Pandas, Matplotlib, and Seaborn are three popular Python libraries used for data manipulation, analysis, and visualization. Each of these libraries serves a specific purpose in the data science and data analysis workflow:

Pandas:

Purpose: Pandas is a powerful data manipulation and analysis library that provides data structures and functions for working with structured data, such as tables and time series.

Key Features:

DataFrames: Two-dimensional, labeled data structures that resemble tables or spreadsheets, making it easy to work with structured data.

Series: One-dimensional labeled arrays for handling data with an index, such as time series data.

Data cleaning and preparation: Pandas provides numerous functions for handling missing data, merging and reshaping datasets, and performing data transformations.

Data exploration: Pandas allows you to quickly generate summary statistics, filter data, and perform group-by operations.

Matplotlib:

Purpose: Matplotlib is a widely used library for creating static, animated, and interactive visualizations in Python.

Key Features:

Comprehensive plotting capabilities: Matplotlib provides a wide range of plot types, including line plots, scatter plots, bar charts, histograms, and more.

Customization: You can customize every aspect of your plots, including colors, labels, titles, and axes.

Support for multiple output formats: Matplotlib can generate plots as images, interactive plots for Jupyter Notebooks, or even embeddable plots for use in graphical user interfaces.

Seaborn:

Purpose: Seaborn is a statistical data visualization library that builds on top of Matplotlib. It is designed to create attractive and informative statistical graphics.

Key Features:

High-level interface: Seaborn simplifies the process of creating complex statistical visualizations by providing a high-level interface for common plot types like heatmaps, violin plots, and pair plots.

Integration with Pandas: Seaborn seamlessly integrates with Pandas DataFrames, making it easy to visualize Pandas data structures.

Aesthetic control: Seaborn allows you to easily customize plot aesthetics, including colors and themes, to produce visually appealing and informative charts.

In summary, Pandas is primarily used for data manipulation and preparation, Matplotlib is used for creating a wide range of static and interactive visualizations, and Seaborn is a specialized library for creating statistical and aesthetically pleasing visualizations. These libraries are often used together in data analysis workflows to explore and visualize data effectively.

pd.set\_option('display.max\_columns', None), is used in Python with the Pandas library to set a display option. Specifically, it sets the maximum number of columns to be displayed when printing DataFrames or Series objects to "None," which means that Pandas will display all columns without truncation.

Here's what each part of the code does:

pd: This assumes that you have imported the Pandas library using the common alias "pd."

set\_option(): This is a Pandas function used to set various options that control the behavior of Pandas in your Python environment.

'display.max\_columns': This is the specific option being set. In this case, it refers to the maximum number of columns to display in a DataFrame or Series.

None: Setting this option to None means that there is no maximum limit on the number of columns to display. As a result, when you print a DataFrame or Series, all columns will be displayed without truncation.

This can be useful when working with large datasets or DataFrames with many columns because it ensures that you can see all the data without any columns being omitted due to space limitations in the display.

However, it's important to use this option judiciously, as displaying a large number of columns can make the output harder to read and consume more screen space. It's often used during data exploration or debugging to ensure you see all the available data, but in final reports or presentations, you might want to limit the displayed columns for readability.

df = pd.read\_csv('Data/census\_2011.csv'): This line of code reads the CSV file 'census\_2011.csv' located in the 'Data' directory (relative to the current working directory) using the Pandas read\_csv() function. It loads the data from the CSV file into a Pandas DataFrame and assigns it to the variable df.

df.head(): After loading the data into the DataFrame, this line of code displays the first few rows of the DataFrame using the .head() method. By default, .head() displays the first 5 rows of the DataFrame. This is a common practice to quickly inspect the data and get a sense of its structure.

So, when you run this code, you'll see the first 5 rows of the 'census\_2011.csv' data displayed in your Jupyter Notebook or Python environment. This allows you to get an initial look at the data and understand its columns and values.

relevant\_columns: This is a list that contains the names of the columns you want to select from the original DataFrame df. These columns include information such as state name, district name, population, literacy rates, household counts, and age group statistics.

df\_relevant = df[relevant\_columns]: This line of code uses square brackets to select the columns specified in the relevant\_columns list from the original DataFrame df. It creates a new DataFrame called df\_relevant that includes only the selected columns.

df\_relevant: Finally, when you execute df\_relevant, it displays the new DataFrame that contains only the columns you specified. This allows you to work with a subset of the original data that is relevant to your analysis or visualization tasks, making it easier to focus on specific aspects of the dataset.

