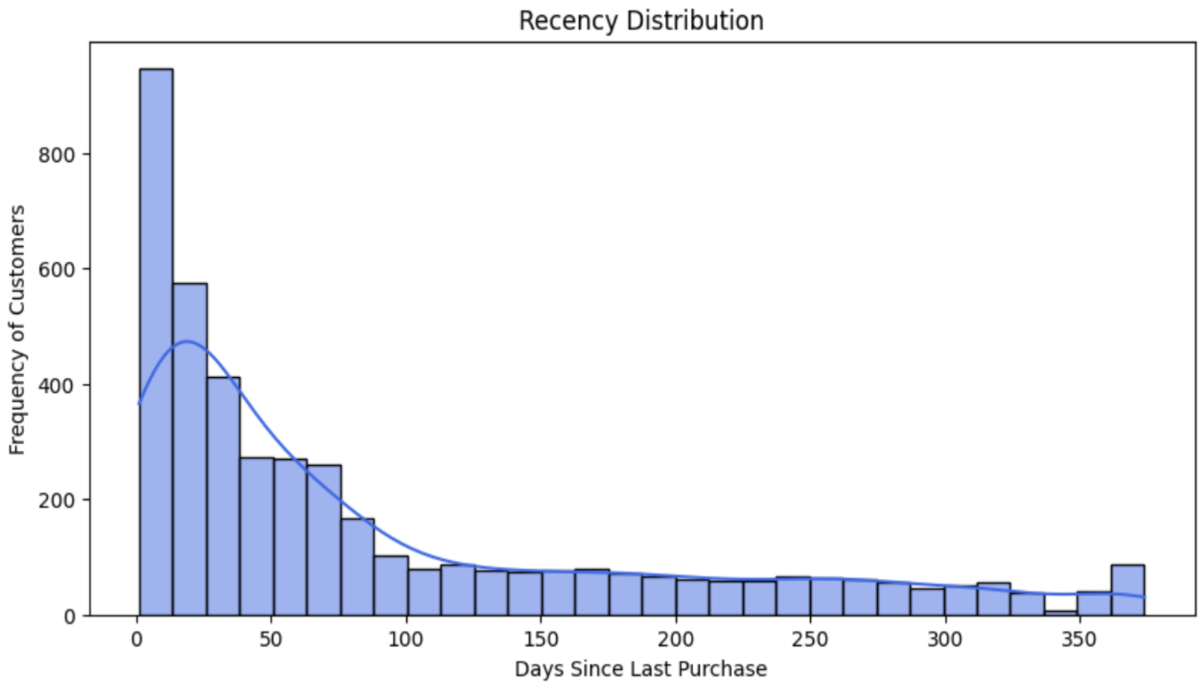
- **df.groupby('CustomerID'):** This groups the main DataFrame df by each unique **CustomerID**.
- **['InvoiceDate'].max():** For each customer group, this selects the **'InvoiceDate'** column and finds the **maximum** date. The maximum date for each customer is their most recent purchase date.
- **reset_index():** This converts the **CustomerID** (which became the index during the grouping) back into a regular column.
- **recency**: This assigns the resulting DataFrame, which now contains a **CustomerID** and their most recent **InvoiceDate**, to a new variable called **recency**.

- **recency['Recency']** = This creates a new column in the recency DataFrame called **'Recency'**.
- **(reference_date - recency['InvoiceDate']):** This performs a subtraction operation. It subtracts the customer's most recent InvoiceDate (from the recency DataFrame) from the **reference_date** that was calculated in a previous step. The **reference_date** is one day after the last transaction in the entire dataset. This subtraction results in a Timedelta object for each customer, representing the number of days that have passed since their last purchase.
- **dt.days:** This is a pandas accessor that extracts the number of full days from the Timedelta object, converting it into a simple integer.

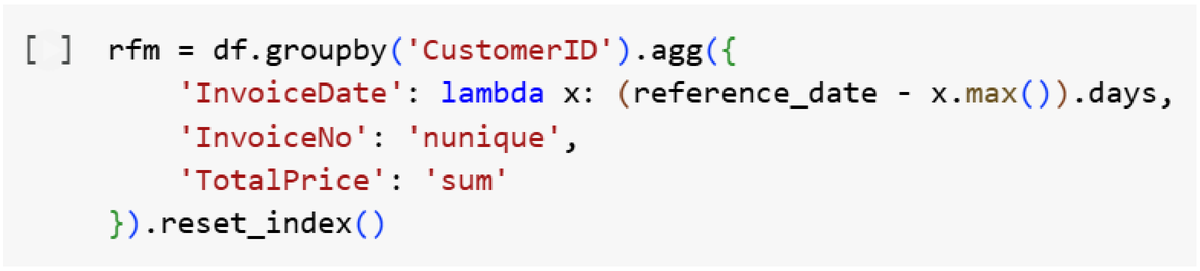
Plotting Recency Distribution:



```
plt.figure(figsize=(10,5))
sns.histplot(recency['Recency'], bins=30, kde=True, color='royalblue')
plt.title('Recency Distribution')
plt.xlabel('Days Since Last Purchase')
plt.ylabel('Frequency of Customers')
plt.show()
```



- The tallest bar is on the far left, representing customers with a Recency of approximately **0-25 days**. The frequency of these customers is very high, close to **900**. This means a large number of customers have made a purchase very recently.
- The frequency of customers decreases sharply as the number of days since their last purchase increases. The bars become progressively shorter, forming a long tail to the right.
- There is a small bump in the frequency of customers in the last bin (**around 350-375 days**). This could be customers who made a purchase at the very beginning of the dataset's time frame but have not returned since.
- **Overlaid KDE Plot (Blue Curve):** The smooth blue curve overlaid on the histogram is a kernel density estimate. It provides a visual representation of the underlying probability density function of the data, helping to smooth out the noise from the bins and better illustrate the overall shape of the distribution. The curve clearly shows a sharp peak at the beginning and a long, gradually declining tail, reinforcing the pattern seen in the bars.



```
[ ] rfm = df.groupby('CustomerID').agg({
        'InvoiceDate': lambda x: (reference_date - x.max()).days,
        'InvoiceNo': 'nunique',
        'TotalPrice': 'sum'
    }).reset_index()
```

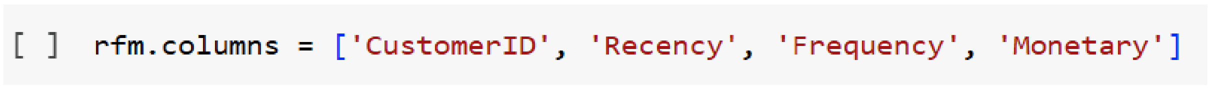
- **df.groupby('CustomerID')**: This is the starting point, where the entire DataFrame df is grouped by each unique **CustomerID**. All subsequent operations will be performed for each individual customer.
- **agg({...})**: The agg() method is used to apply different aggregation functions to different columns simultaneously on

the grouped data. The dictionary inside agg specifies which column to apply which function to.

