ADVANCED DBMS ASSIGNMENT

	NAMES	REG NO
GROUP MEMBERS:	**DIANA CHELAGAT**	**SCT222-0142/2020**
	CHERRLY MUKHISA	**SCT222-0141/2019**
	LILIANROSE WANJIKU	**SCT222-0302/2020**
	EMILE	**SCT222-0671/2018**

Data Collection

!mkdir diana

The code **!mkdir diana** is a command often used in command-line interfaces, such as the terminal in Unix-based systems or the command prompt in Windows.

Here's what this specific command does:

- **!**: In many command-line interfaces, the exclamation mark (**!**) is used to run shell commands from within an environment that supports both shell commands and programming code. This can vary depending on the specific environment.
- **mkdir**: This is a command that stands for "make directory." It is used to create a new directory (folder) in the file system.

- **diana**: This is the name of the directory (folder) that the **mkdir** command is instructed to create. In this case, it's creating a directory named "diana."

When one runs **!mkdir diana**, it will create a new directory named "diana" in the current working directory of the command-line interface.

For example, if you are using a terminal and you run this command, you will see a new directory called "diana" created in the same location where you executed the command.

import os

os.chdir('diana')

The provided code snippet uses the **os** module in Python to change the current working directory to a directory named "diana." Below is a breakdown of each line:

1. **import os**: This line imports the **os** module in Python. The **os** module provides a way to interact with the operating system, allowing you to perform various file and directory operations.
2. **os.chdir('diana')**: This line uses the **chdir** function from the **os** module to change the current working directory. The argument passed to **chdir** is the path of the directory to which you want to change. In this case, it's 'diana'.
    - **chdir**: Stands for "change directory."
    - **'diana'**: The argument passed to **chdir** is the name of the directory to which the current working directory should be changed.

After running this code, the current working directory for the Python script or session will be set to the "diana" directory. Any subsequent file or directory operations will be performed relative to this new working directory.

It's important to note that the "diana" directory should exist for this operation to be successful; otherwise, an error might occur.

url = 'https://covid19.who.int/WHO-COVID-19-global-data.csv' !wget -O covid.csv 'https://covid19.who.int/WHO-COVID-19-global-data.csv'

This code snippet is using the **wget** . The purpose of this code is to download a CSV file containing global COVID-19 data from the World Health Organization (WHO) website.

Let's break down the code:

1. **url = 'https://covid19.who.int/WHO-COVID-19-global-data.csv'**: This line defines a variable named **url** and assigns it the value of the WHO's COVID-19 global data CSV file URL.
2. **!wget -O covid.csv 'https://covid19.who.int/WHO-COVID-19-global-data.csv'**: This line uses the **wget** command, a command-line utility for downloading files from the web. Here's a breakdown of the command:
    - **!**: In many interactive computing environments, the exclamation mark (**!**) is used to run shell commands.
    - **wget**: The **wget** command itself, used for downloading files from the internet.
    - **-O covid.csv**: Specifies the option to rename the downloaded file to "covid.csv." The **-O** option is followed by the desired output file name.
    - **'https://covid19.who.int/WHO-COVID-19-global-data.csv'**: The URL of the file to be downloaded. This is the same URL stored in the **url** variable.

After running this code, the **wget** command will download the WHO's COVID-19 global data CSV file from the specified URL and save it with the name "covid.csv" in the current working directory.

The data in the CSV file likely contains information about COVID-19 cases, deaths, and other related statistics on a global scale.

import pandas as pd

df1 = pd.read_csv('covid.csv')

This code snippet uses the pandas library in Python to read a CSV (Comma-Separated Values) file named 'covid.csv' and store its contents in a DataFrame. Here's a breakdown of each line:

1. **import pandas as pd**: This line imports the pandas library and assigns it the alias 'pd'. This alias is a common convention and makes it more convenient to refer to pandas functions and objects.
2. **df1 = pd.read_csv('covid.csv')**: This line uses the **read_csv** function from pandas to read the contents of a CSV file and create a DataFrame. Here's what each part of this line does:
    - **pd.read_csv**: Calls the **read_csv** function provided by the pandas library. This function is specifically designed to read data from CSV files and create a DataFrame.

