PREPROCESSING

- **Preprocessing aims to prepare data for analysis, cleaning, transforming, and organizing it in a way that makes it easier to work with and improves the accuracy of our results.**

```
#Importing required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as snb
from sklearn import preprocessing
```

- **This code imports necessary Python libraries.**
- **Pandas for data manipulation.**
- **NumPy for numerical computing.**
- **Matplotlib and Seaborn for visualization.**
- **scikit-learn's preprocessing module for machine learning (Preprocessing).**

```
#Collecting the datasets
df1 = pd.read_csv("/content/drive/MyDrive/Singapore Resale /ResaleFlatPricesBasedonApprovalDate19901999.csv")
df2 = pd.read_csv("/content/drive/MyDrive/Singapore Resale /ResaleFlatPricesBasedonApprovalDate2000Feb2012.csv")
df3 = pd.read_csv("/content/drive/MyDrive/Singapore Resale /ResaleFlatPricesBasedonRegistrationDateFromMar2012toDec2014.csv")
df4 = pd.read_csv("/content/drive/MyDrive/Singapore Resale /ResaleFlatPricesBasedonRegistrationDateFromJan2015toDec2016.csv")
df5 = pd.read_csv("/content/drive/MyDrive/Singapore Resale /Resaleflatpricesbasedonregistrationdatefromjan2017onwards.csv")
```

- **This code reads multiple CSV files containing Singapore resale flat price data from different time periods and stores each dataset into separate Data Frame variables (df1, df2, df3, df4, df5) .**

- **I collected these datasets from the data source provided in the problem statement document.**

```
#checking the data shapes
df = [df1,df2,df3,df4,df5]
for i in df:
  print(i.shape)
```

- **This code iterates (Repeat a process multiple times-Looping) through a list of Data Frame variables (df1, df2, df3, df4, df5).**
- **And for each Data Frame, it prints its shape (indicating the number of rows and columns in each dataset).**

MERGING THE 5 DATA FRAMES

```
#concating the all data frames into a single data frame
df = pd.concat([df1,df2,df3,df4,df5],ignore_index=True)
```

- **This code concatenates all the Data Frame variables (df1, df2, df3, df4, df5) into a single Data Frame named "df".**
- **While ignoring the index of the original Data Frames.**

- **Using concat function to merge multiple dataset into one which simplifies future data processing by consolidating all the data into a single Data Frame, thereby facilitating easier analysis and manipulation.**

- **The ignore_index=True parameter ensures that the original indices of the individual Data Frames are ignored and a new index is generated for the concatenated Data Frame.**

DISPLAYING THE DATA FRAMES

```
#Viewing head data
df.head(10)
```

- **This code displays the first 10 rows of the Data Frame "df," allowing for a quick inspection of the data's structure and content.**

```
#Displaying tail data
df.tail(10)
```

- **This code displays the last 10 rows of the Data Frame "df," providing insight into the structure and content of the data from the end of the dataset.**

```
#Checking data shape and info
df.shape
```

- **This code retrieves the shape of the Data Frame "df," which indicates the number of rows and columns in the dataset.**
- **This information helps to understand the size and dimensions of the data being analyzed.**

```
df.info()
```

- **This code provides detailed information about the Data Frame "df", including the data types (Integers, floating-point numbers, strings, Boolean values, and date time objects) of each column, the number of non-null values in each column, and the memory usage.**

CHECKING MISSING VALUES

```
#displaying Null values
df.isnull().sum()
```

- **This code displays the number of null (missing) values in each column of the Data Frame "df" by applying the .isnull() method to identify null values and then using .sum() to count the null values in each column.**

- **Identifying and handling null values is a fundamental step in data preprocessing, essential for accurate and meaningful analysis.**

- **This code drops the column named "remaining_lease" from the Data Frame "df" using the drop() method and assigns the modified Data Frame back to the variable "df".**
- **After dropping the "remaining_lease" column the Data Frame "df" will display without that specific column.**

- **After identifying missing values, the decision to drop the 'remaining_lease' column was made due to the lack of available data.**

- **Since the majority of rows in this column have missing values, its removal does not significantly affect the dataset.**