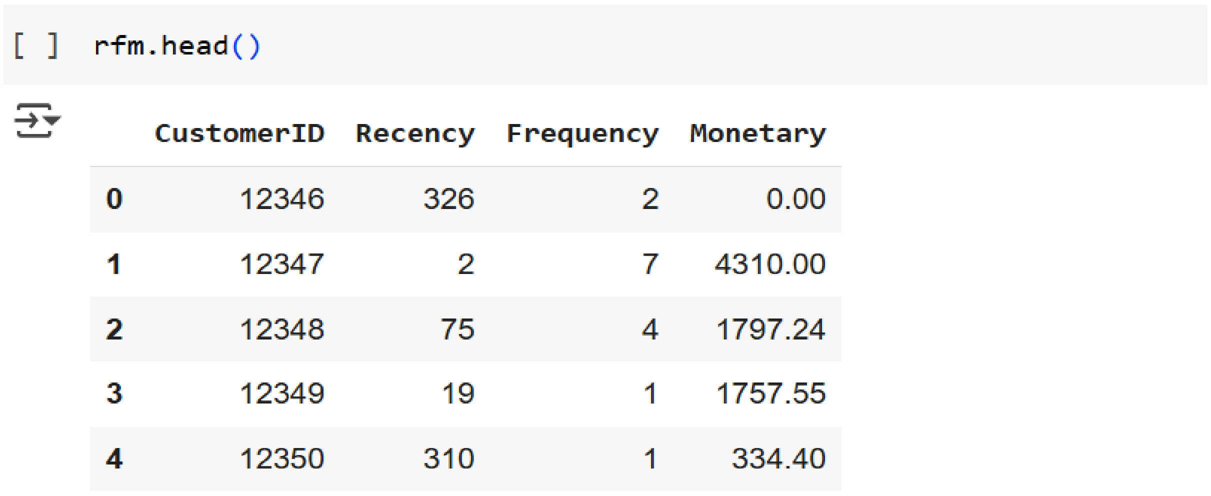
- 'InvoiceDate': lambda x: (reference_date- x.max()).days: This part calculates the Recency score.
    - 'InvoiceDate': The aggregation is performed on the 'InvoiceDate' column.
    - lambda x: ...: This is an anonymous function (lambda function) that will be applied to the 'InvoiceDate' Series for each customer group.
    - x.max(): This finds the most recent purchase date for the current customer.
    - (reference_date- x.max()): This subtracts the customer's last purchase date from the reference_date (which was likely defined in a previous step as one day after the last transaction in the dataset). This results in a Timedelta object.
    - days: This extracts the number of days from the Timedelta object. The result is the recency score.
- 'InvoiceNo': 'nunique': This part calculates the Frequency score.
    - 'InvoiceNo': The aggregation is performed on the 'InvoiceNo' column.
    - 'nunique': This is a built-in pandas string shortcut for the nunique() function, which counts the number of unique invoices for each customer. This represents the number of separate purchases a customer has made.
- 'TotalPrice': 'sum': This part calculates the Monetary score.
    - 'TotalPrice': The aggregation is performed on the 'TotalPrice' column.

    - **'sum'**: This is a built-in pandas string shortcut for the sum() function, which adds up all the **TotalPrice** values for each customer. This represents the total revenue generated by that customer, a measure of their lifetime value (CLV).

- **reset_index()**: After the aggregation, **'CustomerID'** becomes the index. This convert **'CustomerID'** back into a regular column, resulting in a clean DataFrame.
- **rfm** :The final DataFrame, containing CustomerID, the calculated recency, frequency, and monetary scores, is assigned to the variable rfm.



```
[ ]  rfm.columns = ['CustomerID', 'Recency', 'Frequency', 'Monetary']
```

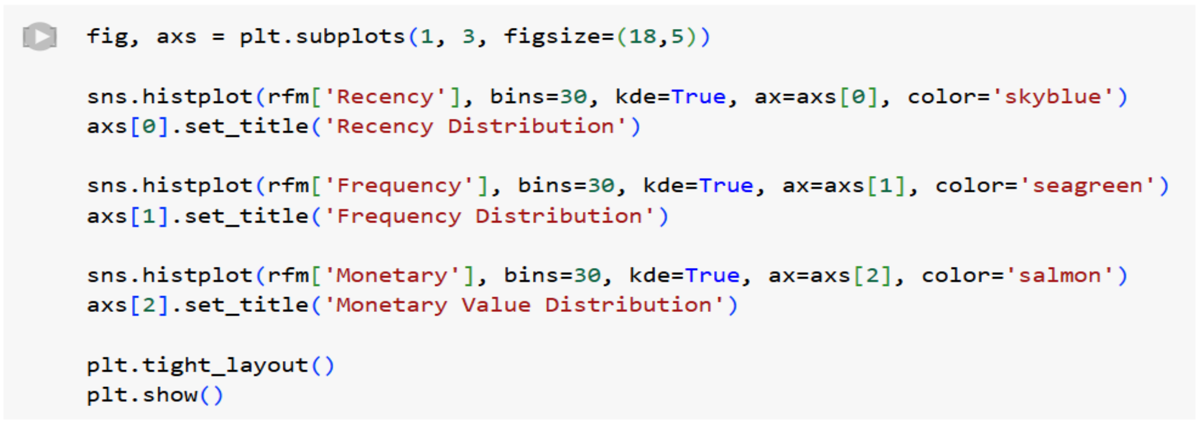


```
[ ]  rfm.head()
```

	CustomerID	Recency	Frequency	Monetary
0	12346	326	2	0.00
1	12347	2	7	4310.00
2	12348	75	4	1797.24
3	12349	19	1	1757.55
4	12350	310	1	334.40

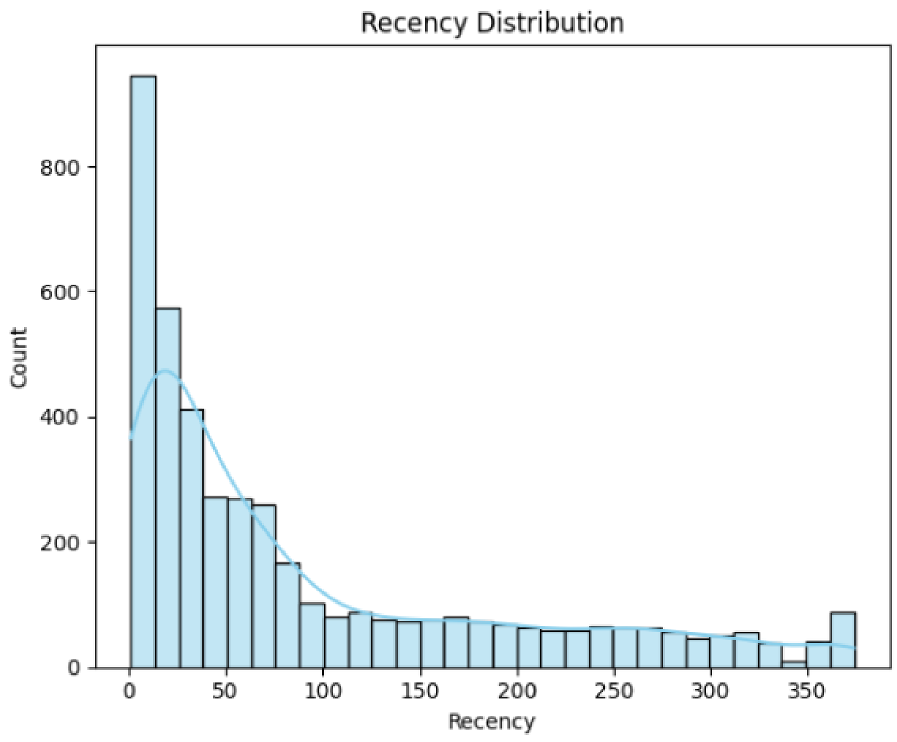
Renaming the columns as **CustomerID, Recency, Frequency, Monetary.**