- **'covid.csv'**: The argument passed to **read_csv** is the path to the CSV file to be read. In this case, it's assumed that 'covid.csv' is in the current working directory.
- **df1**: The result of the **read_csv** operation is assigned to the variable **df1**. This variable is now a pandas DataFrame, which is a two-dimensional labeled data structure with columns that can be of different types (e.g., integers, strings, etc.).

After running this code, you can use the DataFrame **df1** to analyze and manipulate the data contained in the 'covid.csv' file. Pandas provides various functionalities to work with tabular data, allowing you to perform tasks such as filtering, grouping, and aggregating data.

Pre-processing

df1.columns

The **df1.columns** expression is used to retrieve the column labels (names) of a pandas DataFrame. When you read a CSV file or create a DataFrame in pandas, the data is organized into rows and columns, and each column has a label or name associated with it.

Below is what **df1.columns** does:

- **df1**: This is the pandas DataFrame that you created earlier using **pd.read_csv('covid.csv')**.
- **columns**: This is an attribute of a pandas DataFrame that represents the column labels.

When one execute **df1.columns**, it returns an Index object containing the names of all the columns in the DataFrame **df1**. You can use this information to understand the structure of your DataFrame and refer to specific columns by their names.

For example, if you print the result of **df1.columns**, you'll see something like:

Index(['Column1', 'Column2', 'Column3', ...], dtype='object')

Replace 'Column1', 'Column2', 'Column3', etc., with the actual names of the columns in your DataFrame. This output shows the names of all the columns in the order they appear in the DataFrame.

features=['Cumulative_cases','Cumulative_deaths','Country_code']

df1=df1[features]

The provided code is selecting a subset of columns from a pandas DataFrame **df1** and creating a new DataFrame with only those selected columns. Let's break down each line:

1. **features=['Cumulative_cases','Cumulative_deaths','Country_code']**: This line creates a list named **features** containing the names of three columns: 'Cumulative_cases',

'Cumulative_deaths', and 'Country_code'. These columns are the features of interest that you want to retain in your DataFrame.

2. **df1=df1[features]**: This line uses the square bracket notation to select only the columns specified in the **features** list from the original DataFrame **df1**. The result is a new DataFrame (also named **df1** in this case) that contains only the selected columns.

After running this code, **df1** will be a DataFrame that includes only the columns 'Cumulative_cases', 'Cumulative_deaths', and 'Country_code'. The other columns from the original DataFrame are excluded.

This kind of column selection is useful when you want to focus on specific columns of interest for analysis or visualization and disregard the rest of the columns in the original DataFrame. It helps in working with a more compact and relevant subset of the data.

df1 = df1.rename(columns={"Cumulative_cases":"total","Cumulative_deaths":"deaths"})