VISUALIZATION

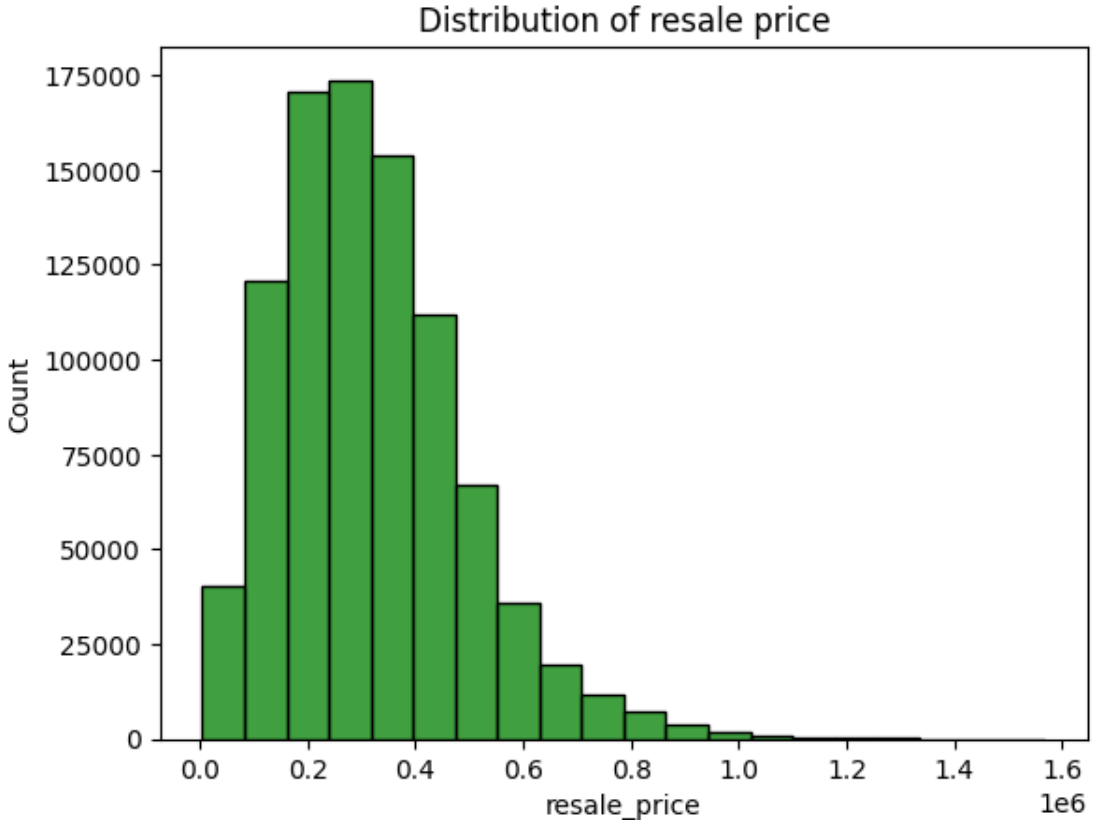
- **Visualization allow us to understand plot data more effectively because it enables us to observe the distribution of specific data points, such as range, peaks, patterns and outliers.**

```
#Distribution of resale
snb.histplot(df['resale_price'], bins = 20, color='Green')
plt.title('Distribution of resale price')
plt.show()
```

- **This code generates a histogram to visualize the distribution of resale prices from the Data Frame "df".**
- **It uses Seaborn's histplot() function, specifying the column 'resale_price' as the data to be plotted .**
- **The histogram is divided into 20 bins and colored green.**
- **Additionally, it includes a title "Distribution of resale price" and displays the plot using Matplotlib's plt.show() function.**

- **"20 bins" refers to the number of intervals the resale prices are divided into for the histogram. In this case, the histogram will display the frequency distribution of resale prices across 20 equally spaced intervals.**
- **"colored green" specifies the color of the histogram bars. The histogram will be displayed with green bars, indicating the visual representation of the resale price distribution.**

- **Visualization is used to understand the distribution of resale prices within the dataset. By plotting a histogram of resale prices with 20 bins and colouring the bars green, we can visually assess how resale prices are distributed across different price ranges.**
- **This helps in identifying patterns, outliers, and overall trends in resale prices, providing valuable insights for analysis and decision-making.**



- **In a resale histogram, the x-axis displays the price values, and the y-axis represents the count of occurrences.**
- **Observing this, a right-skewed distribution is indicated, implying that most data points are concentrated towards the lower end.**
- **This distribution pattern is characterized by a peak occurring towards the left side.**

