**Hexaware Technologies**

Python-Case Study

**Step 1: Select a Real-World Dataset**

For this case study, I have selected the **Rental Pricing Dataset** of Malaysia, specifically focused on rental prices in Kuala Lumpur and Selangor. This dataset is sourced from **Kaggle** and provides data on various property listings, including details like the property type, location, number of rooms, rental price, and more.

* **Dataset Source:** [Kaggle - Rental Pricing Dataset](https://www.kaggle.com/datasets/ariewijaya/rent-pricing-kuala-lumpur-malaysi)
* **Dataset Overview:** This dataset contains **19,991 rows** and **14 columns**, which fulfills the requirement of having at least **10,000 rows** and **5 columns** as specified in the project instructions.

**Shape of the Dataset:**

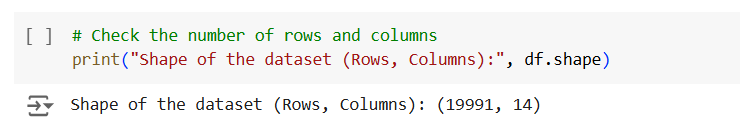
* **Rows:** 19,991
* **Columns:** 14

**Step 2: Perform data preparation & cleaning**

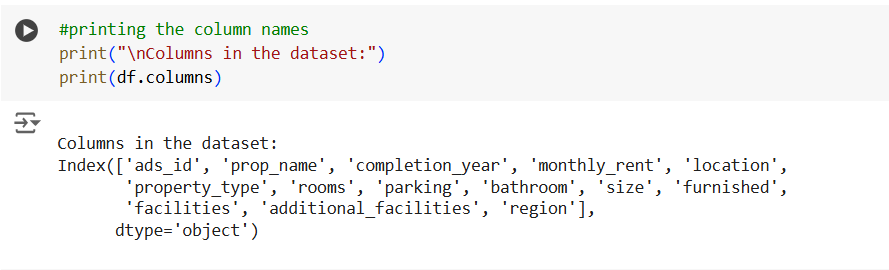
* **Load the dataset into a data frame using Pandas**
* **Import Pandas**: We use the import pandas as pd statement to bring the Pandas library into our Python environment, which is necessary for handling data in tabular form (DataFrames).
* **Load the Data**: We use **pd.read\_csv(file\_path)** to load the dataset from the specified file path (file\_path). This function reads the CSV file and converts it into a DataFrame, which is a two-dimensional table-like structure that allows easy manipulation of the data.
* **Preview Data**: The **df.head()** function is used to preview the first few rows (default 5 rows) of the dataset. It helps us quickly understand the structure of the data, including column names and a sample of the entries.



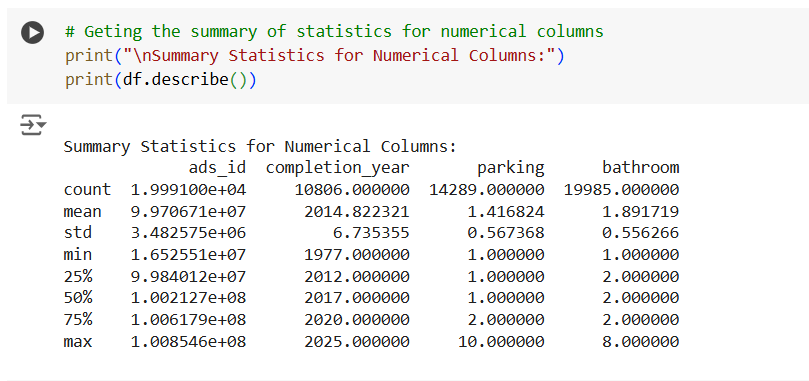
* **Explore the number of rows & columns, ranges of values etc**
  1. **Checking the Shape of the Dataset**: We use **df.shape** to get the number of rows and columns in the dataset. This gives us a quick overview of the dataset's size, which is important for understanding the volume of data and deciding on the next steps for analysis. The result is a tuple of **(rows, columns)**.