By creating df\_relevant, you can perform data analysis, visualization, or other operations on a more focused subset of the data without dealing with unnecessary columns.

column\_name\_mapping: This is a dictionary where the keys represent the current column names in df\_relevant, and the values represent the desired new column names. It specifies the column name mappings you want to apply.

df\_renamed = df\_relevant.rename(columns=column\_name\_mapping): This line of code uses the .rename() method to rename the columns of the DataFrame df\_relevant based on the mappings provided in the column\_name\_mapping dictionary. The columns parameter is set to column\_name\_mapping, which specifies how the columns should be renamed.

df\_renamed: Finally, when you execute df\_renamed, it displays the updated DataFrame with the new column names. The DataFrame now contains columns with the names specified in the column\_name\_mapping dictionary.

By renaming the columns in this way, you can make your DataFrame's column names more descriptive and easier to work with, which can improve the clarity and readability of your data analysis or visualization code.

def custom\_title\_case(name):

words = name.split()

title\_words = []

for word in words:

if word.lower() == "and":

title\_words.append(word.lower())

else:

title\_words.append(word.capitalize())

return " ".join(title\_words)

df\_renamed['State/UT'] = df\_renamed['State/UT'].apply(custom\_title\_case)

df\_renamed

custom\_title\_case(name): This is a custom function that takes a single argument name. Inside the function:

It splits the input string name into individual words using the .split() method, which separates the words based on spaces and creates a list of words.

It initializes an empty list title\_words to store the modified words.

It iterates through each word in the list of words:

If the word is "and" (case-insensitive), it appends it to the title\_words list in lowercase.

Otherwise, it capitalizes the word using .capitalize() and appends it to the title\_words list.

Finally, it joins the modified words in the title\_words list back into a single string with spaces between them using " ".join(title\_words) and returns the result.

df\_renamed['State/UT'].apply(custom\_title\_case): This line of code applies the custom\_title\_case function to each value in the 'State/UT' column of the DataFrame df\_renamed. It capitalizes words in the column except for the word "and," which remains in lowercase.

df\_renamed: When you execute df\_renamed, it displays the updated DataFrame with the 'State/UT' column values modified by the custom\_title\_case function. The column now contains title-cased state/UT names with "and" in lowercase.

This type of data transformation is useful for ensuring consistent formatting in text columns, especially when you want to apply specific capitalization rules or exclude certain words from capitalization.

telangana\_districts = []

with open('Data/Telangana.txt', 'r') as file:

telangana\_districts = [line.strip() for line in file]

df\_renamed.loc[df\_renamed['District'].isin(telangana\_districts), 'State/UT'] = 'Telangana'

df\_renamed

telangana\_districts = []: This line initializes an empty list called telangana\_districts to store the names of the districts in Telangana.

with open('Data/Telangana.txt', 'r') as file:: This line opens the 'Telangana.txt' file located in the 'Data' directory in read ('r') mode. The file is opened using a context manager, ensuring that it is automatically closed when the block is exited.

telangana\_districts = [line.strip() for line in file]: Inside the context manager, this line reads each line from the 'Telangana.txt' file, strips any leading or trailing whitespace using .strip(), and adds each line (which is assumed to be a district name) to the telangana\_districts list.

df\_renamed.loc[df\_renamed['District'].isin(telangana\_districts), 'State/UT'] = 'Telangana': This line of code locates rows in the DataFrame df\_renamed where the 'District' column matches any of the district names in the telangana\_districts list. For these rows, it updates the 'State/UT' column to 'Telangana', effectively assigning 'Telangana' as the state/UT name for these specific districts.

df\_renamed: When you execute df\_renamed, it displays the updated DataFrame with the 'State/UT' column modified to 'Telangana' for the districts listed in the 'Telangana.txt' file. This allows you to assign the correct state/UT name to specific districts based on the contents of the text file.

ladakh\_districts = ['Leh(Ladakh)', 'Kargil']

df\_renamed.loc[df\_renamed['District'].isin(ladakh\_districts), 'State/UT'] = 'Ladakh'

df\_renamed

ladakh\_districts = ['Leh(Ladakh)', 'Kargil']: This line defines a list called ladakh\_districts containing the names of districts that belong to the region of Ladakh. The district names are 'Leh(Ladakh)' and 'Kargil'.