- **The right-skewed distribution means that the tail of the distribution extends towards the right side, with most of the data clustered towards the left and a few large values pulling the mean to the right.**

- **A distribution with a peak towards the left side typically indicates that the majority of the data points are clustered at lower values, tapering off towards higher values.**

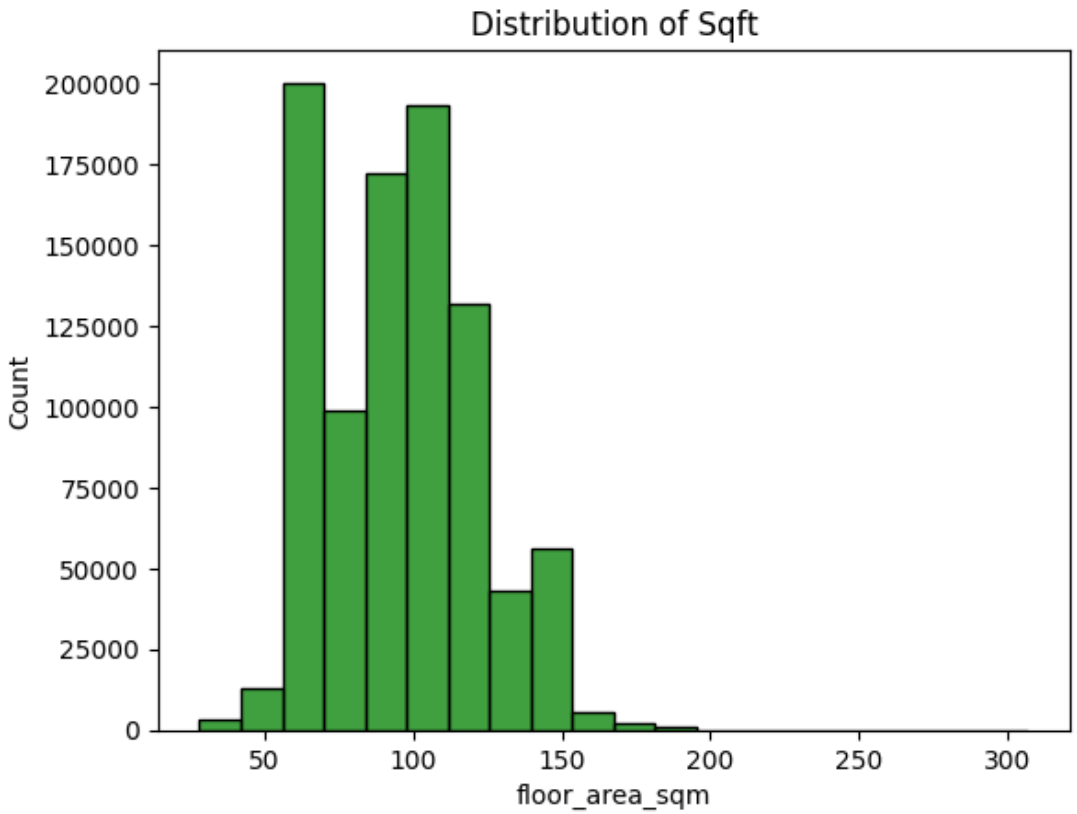
```
#Distribution of floor area
snb.histplot(df['floor_area_sqm'], bins = 20, color='Green')
plt.title('Distribution of Sqft')
plt.show()
```

- **This code generates a histogram to visualize the distribution of floor area from the Data Frame "df".**
- **It uses Seaborn's histplot() function, specifying the column 'floor_area_sqm' as the data to be plotted.**
- **The histogram is divided into 20 bins and colored green.**
- **Additionally, it includes a title "Distribution of sqft" and displays the plot using Matplotlib's plt.show() function.**

- **"20 bins" indicates that the histogram will be divided into 20 equal intervals along the x-axis, where the data values will be grouped.**
- **"Colored green" specifies that the bars in the histogram will be displayed in the color green. This color choice helps to visually distinguish the histogram bars from the background and other elements in the plot.**

- **Visualization is utilized for better comprehension of the distribution of floor areas (represented by the 'floor_area_sqm' variable).**

- **By visualizing the data through a histogram with 20 bins (intervals) and green colour, we can easily grasp how floor area values are distributed across different ranges. This aids in understanding the dataset's characteristics and facilitates decision-making during analysis.**

