**Python Coding Challenge**

**Name : Podutur Lahari - DE126**

**Date : 15-11-2024**

**1. Printing rows of the data :**

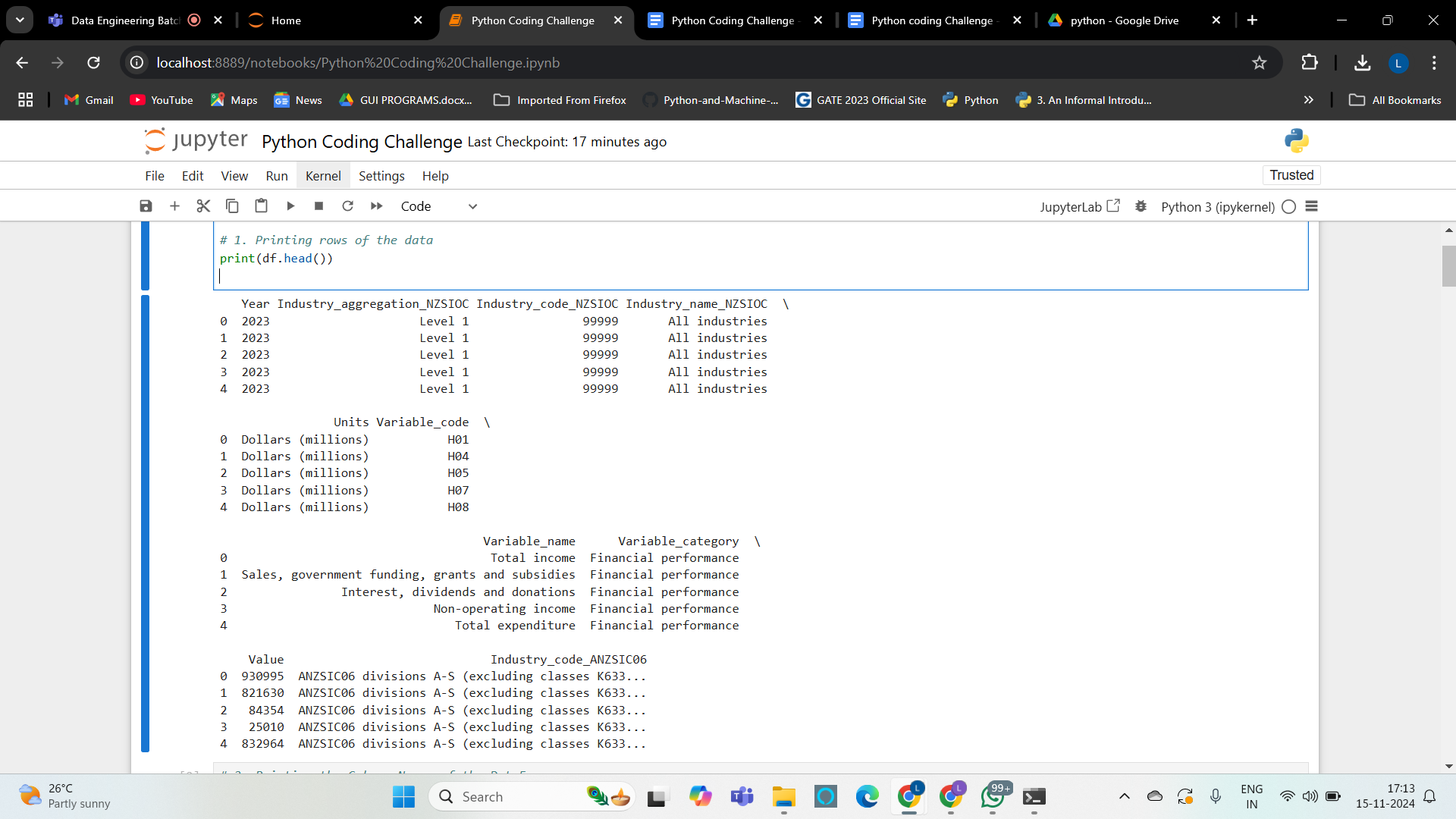
* The **head()** function displays the first five rows by default.
* This allows a quick look at the data structure and the contents of the dataset.
* We can pass an integer n to view the first n rows as needed.