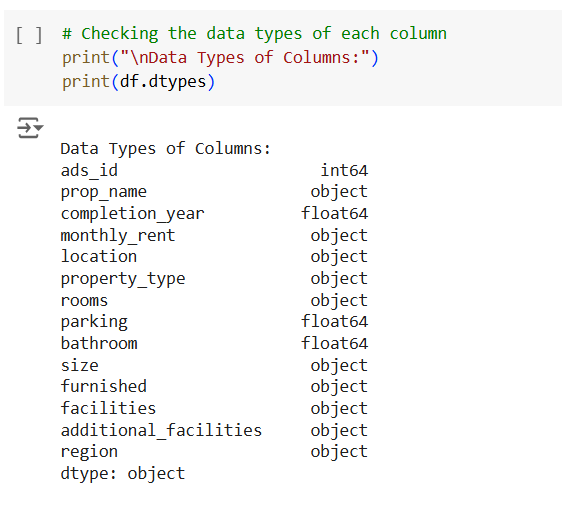
* 1. **Printing Column Names:** Using **df.columns,** we retrieve and print the names of all the columns in the dataset. This helps us understand the structure of the data and what each column represents. It’s useful for determining if the dataset contains the necessary information for further analysis.



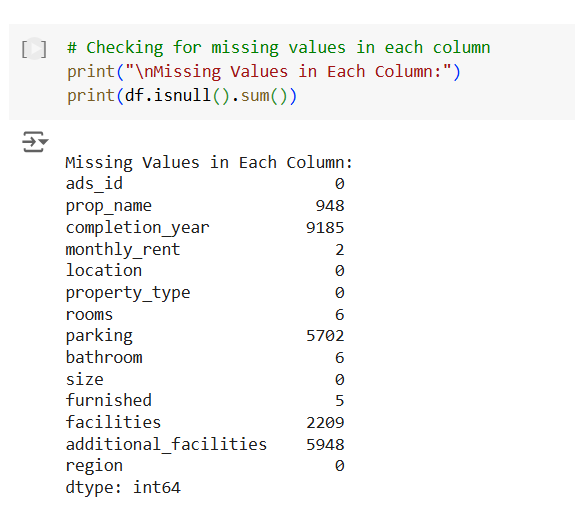
* 1. **Summary Statistics for Numerical Columns**: The **df.describe()** function provides a statistical summary of numerical columns in the dataset. It includes measures like mean, standard deviation, minimum, maximum, and percentiles. This is important to understand the distribution and spread of the numerical data.



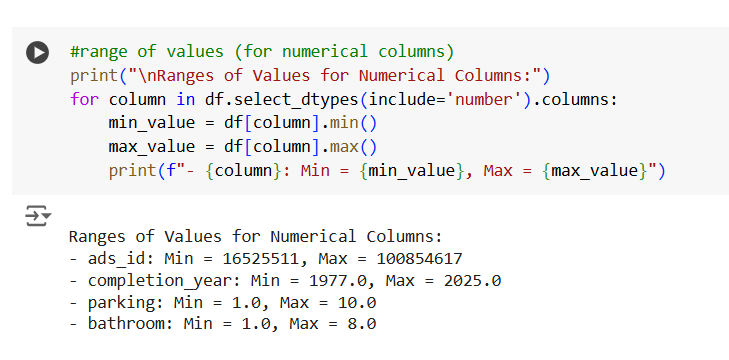
* 1. **Checking Data Types of Columns**: The **df.dtypes** function shows the data types of each column in the dataset. This helps us ensure that the columns are of the correct data types (e.g., numerical, categorical) for analysis. Incorrect data types may need to be corrected during data cleaning.