df1

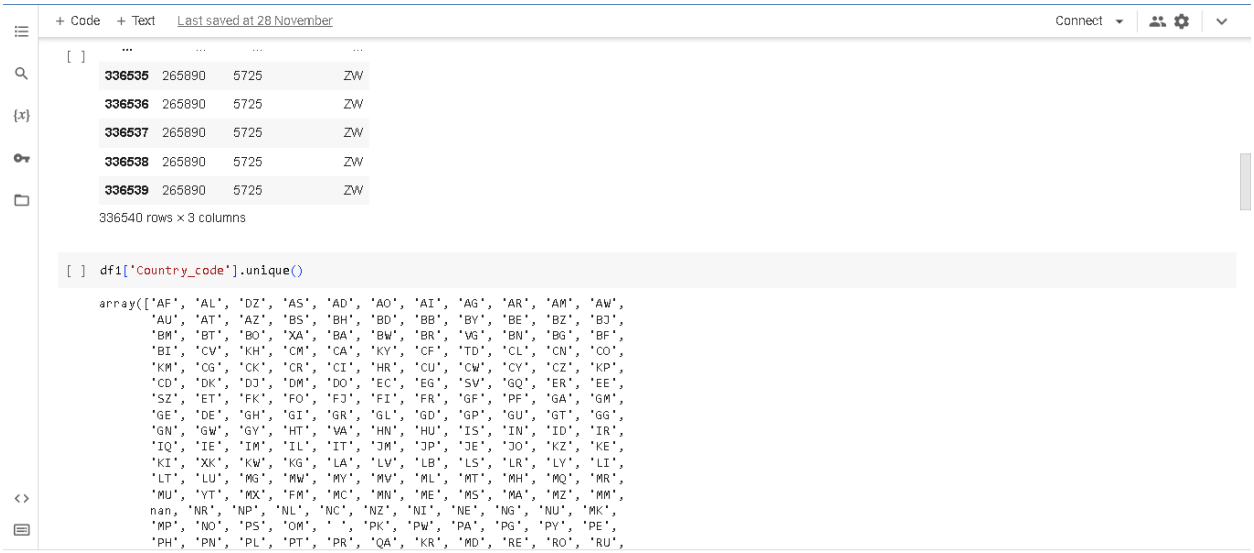
The provided code is renaming specific columns in a pandas DataFrame **df1**. Let's break down each line:

1. **df1 = df1.rename(columns={"Cumulative_cases":"total","Cumulative_deaths":"deaths"})**: This line uses the **rename** method of the DataFrame to rename specific columns. The argument **columns={"Cumulative_cases":"total","Cumulative_deaths":"deaths"}** is a dictionary where keys are the current column names, and values are the new column names. In this case, it renames the 'Cumulative_cases' column to 'total' and the 'Cumulative_deaths' column to 'deaths'.

2. **df1**: The result of the renaming operation is assigned back to the variable **df1**. This means that the original DataFrame is updated with the new column names.

After running this code, **df1** will have updated column names. If you print **df1**, you will see the DataFrame with columns 'total', 'deaths', and 'Country_code', assuming 'Country_code' was present in the original DataFrame.

This kind of column renaming is often done to make the column names more concise, descriptive, or consistent with a particular naming convention. It can improve code readability and align with the analysis or visualization tasks being performed on the DataFrame.



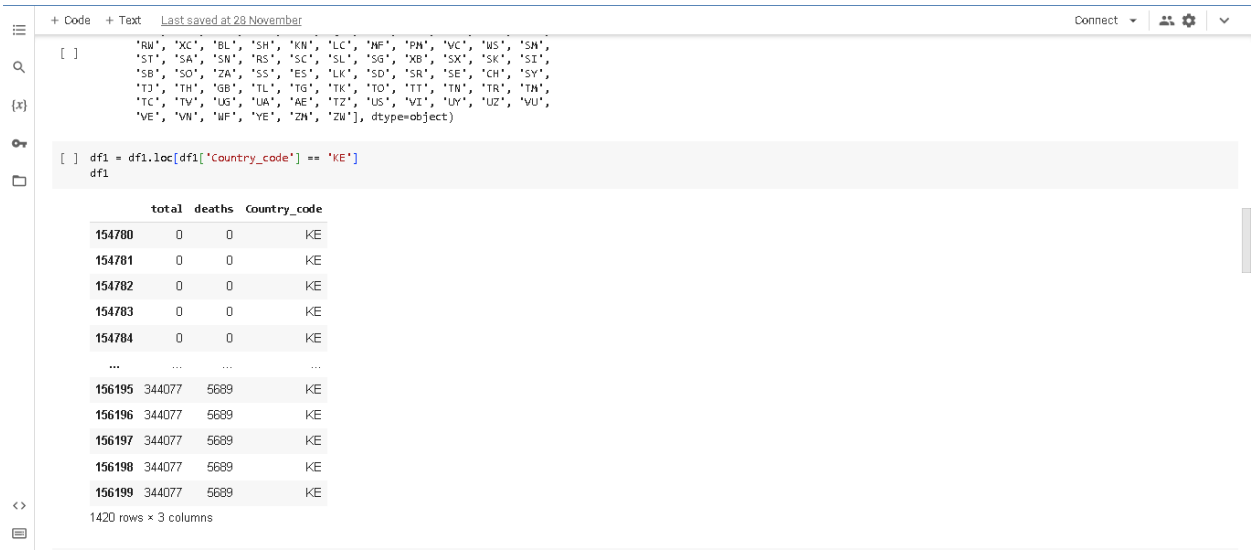
df1['Country_code'].unique()