**Code :**

import pandas as pd

df = pd.read\_csv("your\_dataset.csv")

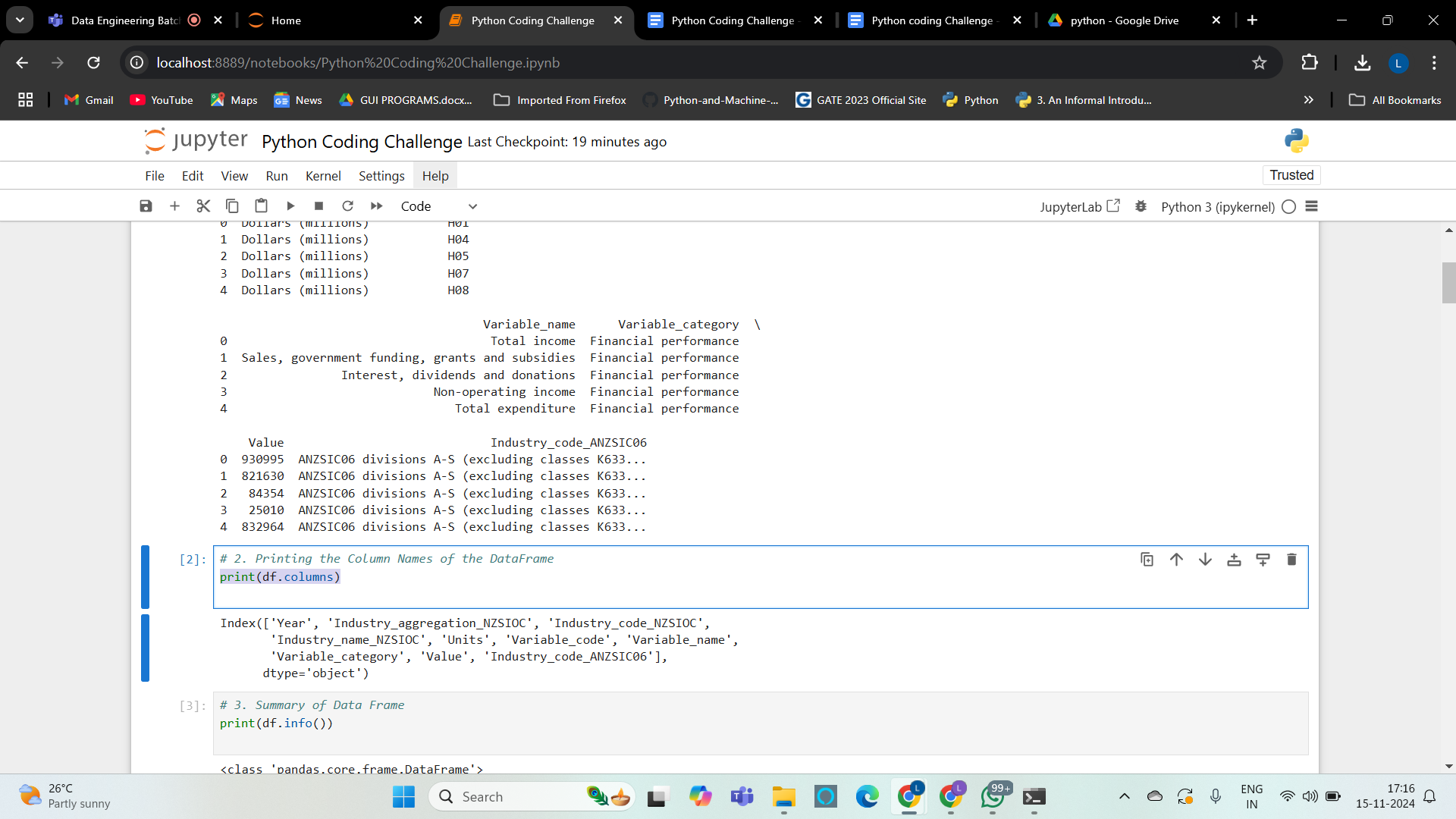
print(df.head()) # or df.head(n) for first n rows



**2. Printing the Column Names of the DataFrame :**

* **df.columns** returns an Index object containing column names.
* We can view and verify the correct loading of columns.
* This is especially useful for understanding the available fields.