* 1. **Checking for Missing Values**: We use **df.isnull().sum()** to check for missing values in each column. This function returns the count of missing values (**NaN**) for each column. Identifying missing values early on helps us decide how to handle them (e.g., by filling, dropping, or imputing).



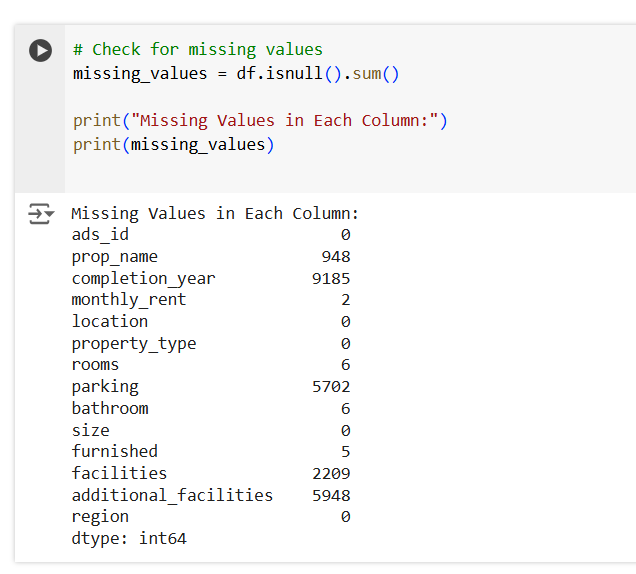
* 1. **Range of Values for Numerical Columns**: For each numerical column, we calculate the minimum and maximum values using **df[column].min()** and **df[column].max()**. This provides insight into the range of values in the dataset, helping us identify outliers or unrealistic values.



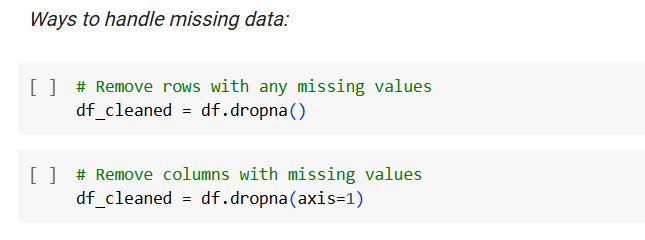
* **Handle missing, incorrect and invalid data**

**1. Identifying Missing Values:**

**df.isnull().sum()** checks for missing (NaN) values in each column and returns the count of missing entries per column. It helps us identify which columns have missing data, which is essential for deciding how to handle them (e.g., removing or imputing).