The code **df1['Country_code'].unique()** is used to obtain an array or list of unique values in the 'Country_code' column of the pandas DataFrame **df1**.Below is a break down of the code:

- **df1['Country_code']**: This part of the code extracts the 'Country_code' column from the DataFrame **df1**. The result is a pandas Series, which is essentially a one-dimensional array with labeled indices.
- **.unique()**: This is a method provided by pandas for Series objects. When applied to a Series, it returns an array containing unique values present in that Series.

When you execute **df1['Country_code'].unique()**, it will return an array containing all the unique country codes present in the 'Country_code' column of the DataFrame **df1**. Each element in this array corresponds to a unique country code.

For example, if the 'Country_code' column contains country codes like 'US', 'CA', 'UK', and so on, the result might look like **array(['US', 'CA', 'UK', ...])**. This information is useful when you want to understand the distinct values in a categorical column or when you need to perform operations based on unique identifiers in your dataset.



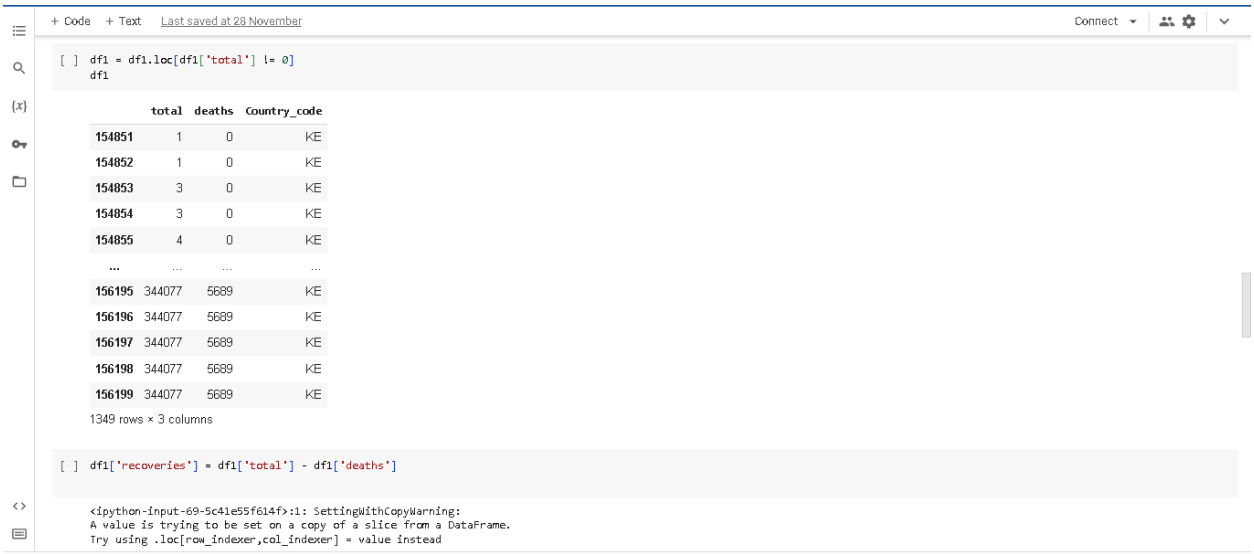
df1 = df1.loc[df1['Country_code'] == 'KE']

df1

The provided code filters the pandas DataFrame **df1** to retain only the rows where the value in the 'Country_code' column is equal to 'KE' (representing Kenya). Below is a breakdown of the code:

1. **df1.loc[df1['Country_code'] == 'KE']**: This line uses the **loc** indexer, which is used for label-based indexing, to select rows based on a condition. The condition here is **df1['Country_code'] == 'KE'**, which checks whether the value in the 'Country_code' column is equal to 'KE'.
    - **df1['Country_code'] == 'KE'**: This creates a boolean mask where each element is **True** if the corresponding 'Country_code' is 'KE' and **False** otherwise.
    - **df1.loc[...]**: The **loc** indexer is then used to select the rows where the condition is **True**. Only rows with 'Country_code' equal to 'KE' will be included in the resulting DataFrame.
2. **df1**: The result of the filtering operation is assigned back to the variable **df1**. This means that **df1** now contains only the rows where the 'Country_code' is 'KE'.

After running this code, **df1** will be a DataFrame containing data only for Kenya. This type of operation is useful when you want to focus your analysis on specific subsets of the data, such as information related to a particular country in this case.



df1 = df1.loc[df1['total'] != 0]

df1

The provided code filters the pandas DataFrame **df1** to retain only the rows where the value in the 'total' column is not equal to zero. Below is break down the code:

1. **df1.loc[df1['total'] != 0]**: This line uses the **loc** indexer to select rows based on a condition. The condition here is **df1['total'] != 0**, which checks whether the value in the 'total' column is not equal to zero.
    - **df1['total'] != 0**: This creates a boolean mask where each element is **True** if the corresponding 'total' value is not equal to zero and **False** otherwise.
    - **df1.loc[...]**: The **loc** indexer is then used to select the rows where the condition is **True**. Only rows with 'total' values not equal to zero will be included in the resulting DataFrame.
2. **df1**: The result of the filtering operation is assigned back to the variable **df1**. This means that **df1** now contains only the rows where the 'total' column is not equal to zero.

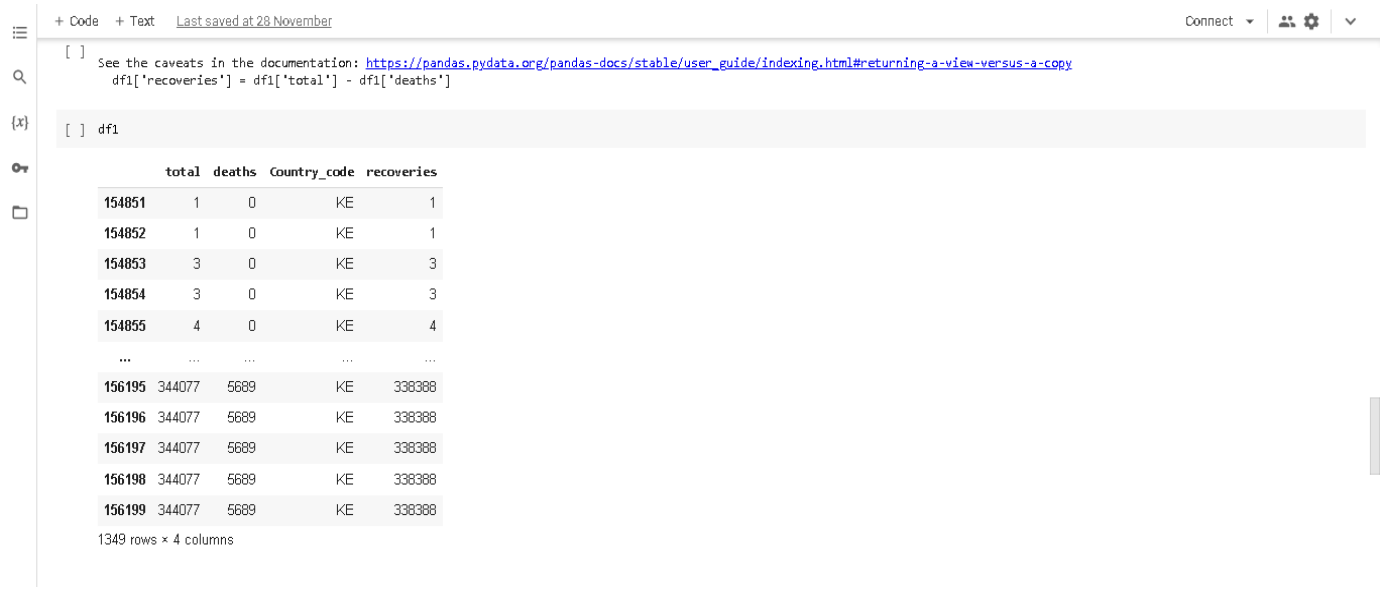
After running this code, **df1** will be a DataFrame containing data where the 'total' column has non-zero values. This type of operation is often performed to exclude rows with certain characteristics that might not be relevant to the analysis, such as rows with zero total cases in this case.