**Code :**

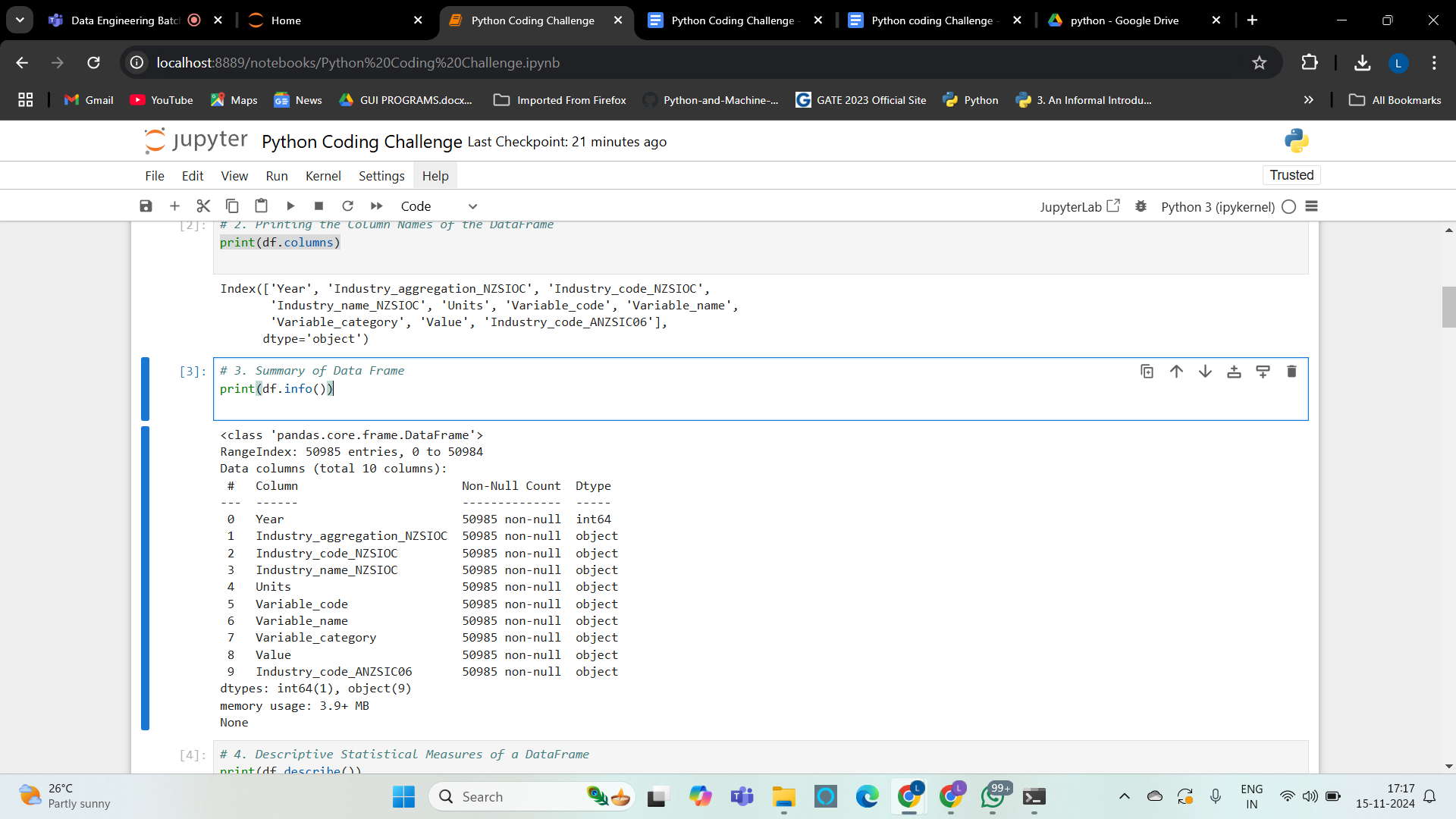
print(df.columns)

**3. Summary of Data Frame :**

* **df.info()** provides information about data types and null values for each column.
* It’s useful for assessing memory usage and data completeness.
* Essential for understanding the data format (numeric, object, etc.) and preparing for further analysis.

**Code :**

print(df.info())