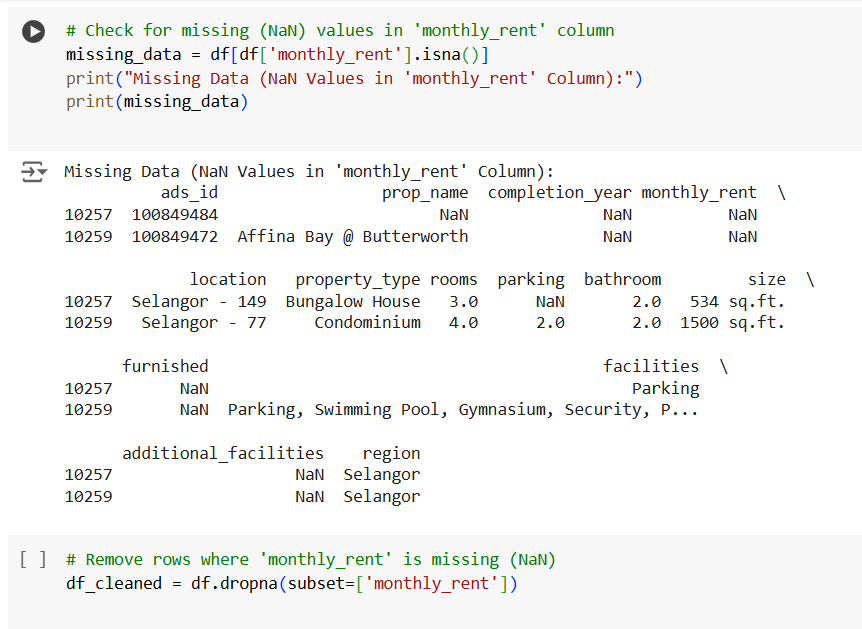


* **Removing Rows with Missing Values: df.dropna()** removes rows that contain any missing values. This is suitable when the missing data is insignificant and not expected to drastically affect the analysis.
* **Removing Columns with Missing Values: df.dropna(axis=1)** removes columns that contain any missing values. This can be useful when columns have a large proportion of missing data, making them unreliable for analysis.



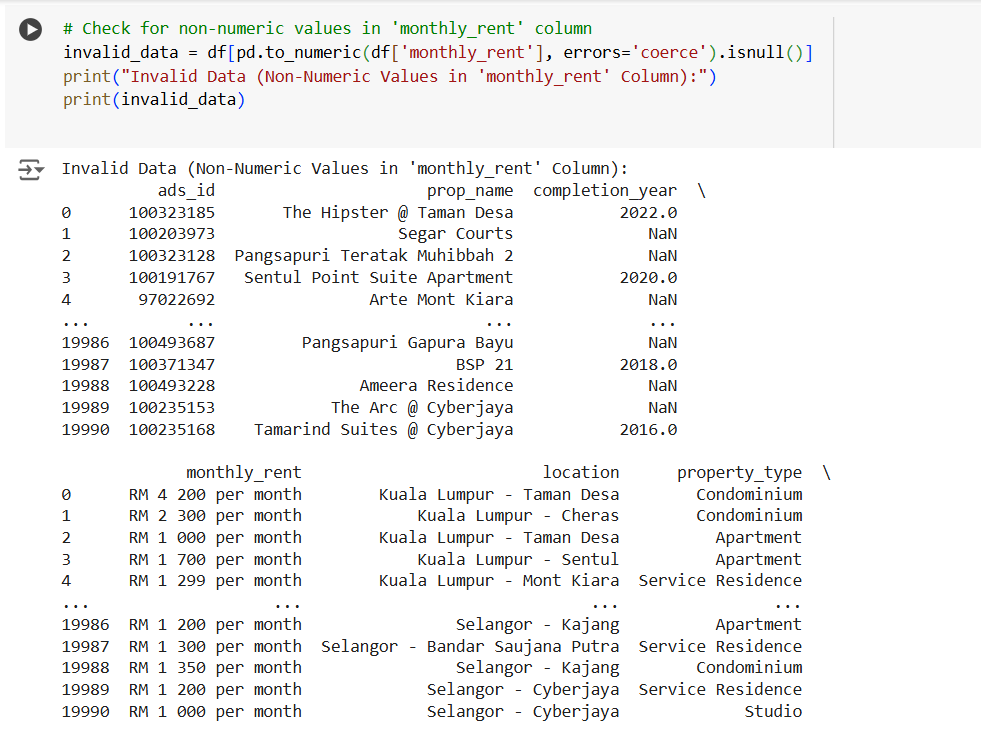
**2. Handling Incorrect Data:**

* **Identifying Missing Data in Specific Columns:** **df[df['monthly\_rent'].isna()]** filters and displays rows where the **monthly\_rent** column has NaN values. This helps isolate problematic rows that need further attention.
* **Removing Rows with Missing Values in Specific Columns:** As mentioned earlier, **df.dropna(subset=['monthly\_rent'])** removes rows where the **monthly\_rent** column is missing, ensuring that the column has complete data for further analysis.