Plotting Recency, Frequency, Monetary distributions:

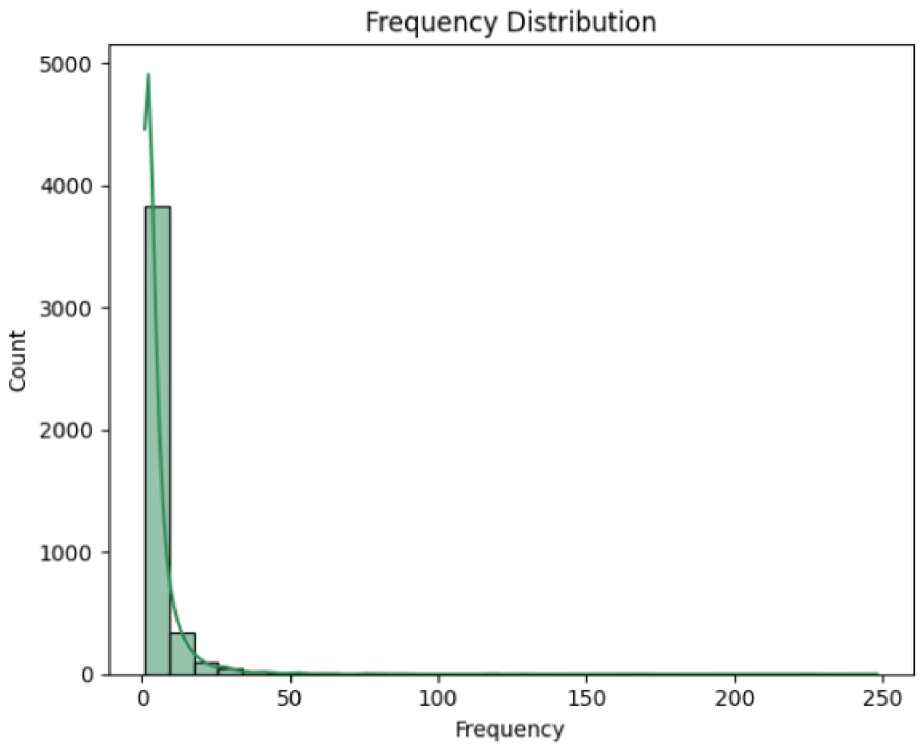


```
fig, axs = plt.subplots(1, 3, figsize=(18,5))

sns.histplot(rfm['Recency'], bins=30, kde=True, ax=axs[0], color='skyblue')
axs[0].set_title('Recency Distribution')

sns.histplot(rfm['Frequency'], bins=30, kde=True, ax=axs[1], color='seagreen')
axs[1].set_title('Frequency Distribution')

sns.histplot(rfm['Monetary'], bins=30, kde=True, ax=axs[2], color='salmon')
axs[2].set_title('Monetary Value Distribution')

plt.tight_layout()
plt.show()
```