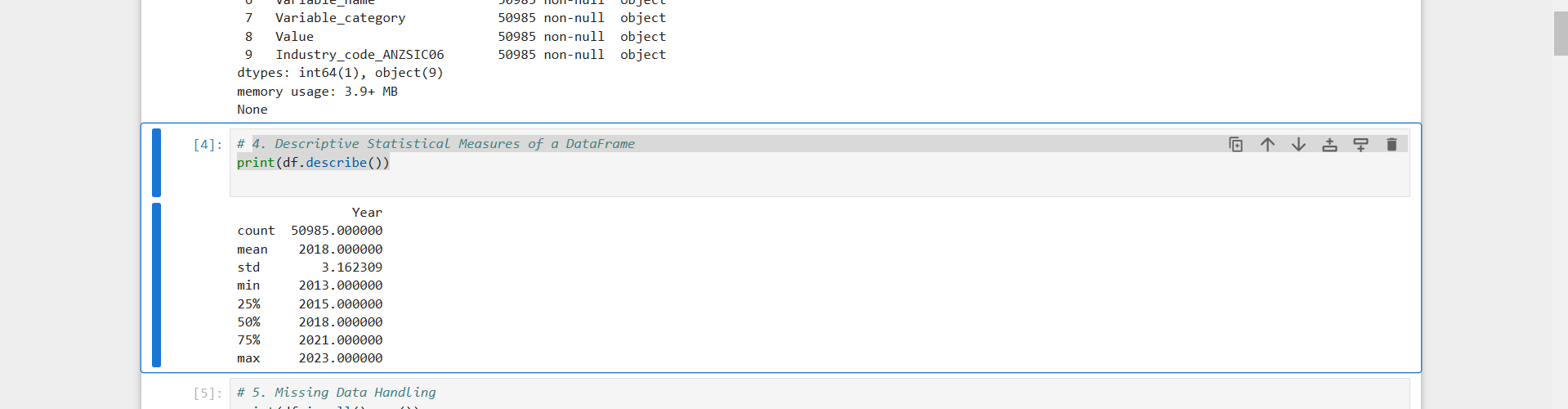


**4. Descriptive Statistical Measures of a DataFrame :**

* **describe()** calculates key statistical metrics (mean, std, min, etc.) for each numerical column.
* Helps in understanding data distribution, outliers, and central tendency.
* Useful for identifying any potential anomalies.

**Code :**

print(df.describe())



**5. Missing Data Handling :**

* **isnull().sum()** gives a count of missing values per column.
* We can handle missing data by using **fillna()** to replace NaNs or **dropna()** to remove rows with missing values.
* This ensures that the data is complete for analysis or avoids errors during computation.

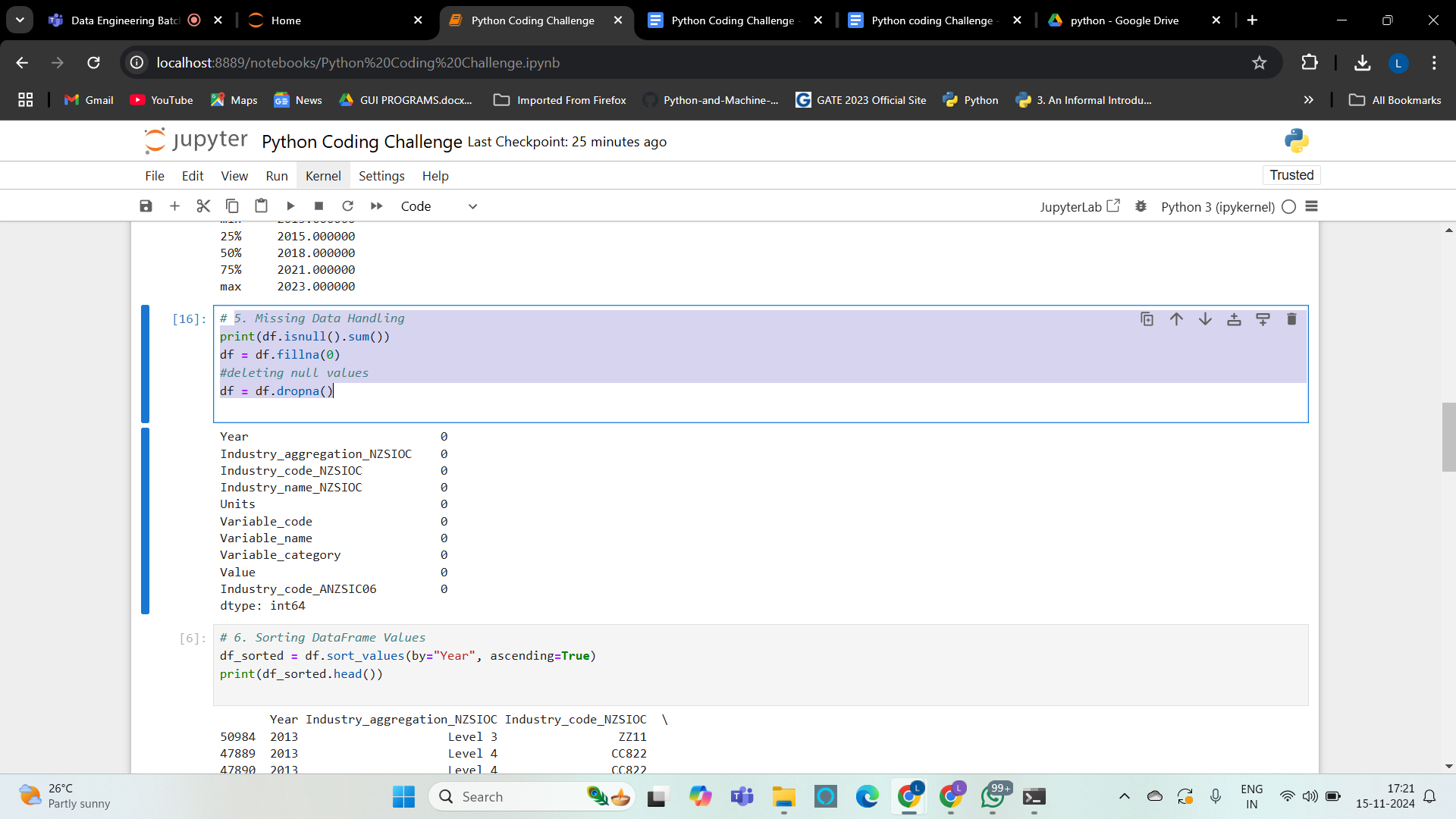
**Code :**

print(df.isnull().sum())

df = df.fillna(0)