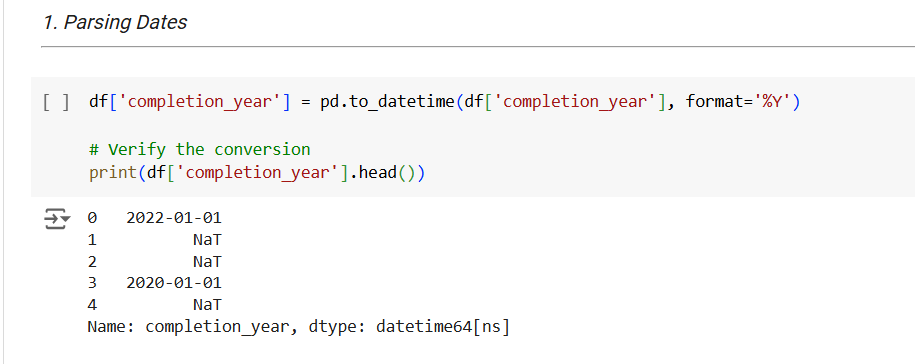


**3. Handling Invalid Data:**

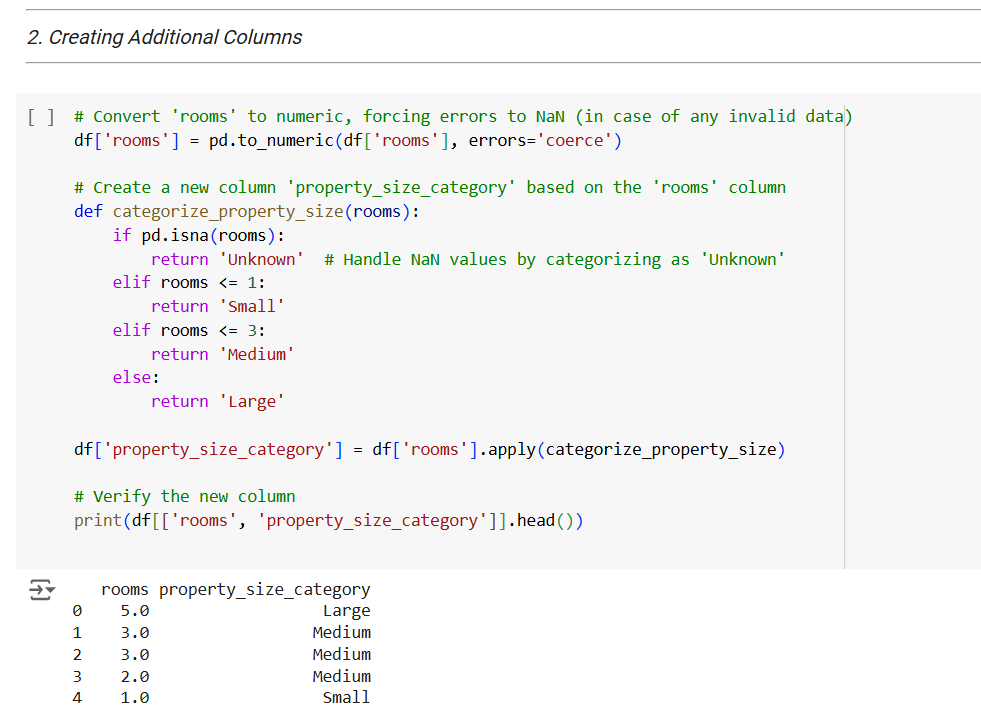
**Identifying Invalid Data (Non-Numeric Values in a Numeric Column): pd.to\_numeric(df['monthly\_rent'], errors='coerce')** attempts to convert values in the **monthly\_rent** column to numeric. The **errors='coerce'** argument ensures that any non-numeric values are replaced with NaN. **df[pd.to\_numeric(df['monthly\_rent'], errors='coerce').isnull()]** filters rows with non-numeric values, which need to be cleaned or corrected for valid analysis.