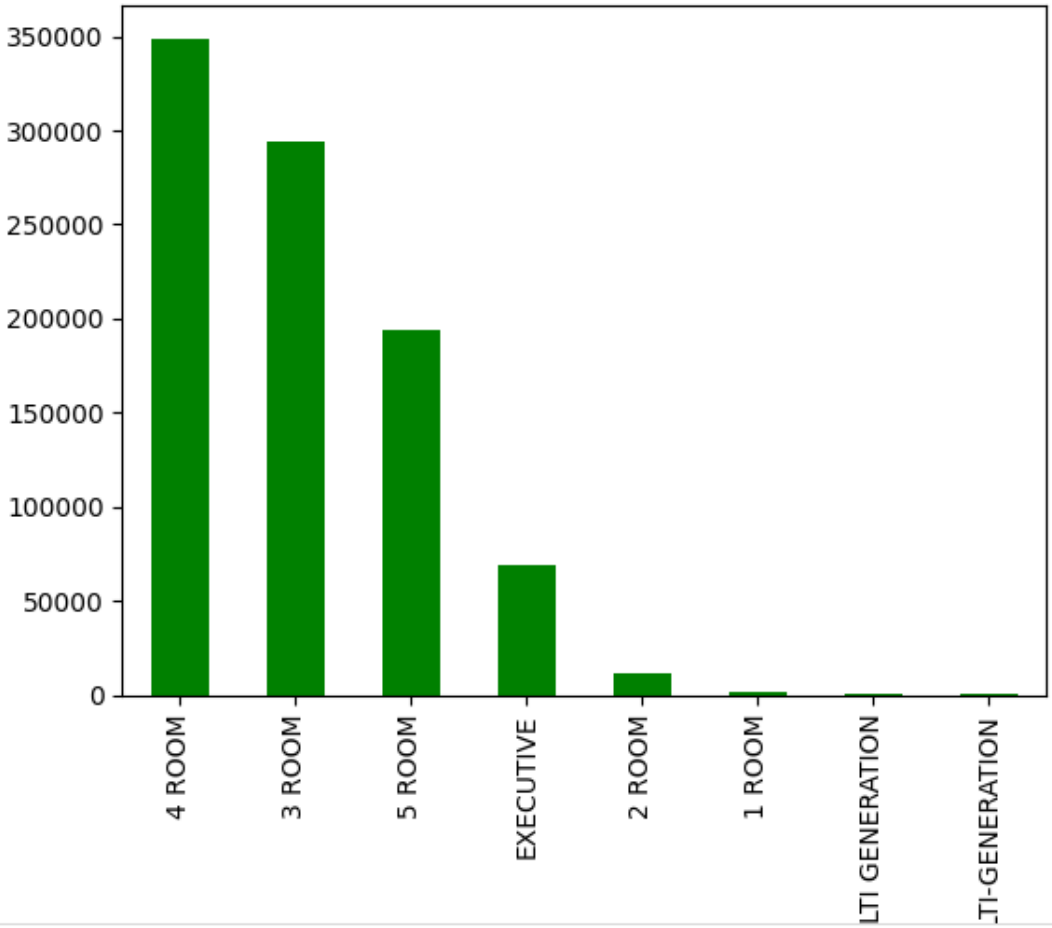


- **When analyzing floor square, it focus on the count within the range of 50 to 300. Observing this, we find that the highest count falls within the range of 50 to 100, providing insight into the distribution of floor square.**

```
#Distribution of flat
df['flat_type'].value_counts().plot(kind = 'bar',color='Green')
plt.show()
```

- **In this code visualization is used to depict the distribution of flat types within the dataset.**

- **The .value_counts() method calculates the frequency of each unique value in the 'flat_type' column.**
- **The .plot(kind='bar') creates a bar plot to represent these frequencies.**
- **The bars are colored green for visual clarity.**
- **This visualization allows for a quick understanding of the prevalence of different types of flats in the dataset.**



- **Using a bar chart simplifies the comparison of different categories. Here, we display the counts of unique values for flat types, enabling us to identify the most frequently occurring value.**

DEALING WITH FEATURES

```
#Flat type
df['flat_type'].value_counts()
```

- **This code displays the count of each unique value in the 'flat_type' column.**
- **It provides insight into the distribution of flat types.**

```
# Unique lables in flat_type column
df['flat_type'].unique()
```

- **This code retrieves the unique labels present in the 'flat_type' column of the Data Frame, displaying the distinct types of flats listed in the dataset.**