#deleting null values

df = df.dropna()



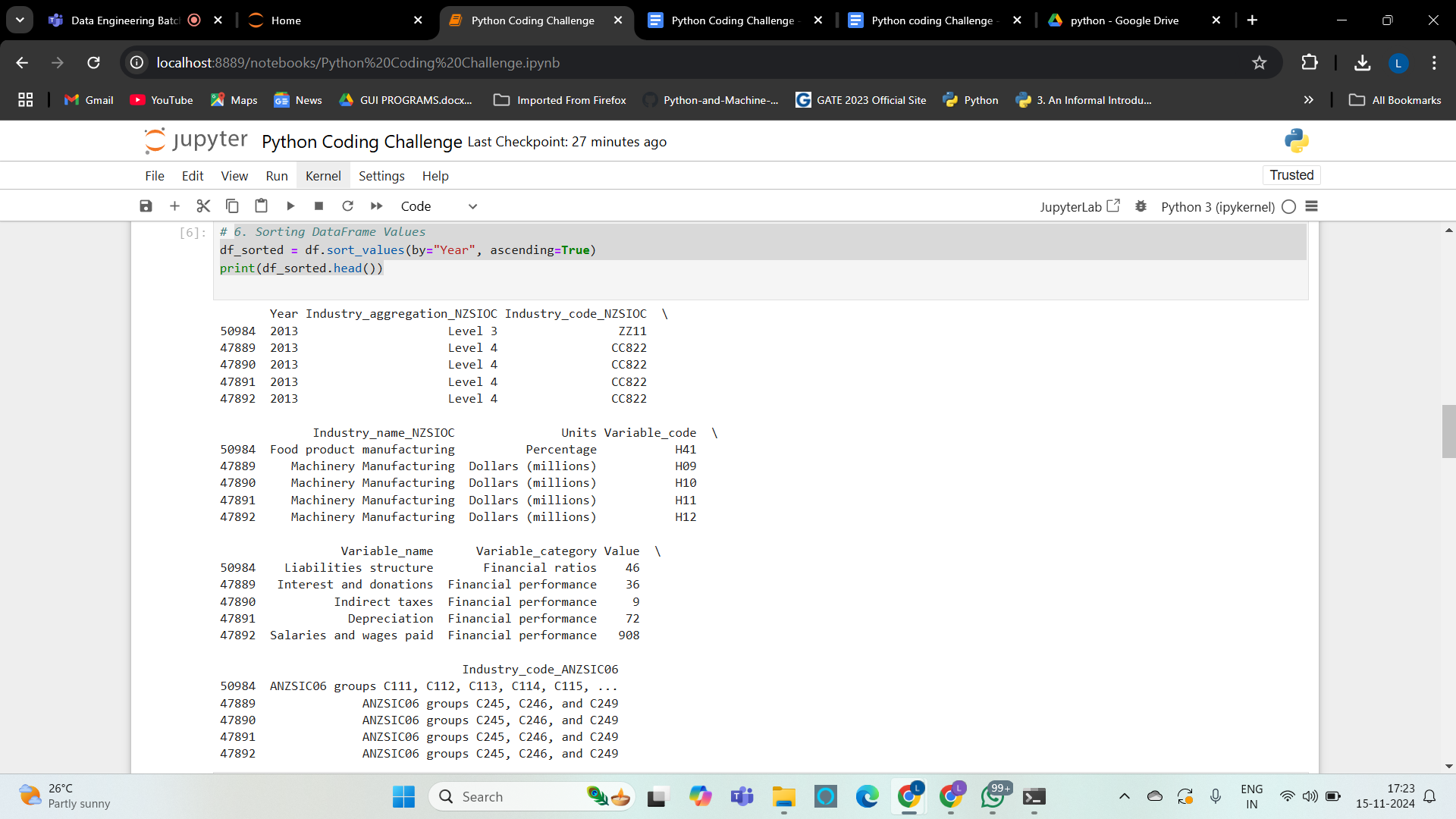
**6. Sorting DataFrame Values :**

* **sort\_values()** sorts the DataFrame by a specified column (e.g., Year).
* Sorting data helps in organizing it for easier visualization and analysis.
* **ascending=True** sorts in ascending order, but we can also set it to False for descending.