df\_renamed.loc[df\_renamed['District'].isin(ladakh\_districts), 'State/UT'] = 'Ladakh': This line of code uses the .loc indexer to locate rows in the DataFrame df\_renamed where the 'District' column matches any of the district names in the ladakh\_districts list. For these rows, it updates the 'State/UT' column to 'Ladakh', effectively assigning 'Ladakh' as the state/UT name for these specific districts.

df\_renamed: When you execute df\_renamed, it displays the updated DataFrame with the 'State/UT' column modified to 'Ladakh' for the districts 'Leh(Ladakh)' and 'Kargil'. This allows you to assign the correct state/UT name to these specific districts.

op = df\_renamed['Population'].isnull().sum()

ol = df\_renamed['Literate'].isnull().sum()

op = df\_renamed['Population'].isnull().sum()

oh = df\_renamed['Households'].isnull().sum()

op = df\_renamed['Population'].isnull().sum(): This line calculates the number of missing values in the 'Population' column of the DataFrame df\_renamed. It uses the .isnull() method to create a Boolean mask where True indicates a missing value and False indicates a non-missing value. Then, .sum() is used to count the number of True values, which corresponds to the number of missing values in the 'Population' column. The result is stored in the variable op.

ol = df\_renamed['Literate'].isnull().sum(): This line calculates the number of missing values in the 'Literate' column of the DataFrame df\_renamed in a similar manner. It counts the missing values in the 'Literate' column and stores the count in the variable ol.

oh = df\_renamed['Households'].isnull().sum(): This line calculates the number of missing values in the 'Households' column of the DataFrame df\_renamed. It counts the missing values in the 'Households' column and stores the count in the variable oh.

After running these lines of code, you will have the following variables containing the counts of missing values:

op: Number of missing values in the 'Population' column.

ol: Number of missing values in the 'Literate' column.

oh: Number of missing values in the 'Households'

df\_renamed[df\_renamed['Population'].isnull()].index

df\_renamed.loc[df\_renamed['Literate'].isnull(), 'Literate'] = df\_renamed['Literate\_Male'] + df\_renamed['Literate\_Female']

df\_renamed.loc[df\_renamed['Population'].isnull(), 'Population'] = df\_renamed['Young\_and\_Adult'] + df\_renamed['Middle\_Aged'] + df\_renamed['Senior\_Citizen'] + df\_renamed['Age\_Not\_Stated']

df\_renamed.loc[df\_renamed['Households'].isnull(), 'Households'] = df\_renamed['Households\_Rural'] + df\_renamed['Households\_Urban']

np = df\_renamed['Population'].isnull().sum()

nl = df\_renamed['Literate'].isnull().sum()

np = df\_renamed['Population'].isnull().sum()

nh = df\_renamed['Households'].isnull().sum()

old\_missing\_values = [op, ol, op, oh]

new\_missing\_values = [np, nl, np, nh]

columns = ['Population', 'Literate', 'Population', 'Households']

plt.figure(figsize=(10, 6))

plt.bar(columns, old\_missing\_values, color='blue', label='Before Filling')

plt.bar(columns, new\_missing\_values, color='orange', label='After Filling')

plt.xlabel('Columns')

plt.ylabel('Number of Missing Values')

plt.title('Comparison of Missing Data Before and After Filling')

plt.legend()

plt.show()

df\_renamed[df\_renamed['Population'].isnull()].index: This line of code retrieves the indices of rows where the 'Population' column has missing values (null values). It identifies the rows with missing population data.

df\_renamed.loc[df\_renamed['Literate'].isnull(), 'Literate'] = df\_renamed['Literate\_Male'] + df\_renamed['Literate\_Female']: This line fills missing values in the 'Literate' column with the sum of 'Literate\_Male' and 'Literate\_Female' columns. It assumes that the total literacy can be calculated by adding male and female literacy counts.

df\_renamed.loc[df\_renamed['Population'].isnull(), 'Population'] = df\_renamed['Young\_and\_Adult'] + df\_renamed['Middle\_Aged'] + df\_renamed['Senior\_Citizen'] + df\_renamed['Age\_Not\_Stated']: This line fills missing values in the 'Population' column with the sum of 'Young\_and\_Adult', 'Middle\_Aged', 'Senior\_Citizen', and 'Age\_Not\_Stated' columns. It assumes that the total population can be estimated based on age groups and age groups with unspecified ages.