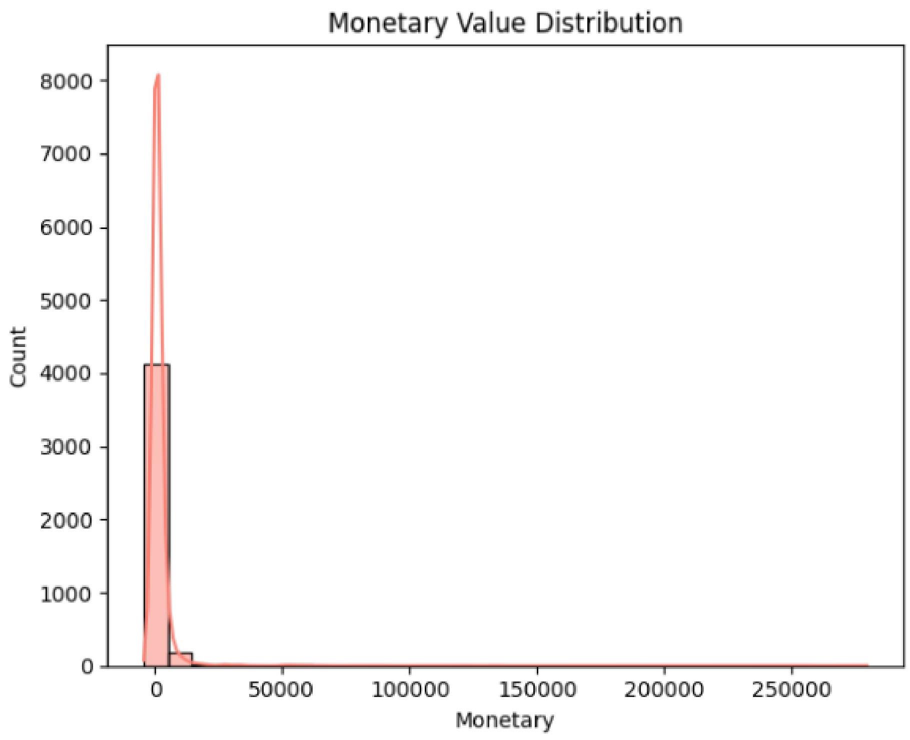
Recency Distribution



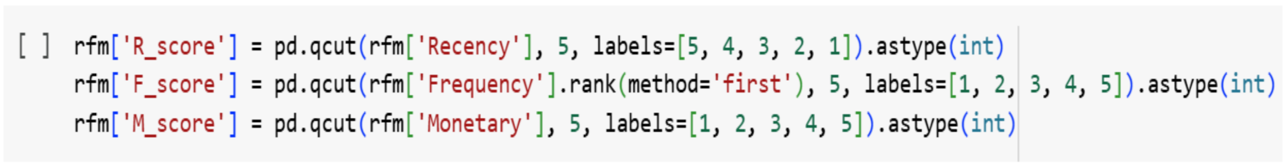
Frequency Distribution



Monetary Value Distribution



Customer Segmentation



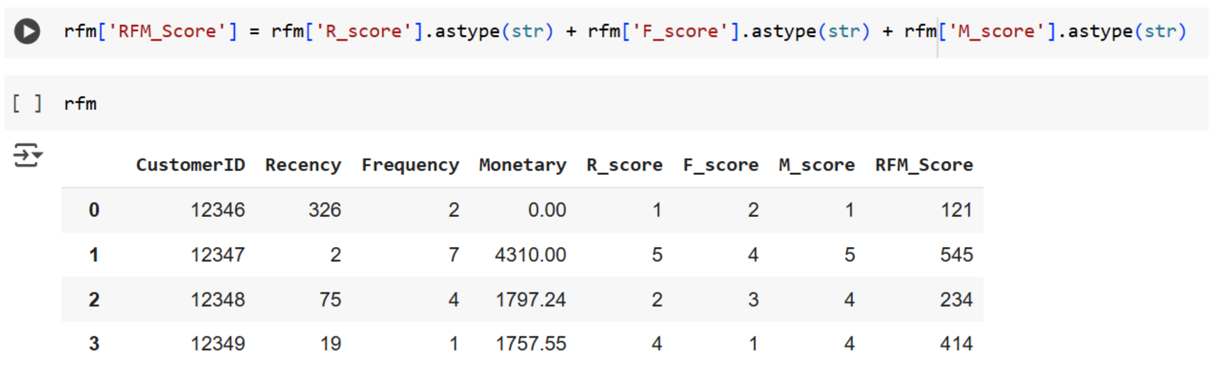
```
[ ]  rfm['R_score'] = pd.qcut(rfm['Recency'], 5, labels=[5, 4, 3, 2, 1]).astype(int)
     rfm['F_score'] = pd.qcut(rfm['Frequency'].rank(method='first'), 5, labels=[1, 2, 3, 4, 5]).astype(int)
     rfm['M_score'] = pd.qcut(rfm['Monetary'], 5, labels=[1, 2, 3, 4, 5]).astype(int)
```

- **pd.qcut(...):** This is a pandas function that performs a quantile-based discretization. It divides the data into q equal-sized bins based on the value distribution.
- **rfm['Recency']:** The data being discretized is the Recency score column from the rfm DataFrame.
- **5**: This specifies that the data should be divided into 5 bins (quintiles).
- **labels = [5, 4, 3, 2, 1]:** This assigns a label to each bin. The labels are applied from the smallest value to the largest. Since lower Recency values (fewer days since last purchase) are considered better, the labels are assigned in reverse order. The customers in the top 20% (the most recent buyers) get a score of 5, while those in the bottom 20% (least recent) get a score of 1.
- **astype(int):** This converts the resulting score (which might be a categorical type) into an integer.
- **rfm['R_score'] = :** The resulting integer scores are stored in a new column named 'R_score' in the rfm DataFrame that contains the Recency score.

- **rfm['Frequency'].rank(method='first'):** This part is important. Quantile cutting with **qcut** can sometimes have issues with tied values. By first ranking the Frequency values, the code ensures that each unique value is given a unique rank before being divided into quantiles, which helps handle ties correctly.
- **pd.qcut(..., 5, labels=[1, 2, 3, 4, 5])**: Similar to the Recency score, the ranked frequencies are divided into 5 bins. However, for

Frequency, a higher value (more purchases) is better. Therefore, the labels are assigned in ascending order: the customers with the lowest frequency get a score of 1, and the most frequent buyers get a score of 5.

- **astype(int):** Converts the scores to integers.
- **rfm['F_score'] = :** Stores the scores in a new column 'F_score' having the Frequency Score.
- **pd.qcut(rfm['Monetary'], 5, labels=[1, 2, 3, 4, 5]):** This is identical in logic to the Frequency score. Higher monetary value is better. The Monetary values are divided into 5 bins, and the labels are assigned in ascending order.
- **astype(int):** Converts the scores to integers.
- **rfm['M_score'] =:** Stores the scores in a new column 'M_score' i.e. the Monetary value.

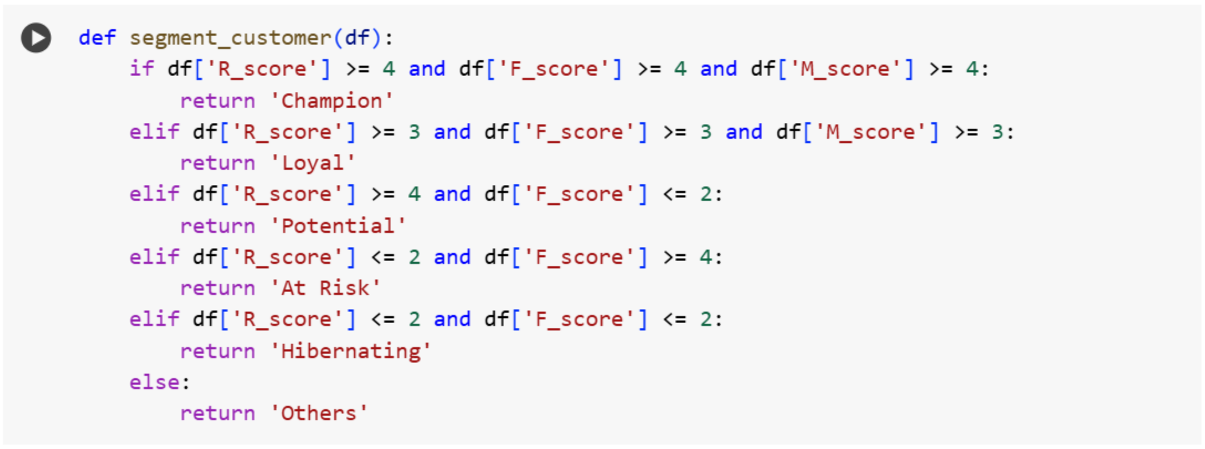


```
rfm['RFM_Score'] = rfm['R_score'].astype(str) + rfm['F_score'].astype(str) + rfm['M_score'].astype(str)
```

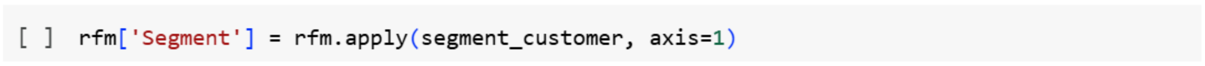
```
rfm
```

	CustomerID	Recency	Frequency	Monetary	R_score	F_score	M_score	RFM_Score
0	12346	326	2	0.00	1	2	1	121
1	12347	2	7	4310.00	5	4	5	545
2	12348	75	4	1797.24	2	3	4	234
3	12349	19	1	1757.55	4	1	4	414

- This creates a new column in the **rfm** DataFrame called **'RFM_Score'**.
- **rfm['R_score'].astype(str)**: This selects the **'R_score'** column and converts the numerical scores (e.g., 5, 4, 3) into strings ('5', '4', '3').
- **The + operator**, when used with strings, performs string concatenation.



```python
def segment_customer(df):
    if df['R_score'] >= 4 and df['F_score'] >= 4 and df['M_score'] >= 4:
        return 'Champion'
    elif df['R_score'] >= 3 and df['F_score'] >= 3 and df['M_score'] >= 3:
        return 'Loyal'
    elif df['R_score'] >= 4 and df['F_score'] <= 2:
        return 'Potential'
    elif df['R_score'] <= 2 and df['F_score'] >= 4:
        return 'At Risk'
    elif df['R_score'] <= 2 and df['F_score'] <= 2:
        return 'Hibernating'
    else:
        return 'Others'
```