df1['recoveries'] = df1['total'] - df1['deaths']

The provided code creates a new column named 'recoveries' in the pandas DataFrame df1 and populates it with values calculated based on the existing 'total' and 'deaths' columns. Below is a breakdown of the code:

1. df1['recoveries']: This part of the code specifies the creation of a new column named 'recoveries' in the DataFrame df1. If the column already exists, it will be overwritten with the new values.
2. df1['total'] - df1['deaths']: This expression calculates the difference between the values in the 'total' column and the 'deaths' column for each row in the DataFrame. The result is a pandas Series containing the calculated differences.
    - If 'total' represents the total number of cases, and 'deaths' represents the total number of deaths, then the expression calculates the number of recoveries for each row.
3. df1['recoveries'] = ...: This assigns the calculated differences to the 'recoveries' column in the DataFrame. Each row in the 'recoveries' column will represent the calculated number of recoveries for the corresponding row based on the 'total' and 'deaths' values.

After running this code, the DataFrame df1 will have a new column 'recoveries' containing the calculated values representing the difference between the total cases and total deaths. This operation is often performed to derive additional information from existing columns in a DataFrame and is useful for further analysis or visualization.



	total	deaths	Country_code	recoveries
154851	1	0	KE	1
154852	1	0	KE	1
154853	3	0	KE	3
154854	3	0	KE	3
154855	4	0	KE	4
...	...	...	...	...
156195	344077	5689	KE	338388
156196	344077	5689	KE	338388
156197	344077	5689	KE	338388
156198	344077	5689	KE	338388
156199	344077	5689	KE	338388

1349 rows × 4 columns

Model Work

from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

from matplotlib import pyplot as plt

This code snippet imports three modules from popular Python libraries:

1. **from sklearn.linear_model import LinearRegression**: This imports the **LinearRegression** class from scikit-learn, which is a machine learning library in Python. The **LinearRegression** class is used to create a linear regression model, a type of supervised learning model that aims to establish a linear relationship between input features and a target variable.
2. **from sklearn.model_selection import train_test_split**: This imports the **train_test_split** function from scikit-learn. This function is commonly used for splitting datasets into training and testing sets. It helps in assessing the performance of machine learning models by training them on one subset of the data and testing their performance on another independent subset.
3. **from matplotlib import pyplot as plt**: This imports the **pyplot** module from the matplotlib library, which is a plotting library in Python. The **pyplot** module provides a convenient interface for creating various types of plots and visualizations.

Together, these imports suggest that the code is likely to involve linear regression modeling using scikit-learn and may include data splitting for training and testing sets. The **matplotlib** library may be used for visualizing the results of the linear regression model or displaying relevant plots.