****

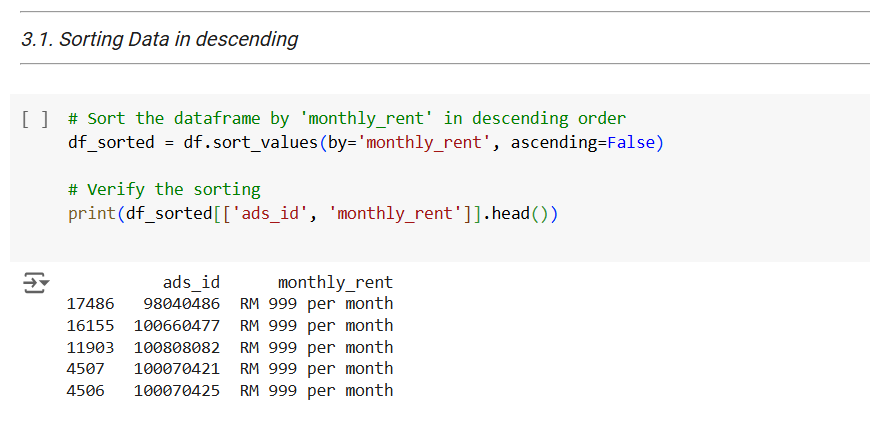
* **Perform any additional steps (parsing dates, creating additional columns, merging multiple dataset etc.)**
* **Parsing Dates**: The **pd.to\_datetime()** function is used to convert the **'completion\_year'** column to a datetime format. This ensures that the column is treated as a date, which is useful for time-based analyses or visualizations. The format **'%Y'** specifies that the year is the only part of the date.



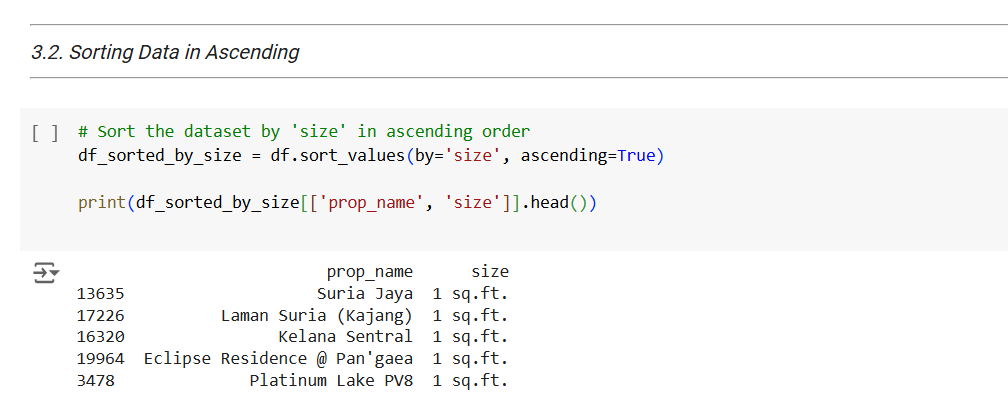
* **Creating Additional Columns**: The **pd.to\_numeric()** function is applied to the 'rooms' column to convert it to numeric values. This helps handle invalid or non-numeric data by coercing them into NaN values. The **apply()** method is then used to create a new column, 'property\_size\_category', based on the number of rooms, categorizing properties into 'Small', 'Medium', or 'Large', making it easier to segment properties for analysis.



* **Sorting Data in Descending Order**: The **sort\_values()** function is used to sort the dataset by the 'monthly\_rent' column in descending order. This is helpful for identifying the highest rental properties and prioritizing them for analysis or reporting.



* **Sorting Data in Ascending Order**: Sorting by the 'size' column in ascending order helps arrange the properties by their size, making it easier to understand the distribution of property sizes and potentially identify trends or anomalies.



* **Data Visualization** :A bar chart is created using **value\_counts()** to visualize the distribution of different property types. This helps in understanding the frequency of each property type, providing insights into the dataset's composition. The chart is styled with **matplotlib** for a clearer and more informative presentation.