- **In the 'flat_type' column, unique labels refer to the distinct categories or types of flats listed in the dataset.**
- **Each label represents a specific type of flat, such as '1 ROOM', '3 ROOM', '4 ROOM', '5 ROOM', '2 ROOM', 'EXECUTIVE', 'MULTI GENERATION', and 'MULTI-GENERATION'.**
- **These labels are unique in the sense that each one represents a different type of residential unit.**

ENCODING THE FLAT TYPE

```
# Assigning the value-pairs to be replaced
cat = {'1 ROOM': 1,
       '2 ROOM':2,
       '3 ROOM':3,
       '4 ROOM':4,
       '5 ROOM':5,
       'EXECUTIVE':6,
       'MULTI GENERATION':7}
```

- **This code creates a dictionary named 'cat' to map categorical flat types to numerical values for the 'flat_type' column.**
- **Each flat type is assigned a numerical representation for further processing.**

```
# Replacing the values in df
df['flat_type'] = df['flat_type'].replace(cat)
df['flat_type'].value_counts()
```

- **This code replaces the categorical values in the 'flat_type' column of the Data Frame 'df' with their corresponding numerical representations as defined in the dictionary 'cat'.**
- **After replacement, it displays the count of each unique numerical value in the 'flat_type' column to show the distribution of flat types represented by their numerical equivalents.**

```
# Collapsing the flat_type categories
mapping={'MULTI-GENERATION':'MULTI GENERATION'}

df['flat_type'] = df['flat_type'].replace(mapping)
df['flat_type'].unique()
```

- **In this code the 'flat_type' categories are collapsed by renaming the category 'MULTI-GENERATION' to 'MULTI GENERATION' to ensure consistency in representation.**
- **The replace() method is used to perform this renaming operation.**
- **After the replacement, the unique() method is applied to display the unique values in the 'flat_type' column, confirming the collapsed categories.**

- **We use this function to ensure consistency in the representation of categories within the 'flat_type' column. By collapsing similar categories like 'MULTI-GENERATION' and 'MULTI GENERATION' into a single category, 'MULTI GENERATION'**
- **This helps in maintaining data integrity and ensures that subsequent analysis tasks are not affected by variations in category names**

DEALING WITH FLAT TYPE

```
# Flat_model column
print(df['flat_model'].nunique())
df['flat_model'].unique()
```

- **This code displaying the number of unique values in the 'flat_model' column using the nunique() function.**

- **Followed by displaying all unique values in the 'flat_model' column using the unique() function.**

```
#changing into small letters
df['flat_model'] = df['flat_model'].str.lower()
```

- **This code converts all strings in the 'flat_model' column to lowercase using the str.lower() function.**

- **This ensures uniformity in the representation of flat model names, facilitating easier comparison and analysis of the data**

```
#flat_model value counts
df['flat_model'].value_counts()
```

- **This code displays the count of each unique value in the 'flat_model' column, showing how many occurrences there are for each flat model within the dataset.**

- **It provides insight into the distribution of flat models and helps in understanding which models are more prevalent in the dataset.**

ENCODING THE FLAT MODEL

```
ecode= preprocessing.LabelEncoder()

flat_modelcode= ecode.fit_transform(df['flat_model'])
df.insert(loc = 8,
          column = 'flat_modelcode',
          value = flat_modelcode)
df['flat_modelcode'].value_counts()
```

- **An instance of Label Encoder from the preprocessing module is created and stored in the variable ecode.**
- **The fit_transform() method of the Label Encoder instance is applied to encode the categorical values in the 'flat_model' column into numerical labels.**
- **The resulting numerical labels are stored in the variable flat_modelcode.**
- **A new column named 'flat_modelcode' is inserted into the Data Frame 'df' at index 8, containing the encoded numerical labels.**
- **Finally, the value_counts() function is used to display the count of each unique value in the 'flat_modelcode' column, showing the distribution of encoded flat model labels within the dataset.**

- **The parameter loc=8 specifies the position where the new column 'flat_modelcode' will be inserted into the Data Frame 'df'. In this case, loc=8 indicates that the new column will be inserted at index position 8, meaning it will be placed as the 9th column in the Data Frame.**

TOWN COLUMN

```
# Town column
df['town'].nunique()
```

- **This code retrieves the number of unique values in the 'town' column of the Data Frame 'df'**