**Code :**

df\_sorted = df.sort\_values(by="Year", ascending=True)

print(df\_sorted.head())



**7. Merging Data Frames :**

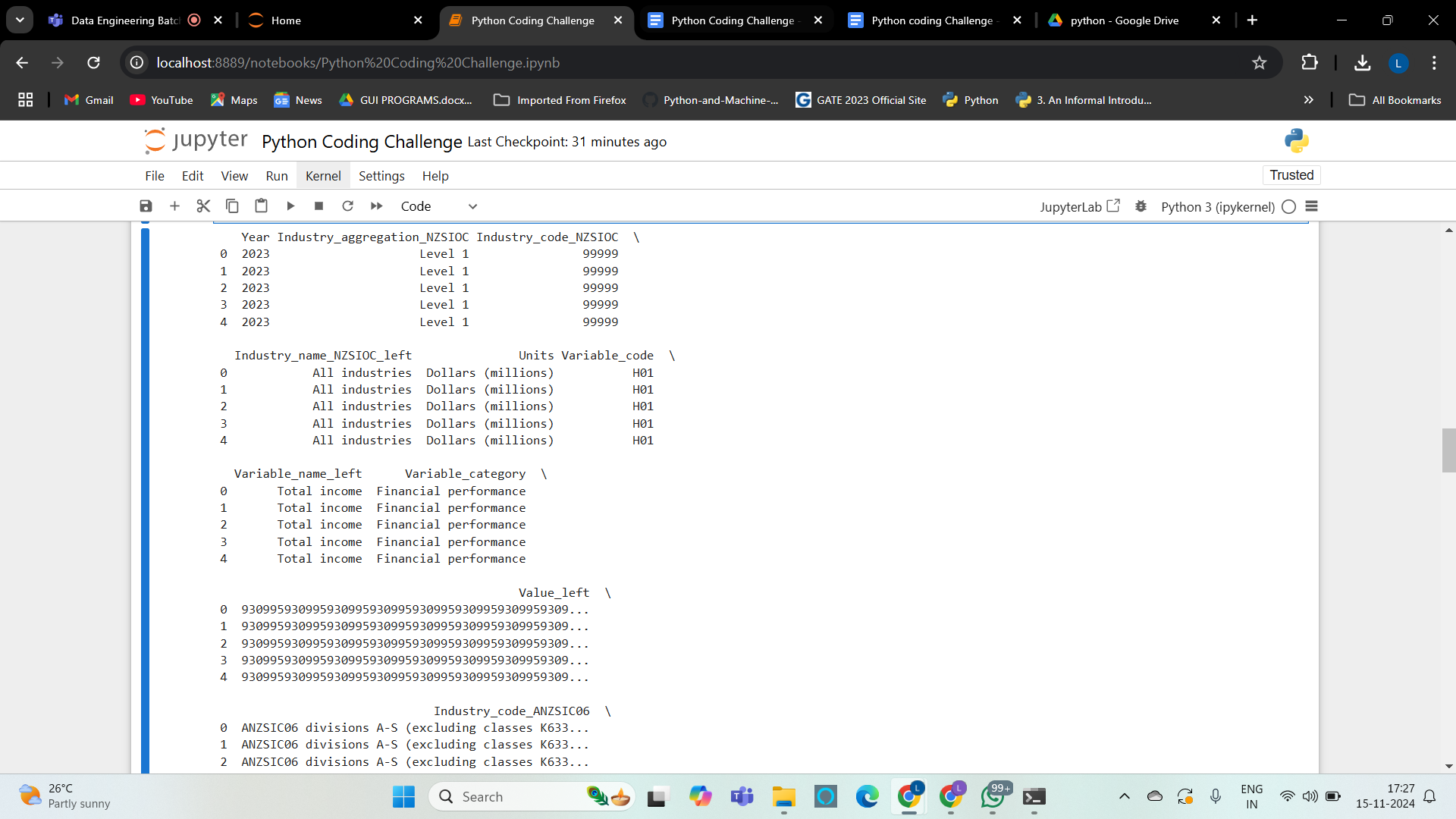
* **Selecting Subset Columns**: df1 is created by selecting columns that could be relevant to merge with the main DataFrame.
* Here, columns like Industry\_code\_ANZSIC06 and Value are selected, with the goal of adding them back in a merged DataFrame to check for consistency or changes.
* Merge Operation: We use **pd.merge()** on the column Industry\_code\_ANZSIC06 to combine df with df1. The suffixes parameter helps differentiate similarly named columns.
* Display Merged Data: **merged\_df.head()** will show the first few rows of the merged DataFrame for verification.