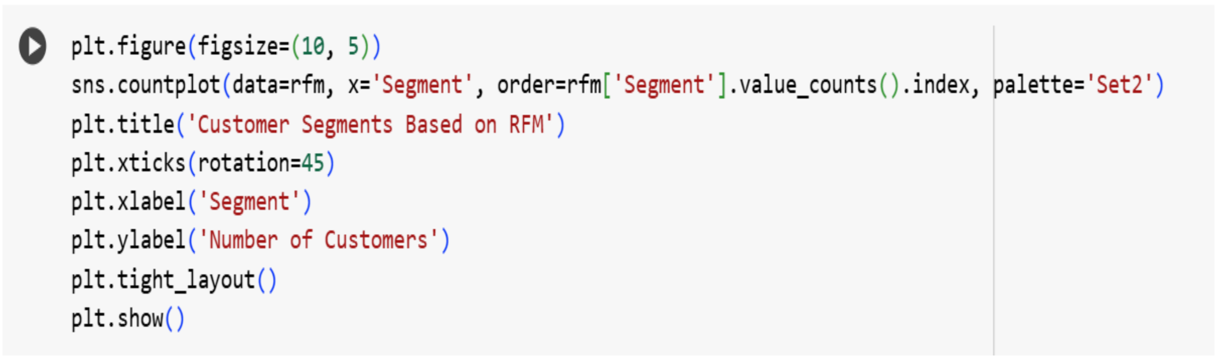


```python
[ ] rfm['Segment'] = rfm.apply(segment_customer, axis=1)
```

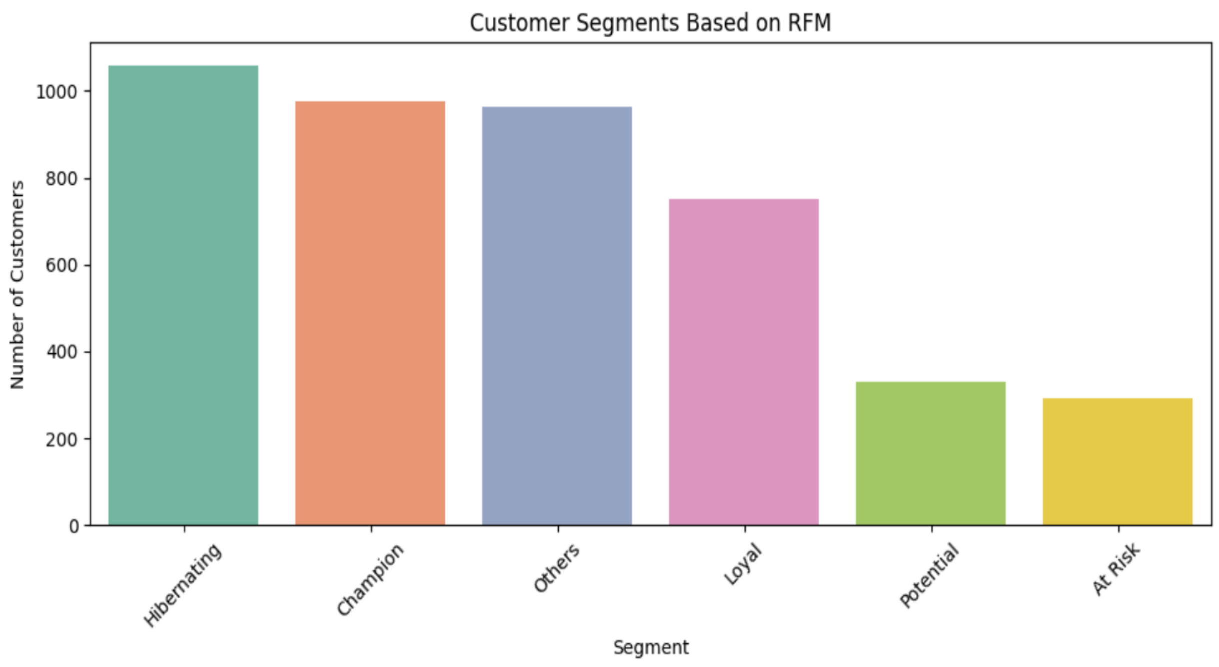
- **def segment_customer(df):** This defines a function named **segment_customer** that takes a single row of a DataFrame as input (represented here as df).
- The function uses a series of if and elif (else if) statements to check the values of the **R_score, F_score, and M_score** for a given customer.
- **if df['R_score'] >= 4 and df['F_score'] >= 4 and df['M_score'] >= 4:**: The first condition checks for "**Champion**" customers. These are the most valuable customers who have purchased recently, frequently, and spent a lot of money.
- **elif df['R_score'] >= 3 and df['F_score'] >= 3 and df['M_score'] >= 3:**: The second condition checks for "**Loyal**" customers, who are still valuable but may not score as high as champions.
- **elif df['R_score'] >= 4 and df['F_score'] <= 2:**: This identifies "**Potential**" customers. They have purchased recently (high R score) but not frequently (low F score). They represent a good opportunity for up-selling or cross-selling.
- **elif df['R_score'] <= 2 and df['F_score'] >= 4:**: This identifies "**At Risk"** customers. They have purchased frequently in the past (high F score) but not recently (low R score). They are at risk of churning and may need targeted retention campaigns.
- **elif df['R_score'] <= 2 and df['F_score'] <= 2:**: This identifies "**Hibernating**" customers. They have not purchased recently or frequently and are likely inactive.

- **else: return 'Others'**: If a customer's scores don't fit into any of the predefined segments, they are classified as **'Others'**.
- **rfm.apply(segment_customer, axis=1):** This is a powerful pandas method that applies the **segment_customer** function to every row (axis=1) of the **rfm** DataFrame. A new column named **'Segment'** is created in the **rfm** DataFrame, and the output from the apply function is populated into this new column.

Plotting Segmented customers:



```
plt.figure(figsize=(10, 5))
sns.countplot(data=rfm, x='Segment', order=rfm['Segment'].value_counts().index, palette='Set2')
plt.title('Customer Segments Based on RFM')
plt.xticks(rotation=45)
plt.xlabel('Segment')
plt.ylabel('Number of Customers')
plt.tight_layout()
plt.show()
```



- The tallest bar is for the "**Hibernating**" segment, with a count of over **1000** customers. This indicates that a significant portion of the customer base has not purchased recently or frequently and is currently inactive.

- The second tallest bar is for "Champion" customers, with a count of just under 1000. These are the most valuable and active customers.
- The "Others" and "Loyal" segments have a similar number of customers, both with a count between 800 and 1000.
- The "Potential" and "At Risk" segments have the fewest customers, with counts below 400 and 300, respectively.

This bar chart provides a high-level overview of the customer base, segmented according to their RFM scores. It shows that the largest single group of customers is "Hibernating," followed closely by "Champions," "Others," and "Loyals." This information is extremely valuable for business strategy, as it allows for targeted actions:

- Hibernating: Requires a re-engagement strategy to bring them back.
- Champions: Should be rewarded and retained with exclusive offers.
- Loyal: Need to be nurtured to maintain their loyalty.
- Potential: Can be targeted with promotions to increase their purchase frequency.
- At Risk: Should be the focus of retention campaigns to prevent them from churning. This chart is the final outcome of the RFM analysis, presenting actionable insights in a clear, visual format.