- **This provides insight into the variety of towns represented in the dataset.**

```
#checking value counts
df['town'].value_counts()
```

- **This code displays the count of each unique value in the 'town' column, indicating how many occurrences there are for each town within the dataset.**

- **It helps in understanding the distribution of flats across different towns and provides insight into the composition of the dataset in terms of geographical locations.**

```
# Creating a column with encoded value
town = ecode.fit_transform(df['town'])
df.insert(loc = 2,
          column = 'town_code',
          value = town)
```

- **In this code an instance of Label Encoder from the preprocessing module is created and stored in the variable ecode.**
- **The fit_transform() method of the Label Encoder instance is applied to encode the categorical values in the 'town' column into numerical labels.**
- **The resulting numerical labels are stored in the variable town.**

- **loc = 2: Specifies the index location where the new column will be inserted. In this case, the new column will be placed at index position 2 (i.e., the third column).**
- **column = 'town_code': Specifies the name of the new column to be inserted, which will be 'town_code'.**
- **value = town: Specifies the values to be inserted into the new column. These values are taken from the 'town' variable, which contains the encoded numerical labels for towns.**

```
# Encoded value count
df['town_code'].value_counts()
```

- **This code displays the count of each unique value in the 'town_code' column, revealing how many occurrences there are for each encoded numerical label representing towns within the dataset.**

- **"occurrences" - represent the frequency with which each unique value appears in the 'town_code' column of the dataset.**

SPLITING THE COLUMNS

```
# Print header of column
df['storey_range'].head()
```

- **This code prints the first few values from the 'storey_range' column of the Data Frame 'df', providing a quick view of the data in that specific column.**

- **The .head() function in Pandas is used to display the first few rows of a Data Frame.**

```
# Splitting storey range column
storey=df['storey_range'].str.split(' TO ',expand = True)
storey.head()
```

- **This code splits the 'storey_range' column of the Data Frame 'df' into two separate columns using the string method .str.split(' TO ', expand=True).**
- **The resulting Data Frame, stored in the variable 'storey', contains two columns: one for the lower limit of the storey range and another for the upper limit.**
- **The .head() function then displays the first few rows of this new Data Frame to show the split values.**

- .str.split(' TO ', expand=True) means that the string in the 'storey_range' column will be split into two parts wherever the substring ' TO ' is encountered.
- The resulting split parts will be returned as separate columns in the Data Frame, with one column containing the values before ' TO ' and another column containing the values after ' TO '.
- Setting expand=True ensures that the split parts are returned as separate columns instead of a list.

```
# Creating the storey minimum values as column
df.insert(loc = 6,
          column = 'storey_min',
          value = storey[0])
# Creating the storey maximum values as column
df.insert(loc = 7,
          column = 'storey_max',
          value = storey[1])
```

- In this code Two new columns, 'storey_min' and 'storey_max', are inserted into the Data Frame 'df'.
- The values for 'storey_min' are taken from the first column of the 'storey' Data Frame, which contains the lower limit of the storey range.
- The values for 'storey_max' are taken from the second column of the 'storey' Data Frame, which contains the upper limit of the storey range.
- These values are inserted into their respective columns at index positions 6 and 7 in the Data Frame 'df'.

```
#Spliting the month columns
month=df['month'].str.split('-',expand = True)
month.head()
```

- **This code splits the 'month' column of the Data Frame 'df' into two separate columns using the string method .str.split('-', expand=True).**
- **The resulting Data Frame, stored in the variable 'month', contains two columns: one for the year and another for the month.**
- **The .head() function then displays the first few rows of this new Data Frame to show the split values.**

- **Setting expand=True ensures that the split parts are returned as separate columns instead of a list.**

```
# Creating the year values as column
df.insert(loc = 1,
          column = 'selling_year',
          value = month[0])
# Creating the storey maximum values as column
df.insert(loc = 2,
          column = 'selling_month',
          value = month[1])
```

- **In this code Two new columns, 'selling_year' and 'selling_month', are inserted into the Data Frame 'df'.**
- **The values for 'selling_year' are taken from the first column of the 'month' Data Frame, which contains the year portion of the 'month' column after splitting.**
- **The values for 'selling_month' are taken from the second column of the 'month' Data Frame, which contains the month portion of the 'month' column after splitting.**
- **These values are inserted into their respective columns at index positions 1 and 2 in the Data Frame 'df'.**