df\_renamed.loc[df\_renamed['Households'].isnull(), 'Households'] = df\_renamed['Households\_Rural'] + df\_renamed['Households\_Urban']: This line fills missing values in the 'Households' column with the sum of 'Households\_Rural' and 'Households\_Urban' columns. It assumes that the total number of households can be calculated by adding rural and urban households.

np = df\_renamed['Population'].isnull().sum(): This line calculates the number of missing values in the 'Population' column after filling and stores it in the variable np.

nl = df\_renamed['Literate'].isnull().sum(): This line calculates the number of missing values in the 'Literate' column after filling and stores it in the variable nl.

np = df\_renamed['Population'].isnull().sum(): This line calculates the number of missing values in the 'Population' column again (a duplicate) and stores it in the variable np.

nh = df\_renamed['Households'].isnull().sum(): This line calculates the number of missing values in the 'Households' column after filling and stores it in the variable nh.

old\_missing\_values and new\_missing\_values: These lists are created to store the counts of missing values before and after filling, respectively.

columns: This list stores the names of the columns ('Population', 'Literate', 'Population', 'Households') corresponding to the missing values.

The code then uses Matplotlib to create a bar chart comparing the number of missing values before and after filling for each column. It uses blue bars for missing values before filling and orange bars for missing values after filling. The chart is displayed with labels and a legend.

This chart provides a visual representation of how the filling of missing values has impacted the dataset.

census\_data = pd.read\_csv('census.csv')

housing\_columns = ['District Name', 'Rural/Urban', 'Total Number of households', 'Total Number of Livable', 'Total Number of Dilapidated', 'Latrine\_premise']

housing\_relevant = housing\_data[housing\_columns]

merged\_data = pd.merge(housing\_relevant, census\_data, left\_on='District Name', right\_on='District', how='inner')

merged\_data['Households\_Rural\_Livable'] = (merged\_data['Total Number of households'] \* merged\_data['Total Number of Livable']) / 100

merged\_data['Households\_Rural\_Dilapidated'] = (merged\_data['Total Number of households'] \* merged\_data['Total Number of Dilapidated']) / 100

merged\_data['Households\_Rural\_Toilet\_Premise'] = (merged\_data['Households\_Rural'] \* merged\_data['Latrine\_premise']) / 100

merged\_data['Households\_Urban\_Livable'] = (merged\_data['Total Number of households'] \* merged\_data['Total Number of Livable']) / 100

merged\_data['Households\_Urban\_Dilapidated'] = (merged\_data['Total Number of households'] \* merged\_data['Total Number of Dilapidated']) / 100

merged\_data['Households\_Urban\_Toilet\_Premise'] = (merged\_data['Households\_Urban'] \* merged\_data['Latrine\_premise']) / 100

column\_mapping = {

'District Name': 'District',

'Rural/Urban': 'Rural\_Urban',

'Households\_Rural\_Livable': 'Households\_Rural\_Livable',

'Households\_Rural\_Dilapidated': 'Households\_Rural\_Dilapidated',

'Households\_Rural\_Toilet\_Premise': 'Households\_Rural\_Toilet\_Premise',

'Households\_Urban\_Livable': 'Households\_Urban\_Livable',

'Households\_Urban\_Dilapidated': 'Households\_Urban\_Dilapidated',

'Households\_Urban\_Toilet\_Premise': 'Households\_Urban\_Toilet\_Premise'

}

merged\_data.rename(columns=column\_mapping, inplace=True)

merged\_data.to\_csv('housing.csv', index=False)

Reads data from two CSV files, 'census.csv' and 'housing.csv', into two separate DataFrames, census\_data and housing\_data, respectively.

Selects relevant columns from the housing\_data DataFrame using the housing\_columns list and stores the resulting DataFrame in housing\_relevant.

Merges the housing\_relevant DataFrame with the census\_data DataFrame based on the 'District Name' column from housing\_relevant and the 'District' column from census\_data. The merge is performed as an inner join, and the resulting merged data is stored in the merged\_data DataFrame.

Calculates new columns in merged\_data based on various combinations of existing columns. These new columns appear to represent different percentages of households, livable households, dilapidated households, and households with toilet premises for both rural and urban areas.

Renames the columns in merged\_data using the column\_mapping dictionary to provide more descriptive column names.