Algorithm for the Project

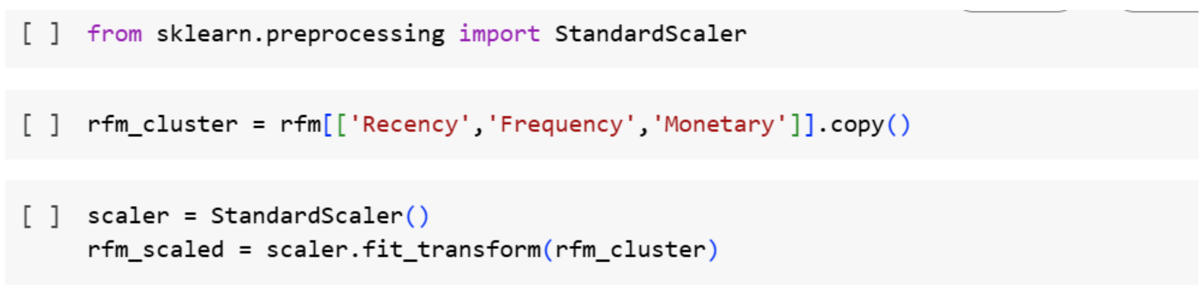
We chose **Cluster Analysis** for customer segmentation in this project because it allows us to uncover **natural groupings** in customer behaviour without needing predefined labels, making it ideal for understanding diverse shopping patterns in the dataset.

The Cluster analysis fits here perfectly because of:

1. **Unsupervised Nature**
   - We don't have "predefined" customer groups in the data.
   - Clustering lets the data itself determine the structure of segments based on purchase behaviour.
2. **Data-Driven Segmentation**
   - Customers can be grouped using key metrics like:
     - **Recency** – How recently they purchased.
     - **Frequency** – How often they purchase.
     - **Monetary Value** – How much they spend.
   - This aligns perfectly with our **RFM model** foundation.
3. **Personalization Potential**
   - Once clusters are identified, each group can be targeted differently:
     - **High-value, frequent buyers** - Loyalty rewards.
     - **At-risk customers** - Win-back campaigns.
     - **New customers** - Welcome offers.
4. **Handles Large Volumes of Data**
   - The Online Retail dataset contains **hundreds of thousands of transactions**.
   - Clustering algorithms like **K-Means** scale well for large datasets and numeric features.
5. **Actionable Insights**
   - Clustering reveals **hidden behavioural patterns** (e.g., customers who buy small quantities frequently vs. bulk seasonal buyers).

    - Enables data-driven decision-making for marketing, stock planning, and customer retention.

Standardising data for cluster analysis:



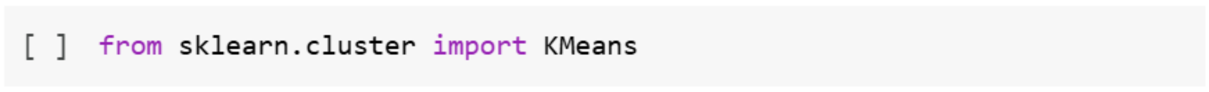
```
[ ]  from sklearn.preprocessing import StandardScaler
```

```
[ ]  rfm_cluster = rfm[['Recency','Frequency','Monetary']].copy()
```

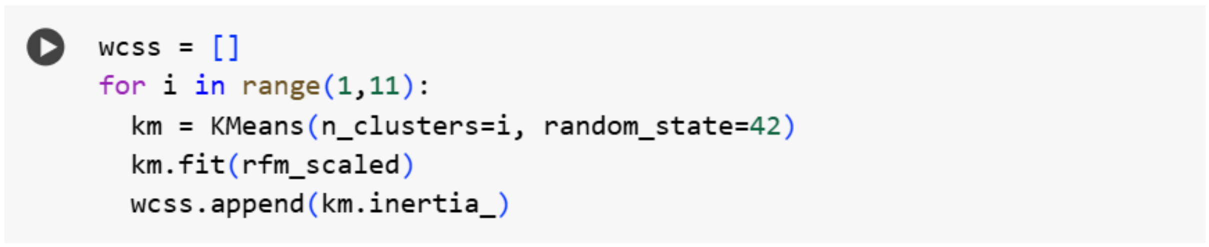
```
[ ]  scaler = StandardScaler()
     rfm_scaled = scaler.fit_transform(rfm_cluster)
```

- StandardScaler from scikit-learn is used to **standardize features**.
- Standardization means **subtracting the mean and dividing by the standard deviation** for each feature so that: Mean = 0, Standard deviation = 1.
- This ensures that **Recency, Frequency, and Monetary** values are on the same scale before clustering.
- Extracts only the **three RFM metrics** from the rfm DataFrame.
- **copy()** makes a separate copy so the original rfm data isn't affected.
- These are the input features for clustering.
- Create an instance of StandardScaler.
- **fit_transform(): fit** → learns the mean and standard deviation for each feature & **transform** → applies the scaling to convert the data into standardized values.
- rfm_scaled will now be a **NumPy array** where each RFM value is normalized — this is important because clustering algorithms like **K-Means** are sensitive to differences in scale.

Implementation of **KMeans** clustering with **Elbow Method:**

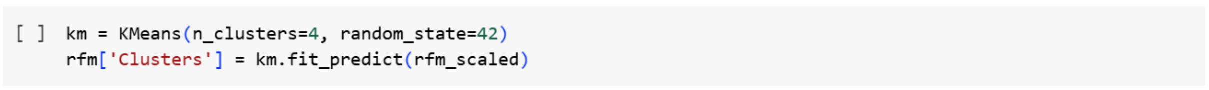


```
[ ] from sklearn.cluster import KMeans
```

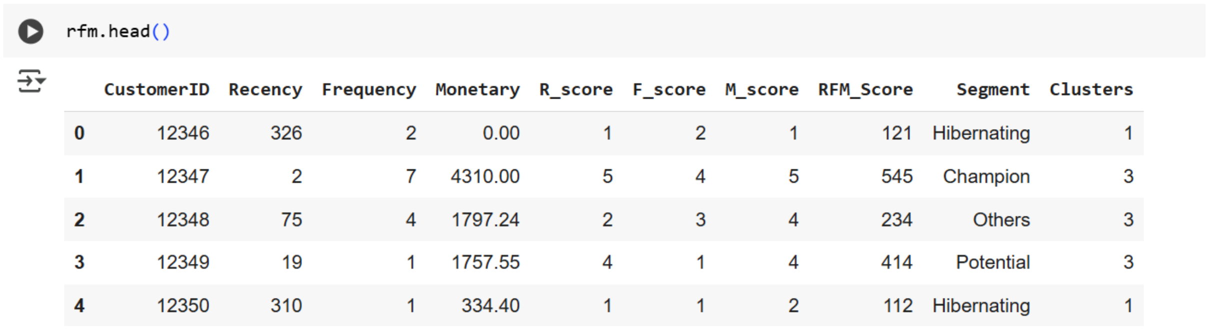


```
wcss = []
for i in range(1,11):
  km = KMeans(n_clusters=i, random_state=42)
  km.fit(rfm_scaled)
  wcss.append(km.inertia_)
```