- loc = 1 specifies the index position where the new column 'selling_year' will be inserted into the Data Frame 'df'. It indicates that the new column will be placed at index position 1 (i.e., the second column).
- loc = 2 specifies the index position where the new column 'selling_month' will be inserted into the Data Frame 'df'. It indicates that the new column will be placed at index position 2 (i.e., the third column).
- So, loc determines the position of the new columns within the Data Frame.
- month[0] contains the values representing the year portion of the 'month' column after splitting.
- month[1] contains the values representing the month portion of the 'month' column after splitting.
- These values are being inserted into the respective columns ('selling_year' and 'selling_month') of the Data Frame 'df'.

```
df = df[['selling_month', 'selling_year','town', 'town_code', 'flat_type',  'block',  'street_name',  'storey_min',
         'storey_max',  'storey_range', 'floor_area_sqm', 'flat_model', 'flat_modelcode',
         'lease_commence_date', 'resale_price' ]]
```

- This code reorders the columns of the Data Frame 'df' according to the specified list of column names.
- The columns are rearranged to the order specified in the list.
- The Data Frame 'df' will now have its columns arranged in this specific order.

- Reording columns making it easier to analyze and interpret the data for subsequent tasks.

```
df.head()
```

- **The df.head() function displays the first few rows of the Data Frame df, providing a quick overview of the data. It allows us to inspect the top records of the Data Frame to ensure that the data looks as expected after transformation.**

CHECKING DATA TYPES

```
df.dtypes
```

- **The df.dtypes provides the data types of each column in the Data Frame df.**

- **It returns a Series containing the data types of all columns, where the index represents the column names and the values represent the corresponding data types.**
- **This information is helpful for understanding the structure of the Data Frame and ensuring that each column has the appropriate data type for analysis and manipulation.**

```
#changing some dtype to numeric data
df_copy = df.copy()
df_copy.loc[:, 'selling_month'] = df_copy['selling_month'].astype(int)
df_copy.loc[:, 'selling_year'] = df_copy['selling_year'].astype(int)
df_copy.loc[:, 'storey_min'] = df_copy['storey_min'].astype(int)
df_copy.loc[:, 'storey_max'] = df_copy['storey_max'].astype(int)
```

- **This code converts specific columns to numeric data type to ensure they are represented as integers for further analysis.**

- **A copy of the Data Frame df is created and stored in df_copy.**
- **Selected columns ('selling_month', 'selling_year', 'storey_min', 'storey_max') are converted to integer data type using .astype(int).**
- **The .loc[] method is used to locate the specified columns and apply the data type conversion.**

```
df.dtypes
```

- **After performing the data type conversions, the df_copy.dtypes display the updated data types of each column in the Data Frame df_copy, showing that the selected columns have been converted to integer data type.**

CORRELATION CHECK

- **When examining the relationship between two variables, whether numerical or categorical, we assess the extent of impact.**
- **In a positive relationship, as x increases, y also increases; conversely, as x increases, y decreases.**

```
corr_df = df[['selling_month', 'selling_year', 'town_code', 'storey_min', 'storey_max',
              'floor_area_sqm', 'flat_modelcode', 'lease_commence_date', 'resale_price']].dropna().corr()

# Create a heatmap
plt.figure(figsize=(10, 8))
snb.heatmap(corr_df,annot=True)
plt.title("Correlation Matrix")
plt.show()
```

- **This code generates a heat map visualizing the correlation matrix between selected numerical columns in the Data Frame, providing insights into the relationships between these variables.**

- **A subset of the Data Frame df containing selected columns is created, followed by dropping any rows with missing values using .dropna().**
- **The .corr() method calculates the pairwise correlation between the selected columns.**
- **figsize=(10, 8) specifies the dimensions of the figure in inches The width of the figure is set to 10 inches, and the height is set to 8 inches.**
- **A heatmap is created using seaborn (snb) library, displaying the co andrrelation matrix using plt.figure() and snb.heatmap().**
- **The heatmap is annotated with correlation values using annot=True.**
- **Finally, the heatmap is displayed with the title "Correlation Matrix" using plt.title() and plt.show().**

```
#downloading cleaned datasets
df.to_csv('final.csv',index=False)
```

- **This code saves the Data Frame df to a CSV file named 'final.csv' without including the index column.**
- **This file will contain the cleaned dataset, allowing it to be easily accessed and shared for further analysis.**