**Code :**

df1 = df[['Industry\_code\_ANZSIC06', 'Industry\_name\_NZSIOC', 'Variable\_name', 'Value']]

merged\_df = pd.merge(df, df1, on="Industry\_code\_ANZSIC06", suffixes=('\_left', '\_right'))

print(merged\_df.head())



**8. Applying a Function :**

* Converting to Numeric: pd.to\_numeric(df['Value'], errors='coerce') converts the Value column to a numeric type. Non-numeric entries are set to NaN.
* Handling NaN Values: increase\_by\_percentage applies the 10% increase only if the value is not NaN.
* Error Prevention: This approach avoids errors by ensuring the function only attempts multiplication on numeric values.

**Code :**

import numpy as np

df['Value'] = pd.to\_numeric(df['Value'], errors='coerce')

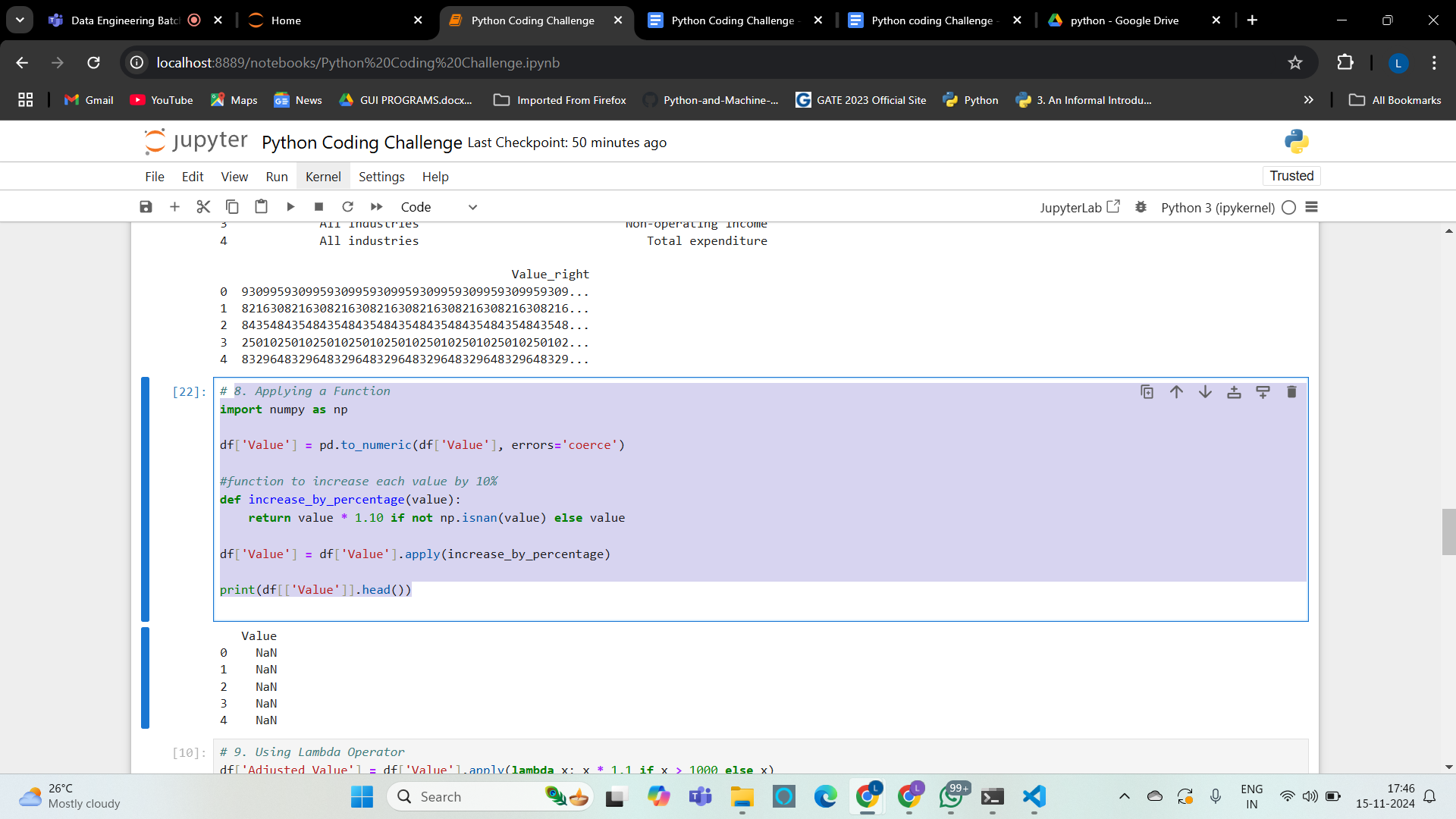
#function to increase each value by 10%

def increase\_by\_percentage(value):

return value \* 1.10 if not np.isnan(value) else value

df['Value'] = df['Value'].apply(increase\_by\_percentage)

print(df[['Value']].head())