- Imports the KMeans clustering algorithm from scikit-learn.
- **WCSS** stands for **Within-Cluster Sum of Squares**.
- It measures the total variance within clusters — smaller values mean tighter clusters.
- We will store the WCSS for each number of clusters tested and loop over it to find the optimal number of clusters.



```
[ ] km = KMeans(n_clusters=4, random_state=42)
    rfm['Clusters'] = km.fit_predict(rfm_scaled)
```



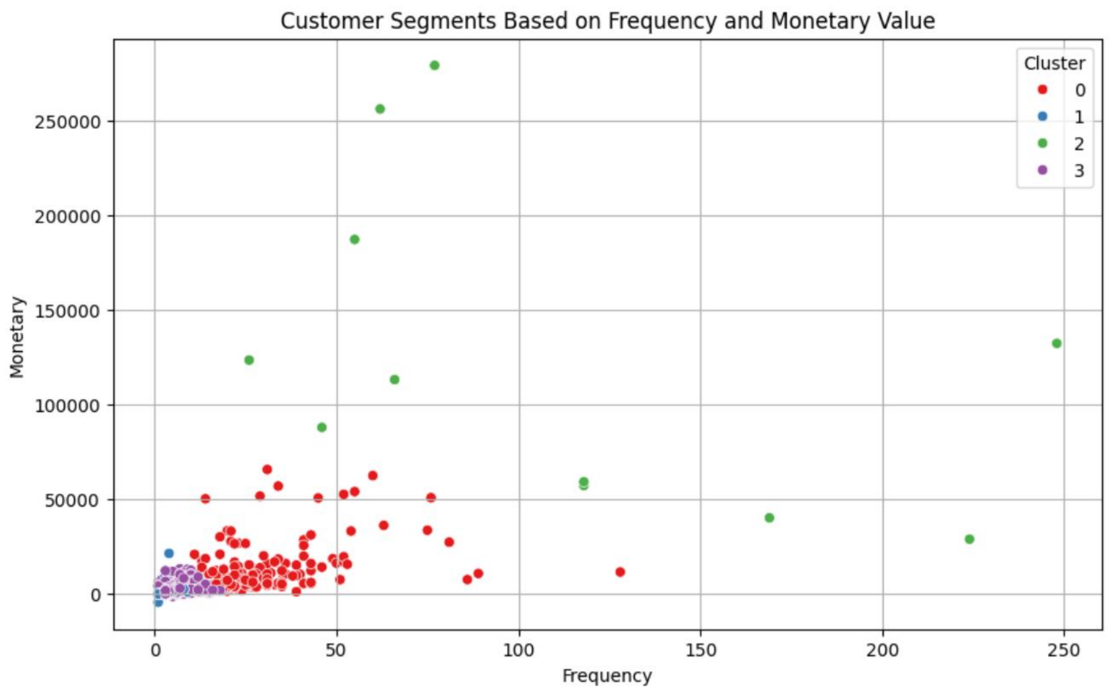
```
rfm.head()
```

	CustomerID	Recency	Frequency	Monetary	R_score	F_score	M_score	RFM_Score	Segment	Clusters
0	12346	326	2	0.00	1	2	1	121	Hibernating	1
1	12347	2	7	4310.00	5	4	5	545	Champion	3
2	12348	75	4	1797.24	2	3	4	234	Others	3
3	12349	19	1	1757.55	4	1	4	414	Potential	3
4	12350	310	1	334.40	1	1	2	112	Hibernating	1

- Creates a **KMeans** clustering model with: n_clusters=4 - the number of clusters to form, random_state=42 - ensures reproducible results.
- fit_predict() → fits the KMeans model on the scaled RFM data (rfm_scaled) and returns the cluster label (0, 1, 2, or 3) for each customer.
- These labels are stored in a new column **"Clusters"** in the rfm DataFrame.

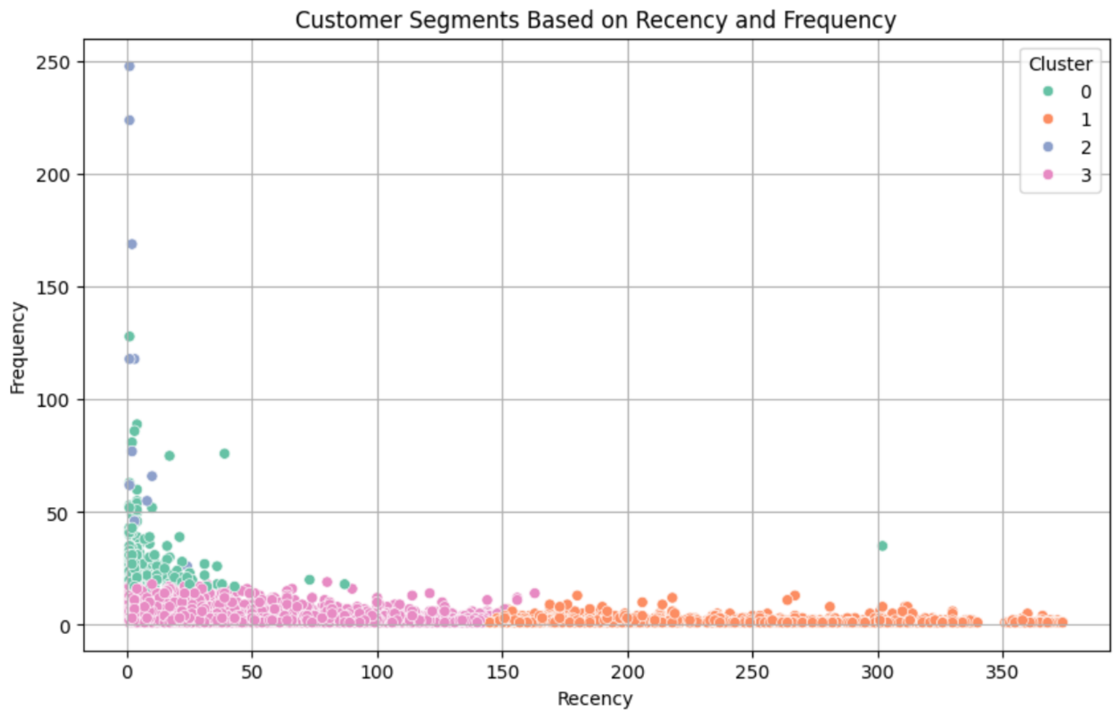
Clustering based Monetary vs Frequency parameter

```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=rfm, x='Frequency', y='Monetary', hue='Clusters', palette='Set1')
plt.title('Customer Segments Based on Frequency and Monetary Value')
plt.xlabel('Frequency')
plt.ylabel('Monetary')
plt.legend(title='Cluster')
plt.grid(True)
plt.show()
```