Writes the final merged\_data DataFrame to a new CSV file named 'housing.csv' without including the index column.

plt.figure(figsize=(10, 6))

plt.bar(merged\_data['Total Number of Livable'], merged\_data['Total Number of Dilapidated'])

plt.xlabel('Number of Households for 100 People')

plt.ylabel('District')

plt.title('Number of Households for 100 People')

plt.tight\_layout()

plt.show()

plt.figure(figsize=(10, 6)): This line sets the figure size for the plot, specifying a width of 10 units and a height of 6 units.

plt.bar(merged\_data['Total Number of Livable'], merged\_data['Total Number of Dilapidated']): This line creates a bar chart using the 'Total Number of Livable' column on the x-axis and the 'Total Number of Dilapidated' column on the y-axis. It effectively compares the number of livable and dilapidated households for each district.

plt.xlabel('Number of Households for 100 People'): This line sets the label for the x-axis to 'Number of Households for 100 People'.

plt.ylabel('District'): This line sets the label for the y-axis to 'District'.

plt.title('Number of Households for 100 People'): This line sets the title of the plot to 'Number of Households for 100 People'.

plt.tight\_layout(): This line ensures that the plot layout is adjusted to prevent overlapping of labels and elements.

plt.show(): This line displays the plot in the current Jupyter Notebook or Python environment.

merged\_data['new\_Toilet\_Premise\_Percentage'] = df\_renamed.Households/df\_renamed.Population \*100

df\_renamed['new'] = df\_renamed.Households/df\_renamed.Population \*100

plt.figure(figsize=(10, 6))

plt.bar(df\_renamed['District'], df\_renamed['new'])

plt.xlabel('Percentage of Households with Toilet Premise')

plt.ylabel('District')

plt.title('Percentage of Households with Toilet Premise')

plt.tight\_layout()

plt.show()

Two new columns, 'new\_Toilet\_Premise\_Percentage' and 'new', are added to the df\_renamed DataFrame to store the calculated percentages.

The bar chart is created with 'District' on the x-axis and the 'new' column (percentage of households with a toilet premise) on the y-axis.

Labels and titles are added to the plot for clarity.

temp = df\_renamed.groupby(['State/UT'])[['Population','Households']].sum().reset\_index()

temp['new'] = temp.Households/temp.Population \*100

plt.figure(figsize=(10, 6))

plt.bar(temp['State/UT'], temp['new'])

plt.xlabel('Percentage of Households with Toilet Premise')

plt.ylabel('District')

plt.title('Percentage of Households with Toilet Premise')

plt.xticks(rotation=90)

plt.tight\_layout()

plt.show()

The groupby operation groups the data by 'State/UT' and calculates the total population and total households for each state/UT.

The 'new' column is added to store the calculated percentages.

The bar chart displays 'State/UT' on the x-axis and the 'new' column (percentage of households with a toilet premise) on the y-axis. The rotation=90 parameter rotates the x-axis labels for better readability.

temp = merged\_data.groupby(['State/UT'])[['Households\_Urban\_Toilet\_Premise','Households\_Rural\_Toilet\_Premise','Households']].sum().reset\_index()

temp['new'] = round(temp['Households\_Urban\_Toilet\_Premise']+temp['Households\_Rural\_Toilet\_Premise'],2)

temp['new1'] = temp.new/temp.Households \*100

plt.figure(figsize=(10, 6))

plt.bar(temp['State/UT'], temp['new1'])

plt.xlabel('Percentage of Households with Toilet Premise')

plt.ylabel('District')

plt.title('Percentage of Households with Toilet Premise')

plt.xticks(rotation=90)

plt.tight\_layout()

plt.show()

The groupby operation groups the data by 'State/UT' and calculates the total urban toilet premises, rural toilet premises, and total households for each state/UT.

The 'new' column stores the calculated combined toilet premises percentage.

The 'new1' column calculates the final percentage of households with a toilet premise by considering both urban and rural areas.

The bar chart displays 'State/UT' on the x-axis and the 'new1' column (percentage of households with a toilet premise) on the y-axis. The rotation=90 parameter rotates the x-axis labels for better readability.

temp['Rural'] = temp.Households\_Rural/temp.Households \*100

temp['Urban'] = temp.Households\_Urban/temp.Households \*100

plt.figure(figsize=(14, 6))

plt.bar(temp.index, temp['Rural'], bar\_width, label='Rural')

plt.bar(temp.index + bar\_width, temp['Urban'], bar\_width, label='Urban')

plt.xlabel('State/UT')

plt.ylabel('Percentage')

plt.title('Rural and Urban Population Percentage')

plt.xticks(temp.index + bar\_width/2, temp['State/UT'], rotation=90)

plt.legend()

plt.tight\_layout()

plt.show()

temp['Rural'] = temp.Households\_Rural/temp.Households \*100: This line calculates the percentage of the rural population by dividing the 'Households\_Rural' column by the 'Households' column and then multiplying by 100. The result is stored in a new 'Rural' column in the temp DataFrame.