In a typical machine learning workflow, you might expect to see further code that involves loading a dataset, preprocessing the data, splitting it into training and testing sets, creating and training a linear regression model, and evaluating its performance.

x_values = df1.recoveries

y_values = df1.deaths

X_train,X_test,y_train,y_test = train_test_split(x_values,y_values)

This code is preparing data for a machine learning task using scikit-learn's **train_test_split** function. Specifically, it looks like it's dealing with variables related to COVID-19 data (recoveries and deaths) from the DataFrame **df1**. Let's break down the code:

1. **x_values = df1.recoveries**: This line extracts the 'recoveries' column from the DataFrame **df1** and assigns it to the variable **x_values**. The 'recoveries' column is likely representing the independent variable or feature.
2. **y_values = df1.deaths**: This line extracts the 'deaths' column from the DataFrame **df1** and assigns it to the variable **y_values**. The 'deaths' column is likely representing the dependent variable or target.
3. **X_train, X_test, y_train, y_test = train_test_split(x_values, y_values)**: This line uses the **train_test_split** function from scikit-learn to split the data into training and testing sets for both the independent variable (**x_values** or 'recoveries') and the dependent variable (**y_values** or 'deaths'). The resulting variables are:
    - **X_train**: This will contain the training data for the independent variable.
    - **X_test**: This will contain the testing data for the independent variable.
    - **y_train**: This will contain the training data for the dependent variable.
    - **y_test**: This will contain the testing data for the dependent variable.

By default, **train_test_split** shuffles the data before splitting it, and it typically allocates 75% of the data for training and 25% for testing. However, the exact proportions can be adjusted using additional parameters of the function.

These training and testing sets are commonly used in machine learning to train a model on one subset of the data and evaluate its performance on another independent subset, helping to assess how well the model generalizes to new, unseen data.

model = LinearRegression()

model.fit(X_train.values.reshape(-1,1),y_train.values)

This code involves creating and training a linear regression model using scikit-learn's **LinearRegression** class. Here's a breakdown of the code:

1. **model = LinearRegression()**: This line creates an instance of the **LinearRegression** class and assigns it to the variable **model**. The **LinearRegression** class represents a linear regression model, which is a type of supervised learning model that aims to establish a linear relationship between input features and a target variable.

2. **model.fit(X_train.values.reshape(-1,1), y_train.values)**: This line trains the linear regression model using the training data. Here's what each part of this line does:
    - **model.fit(...)**: This method is used to train the linear regression model. The arguments passed to this method are the features (**X_train.values.reshape(-1,1)**) and the target variable (**y_train.values**).
    - **X_train.values.reshape(-1,1)**: This part reshapes the training features (**X_train**) to have a single feature per sample. Linear regression models in scikit-learn expect input features to be in a 2D array, and if you have a single feature, it should be reshaped to (-1, 1).
    - **y_train.values**: This represents the target variable, which is the variable the model is trying to predict.

After running this code, the variable **model** will be a trained linear regression model. The training process involves finding the optimal values for the model's parameters (coefficients) to best fit the training data and predict the target variable. This trained model can then be used to make predictions on new data.

prediction = model.predict(X_test.values.reshape(-1,1))

plt.plot(X_test,prediction,label='Linear Regression',color='b')

plt.scatter(X_test,y_test,label='Actual Data',color='g',alpha=.7)

plt.legend()

plt.show()

This code involves making predictions using a trained linear regression model and visualizing the results using matplotlib. Below is the breakdown of the code:

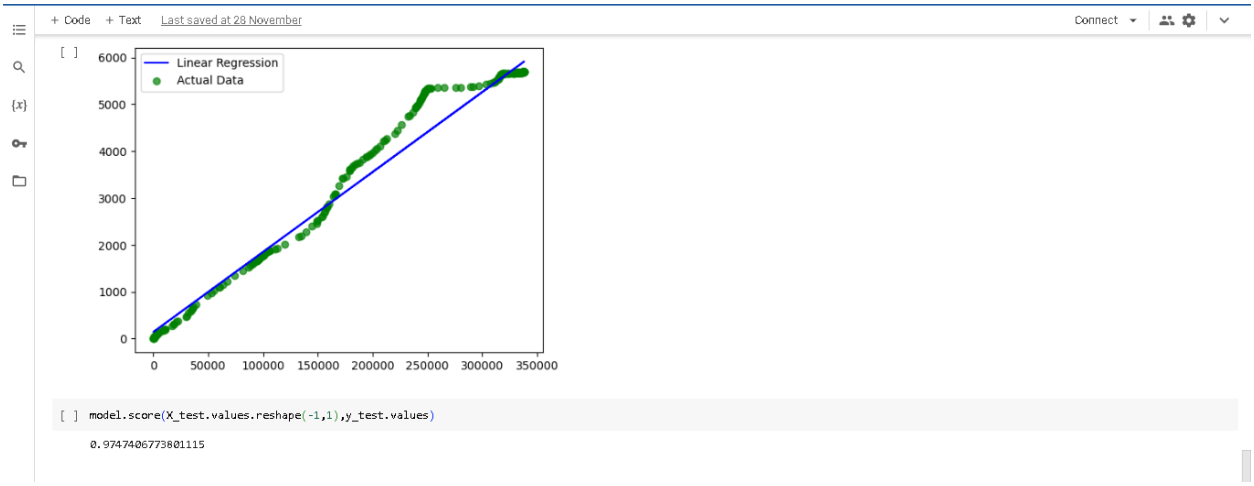
1. **prediction = model.predict(X_test.values.reshape(-1,1))**: This line uses the trained linear regression model (**model**) to make predictions on the test data (**X_test**). The input features are reshaped to have a single feature per sample using **X_test.values.reshape(-1,1)** to match the format expected by the model. The predicted values are then stored in the variable **prediction**.
2. **plt.plot(X_test, prediction, label='Linear Regression', color='b')**: This line creates a line plot using matplotlib (**plt**). It plots the predicted values (**prediction**) against the test data (**X_test**). The label 'Linear Regression' is assigned to this plot, and the color is set to blue ('b').
3. **plt.scatter(X_test, y_test, label='Actual Data', color='g', alpha=.7)**: This line adds a scatter plot to the same figure, showing the actual data points from the test set. The label 'Actual Data' is assigned to this scatter plot, and the color is set to green ('g'). The **alpha** parameter controls the transparency of the points, with a value of 0.7 making them slightly transparent.
4. **plt.legend()**: This line adds a legend to the plot, incorporating the labels provided in the previous lines.

5. **plt.show()**: This line displays the plot.

Overall, this code is creating a visual representation of the linear regression model's predictions on the test data, comparing them with the actual data points.

The line plot represents the predicted values, and the scatter plot represents the actual data. The legend helps distinguish between the two.

This kind of visualization is common in regression analysis to assess how well the model predictions align with the true values.



model.score(X_test.values.reshape(-1,1),y_test.values)

The code **model.score(X_test.values.reshape(-1,1), y_test.values)** calculates the coefficient of determination, commonly known as the R-squared score, for the linear regression model. Let's break down the code:

- **model.score(...)**: This method is part of the scikit-learn **LinearRegression** class. It is used to calculate the R-squared score, which is a measure of how well the linear regression model explains the variance in the target variable.
- **X_test.values.reshape(-1,1)**: This part reshapes the test features (**X_test**) to have a single feature per sample, matching the format expected by the model.
- **y_test.values**: This represents the true target variable values from the test set.

The R-squared score is a value between 0 and 1, with 1 indicating a perfect fit where the model explains all the variability in the target variable, and 0 indicating that the model does not explain any variability. A higher R-squared score generally suggests a better fit of the model to the data.

The returned score is the coefficient of determination, and it represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

When you run **model.score(X_test.values.reshape(-1,1), y_test.values)**, it gives you an indication of how well your linear regression model performs on the test data in terms of explaining the variance in the target variable.