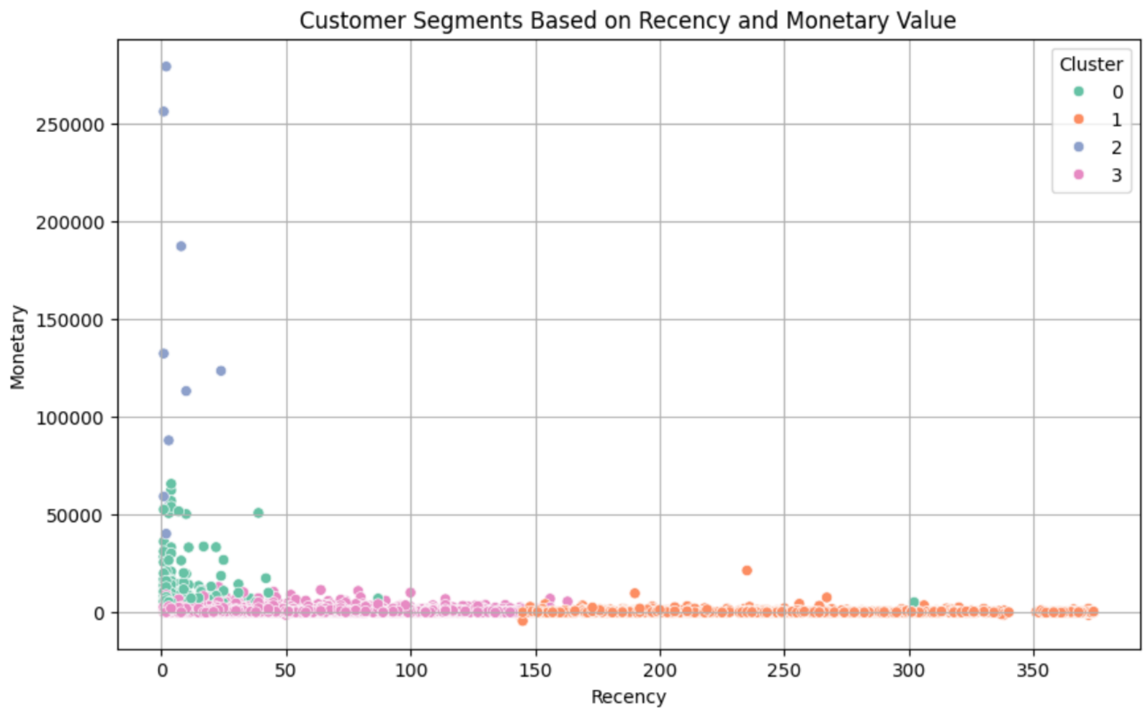
Clustering based Recency vs Frequency parameter

```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=rfm, x='Recency', y='Frequency', hue='Clusters', palette='Set2')
plt.title('Customer Segments Based on Recency and Frequency')
plt.xlabel('Recency')
plt.ylabel('Frequency')
plt.legend(title='Cluster')
plt.grid(True)
plt.show()
```



Clustering based Recency vs Monetary parameter:

```
plt.figure(figsize=(10, 6))
sns.scatterplot(data=rfm, x='Recency', y='Monetary', hue='Clusters', palette='Set2')
plt.title('Customer Segments Based on Recency and Monetary Value')
plt.xlabel('Recency')
plt.ylabel('Monetary')
plt.legend(title='Cluster')
plt.grid(True)
plt.show()
```


Customer Segments Based on Recency and Monetary Value
Cluster
0
1
2
3
Monetary
250000
200000
150000
100000
50000
0
Recency
0
50
100
150
200
250
300
350

Inferences from the Project

The complete set of **inferences** that can be draw from the entire customer segmentation work are:

1. Data Understanding and Cleaning

- The dataset contained sales transaction details, including product descriptions, quantities, prices, dates, customer IDs, and countries.
- Missing values were found in Description and CustomerID, with CustomerID having a large portion of nulls.
- Rows with missing values were removed to ensure accurate analysis, reducing the dataset size but improving reliability.
- A new feature TotalPrice was created (UnitPrice × Quantity), enabling monetary value analysis.

2. Business Insights Before Segmentation

- **Total Customers**: 4,372 unique customers after cleaning.
- **Overall Sales**: Around £8.3 million in total.
- **Monthly Sales Trend**: Strong seasonality, with peaks in November (likely holiday shopping season) and dips in April/February.
- **Daily Sales Trend**: Shows fluctuations with certain days having zero sales, suggesting possible weekends/holidays or stock-outs.

3. Applying KMeans Clustering:

- Traditional RFM scoring gives predefined segments but it can be rigid.
- **K-Means clustering** allowed grouping based purely on patterns in Recency, Frequency, and Monetary values — finding **natural customer groupings** without manual score thresholds.

- Scaling (StandardScaler) ensured no bias due to different value ranges between R, F, and M.

4. Key Findings from Clustering

- **Cluster 0**: Moderate frequency, moderate spending — average but consistent customers.
- **Cluster 1**: Low frequency, low monetary value — inactive/hibernating customers.
- **Cluster 2**: High monetary but moderate recency — valuable customers who may need re-engagement.
- **Cluster 3**: High frequency, high monetary value, recent purchases — **Champions / Loyal customers**.

5. Business Implications

- **Cluster 3 (Champions)**: Retain with loyalty programs, early access to new products, exclusive offers.
- **Cluster 2 (At-risk High Value)**: Re-engage with win-back campaigns, special discounts, reminders.
- **Cluster 0 (Moderate customers)**: Encourage upselling and cross-selling to increase basket size.
- **Cluster 1 (Hibernating)**: Low ROI for aggressive marketing — consider occasional reactivation offers or remove from active campaigns.

6. Overall Conclusion

- Customer segmentation using K-Means provided deeper, data-driven grouping beyond RFM scores.
- Targeted marketing based on segment behaviour can boost sales, improve retention, and reduce churn.
- This analysis can serve as the foundation for predictive modelling (e.g., predicting which cluster a new customer will belong to) and personalized marketing automation.

Conclusion

This project successfully applied **data cleaning, feature engineering, exploratory analysis, and K-Means clustering** to perform customer segmentation for a retail business. The dataset, after handling missing values and enriching it with the TotalPrice metric, revealed valuable insights into customer behaviour and sales patterns.

Exploratory analysis highlighted **seasonal trends**, with sales peaking during the holiday season and fluctuating significantly on a daily basis. These patterns provide actionable intelligence for inventory planning and marketing timing.

Using **RFM (Recency, Frequency, Monetary) analysis** as the foundation and scaling the data for consistency, K-Means clustering grouped customers into **four distinct segments**, each representing unique purchasing behaviours - from high-value loyal customers to dormant, low-engagement ones.

These segments enable **data-driven marketing strategies**:

- Retain loyal, high-value customers with exclusive benefits.
- Re-engage at-risk but valuable customers with targeted offers.
- Nurture moderate customers through upselling and cross-selling.
- Minimize marketing spend on low-engagement customers while using selective reactivation campaigns.

Overall, this segmentation approach allows the business to **personalize communication, optimize resource allocation, and enhance customer lifetime value**. The methodology is scalable, meaning the business can re-run the segmentation periodically to adapt to changing customer patterns, ultimately supporting **sustainable growth and improved profitability**.