temp['Urban'] = temp.Households\_Urban/temp.Households \*100: Similarly, this line calculates the percentage of the urban population by dividing the 'Households\_Urban' column by the 'Households' column and then multiplying by 100. The result is stored in a new 'Urban' column in the temp DataFrame.

plt.figure(figsize=(14, 6)): This line sets the figure size for the plot, specifying a width of 14 units and a height of 6 units.

plt.bar(temp.index, temp['Rural'], bar\_width, label='Rural'): The first plt.bar() call creates bars for the rural population percentages. temp.index represents the x-positions of the bars, temp['Rural'] provides the heights of the bars, and bar\_width is used to specify the width of each bar. The label 'Rural' is added for the legend.

plt.bar(temp.index + bar\_width, temp['Urban'], bar\_width, label='Urban'): The second plt.bar() call creates bars for the urban population percentages. By adding bar\_width to the x-positions, the urban bars are placed side-by-side with the rural bars. The label 'Urban' is added for the legend.

plt.xlabel('State/UT'): This line sets the label for the x-axis to 'State/UT'.

plt.ylabel('Percentage'): This line sets the label for the y-axis to 'Percentage'.

plt.title('Rural and Urban Population Percentage'): This line sets the title of the plot to 'Rural and Urban Population Percentage'.

plt.xticks(temp.index + bar\_width/2, temp['State/UT'], rotation=90): This line sets the x-axis ticks to the middle of each pair of bars and labels them with the 'State/UT' values from the temp DataFrame. The rotation=90 parameter rotates the x-axis labels for better readability.

plt.legend(): This line displays the legend to distinguish between the 'Rural' and 'Urban' bars.

plt.tight\_layout(): This line ensures that the plot layout is adjusted to prevent overlapping of labels and elements.

plt.show(): Finally, this line displays the grouped bar chart.

census\_data = pd.read\_csv('census.csv')

housing\_data = pd.read\_csv('housing.csv')

census\_data['Households\_Rural\_Diff'] = ((census\_data['Households\_Rural'] - housing\_data['Households\_Rural']) / census\_data['Households\_Rural']) \* 100

census\_data['Households\_Urban\_Diff'] = ((census\_data['Households\_Urban'] - housing\_data['Households\_Urban']) / census\_data['Households\_Urban']) \* 100

major\_diff\_districts = census\_data[

(abs(census\_data['Households\_Rural\_Diff']) > 10) |

(abs(census\_data['Households\_Urban\_Diff']) > 10)

]

temp = abs(major\_diff\_districts.groupby('State/UT')[['Households', 'Households\_Rural\_Diff', 'Households\_Urban\_Diff']].sum()).reset\_index()

temp['Rural'] = temp.Households\_Rural\_Diff/temp.Households \*100

temp['Urban'] = temp.Households\_Urban\_Diff/temp.Households \*100

major\_diff\_districts = temp[

(abs(temp['Rural']) > 10) |

(abs(temp['Urban']) > 10)

]

Reads two CSV files, 'census.csv' and 'housing.csv', into DataFrames named census\_data and housing\_data, respectively.

Calculates the percentage difference between the number of rural households in the 'census\_data' and 'housing\_data' DataFrames, as well as the percentage difference for urban households. These differences are calculated based on the formula:

((census\_data['Households\_Rural'] - housing\_data['Households\_Rural']) / census\_data['Households\_Rural']) \* 100

and similarly for urban households.

Identifies major difference districts by filtering the rows in census\_data where either the absolute difference in rural households or the absolute difference in urban households is greater than 10%. The resulting DataFrame is stored in major\_diff\_districts.

Groups major\_diff\_districts by 'State/UT' and calculates the sum of 'Households', 'Households\_Rural\_Diff', and 'Households\_Urban\_Diff' for each state/UT. The result is stored in the temp DataFrame.

Calculates the percentage difference for rural and urban households at the state/UT level and stores these percentages in the 'Rural' and 'Urban' columns of the temp DataFrame.

Reassigns major\_diff\_districts to only include rows where either the absolute 'Rural' percentage difference or the absolute 'Urban' percentage difference is greater than 10%. This further filters the data to focus on state/UTs with significant differences in household data.