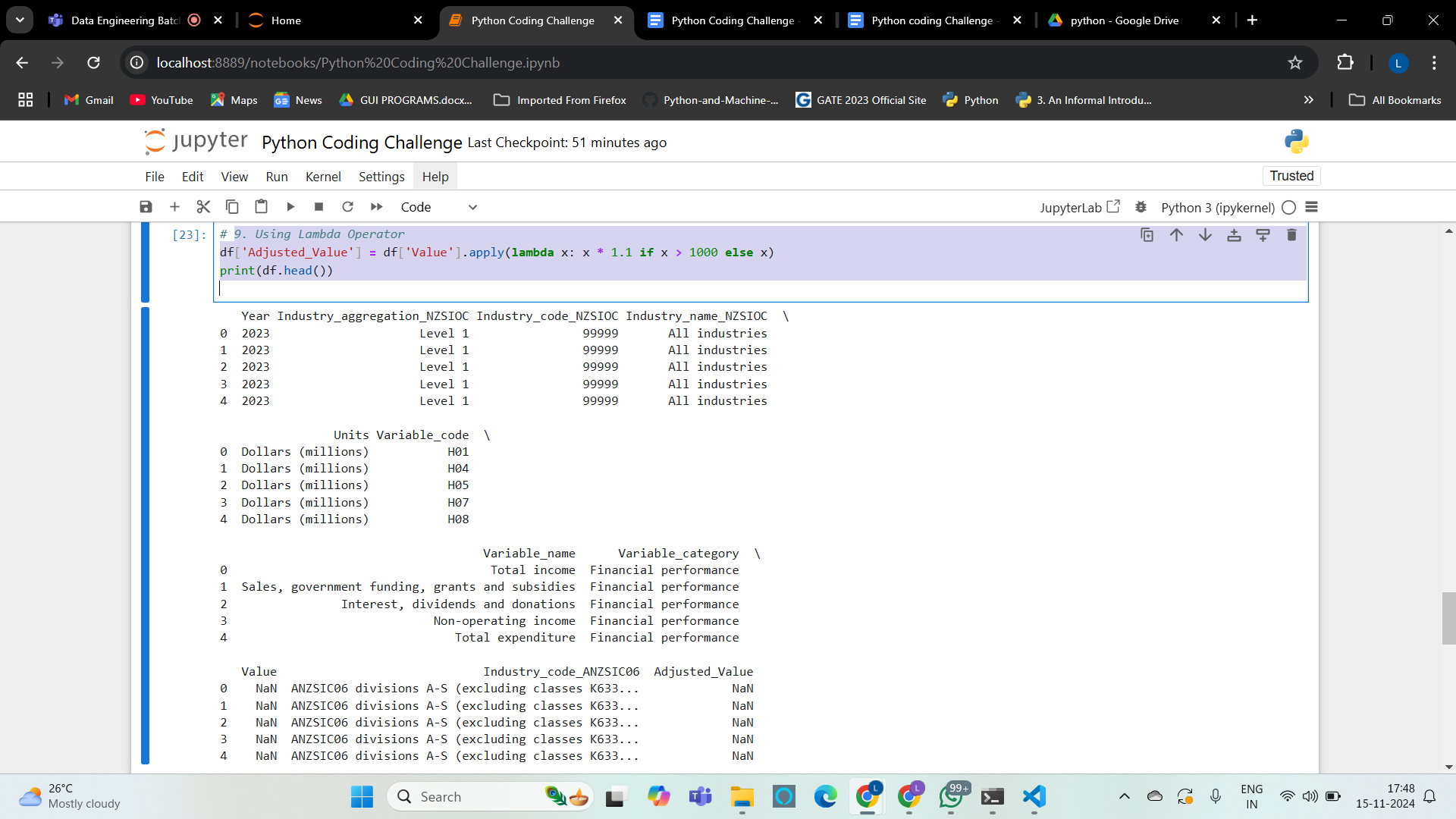
**9. Using Lambda Operator :**

* Converting to Numeric: pd.to\_numeric(df['Value'], errors='coerce') converts the Value column to a numeric type. Non-numeric entries are set to NaN.
* Handling NaN Values: increase\_by\_percentage applies the 10% increase only if the value is not NaN.
* Error Prevention: This approach avoids errors by ensuring the function only attempts multiplication on numeric values.

**Code :**

df['Adjusted\_Value'] = df['Value'].apply(lambda x: x \* 1.1 if x > 1000 else x)

print(df.head())



**10. Visualizing DataFrame :**

* This visualizes the count of entries per year using a bar plot.
* Visualization helps in identifying patterns, trends, and anomalies in data.
* **matplotlib** provides flexibility in customizing plots for better insights.

**Code :**

import matplotlib.pyplot as plt

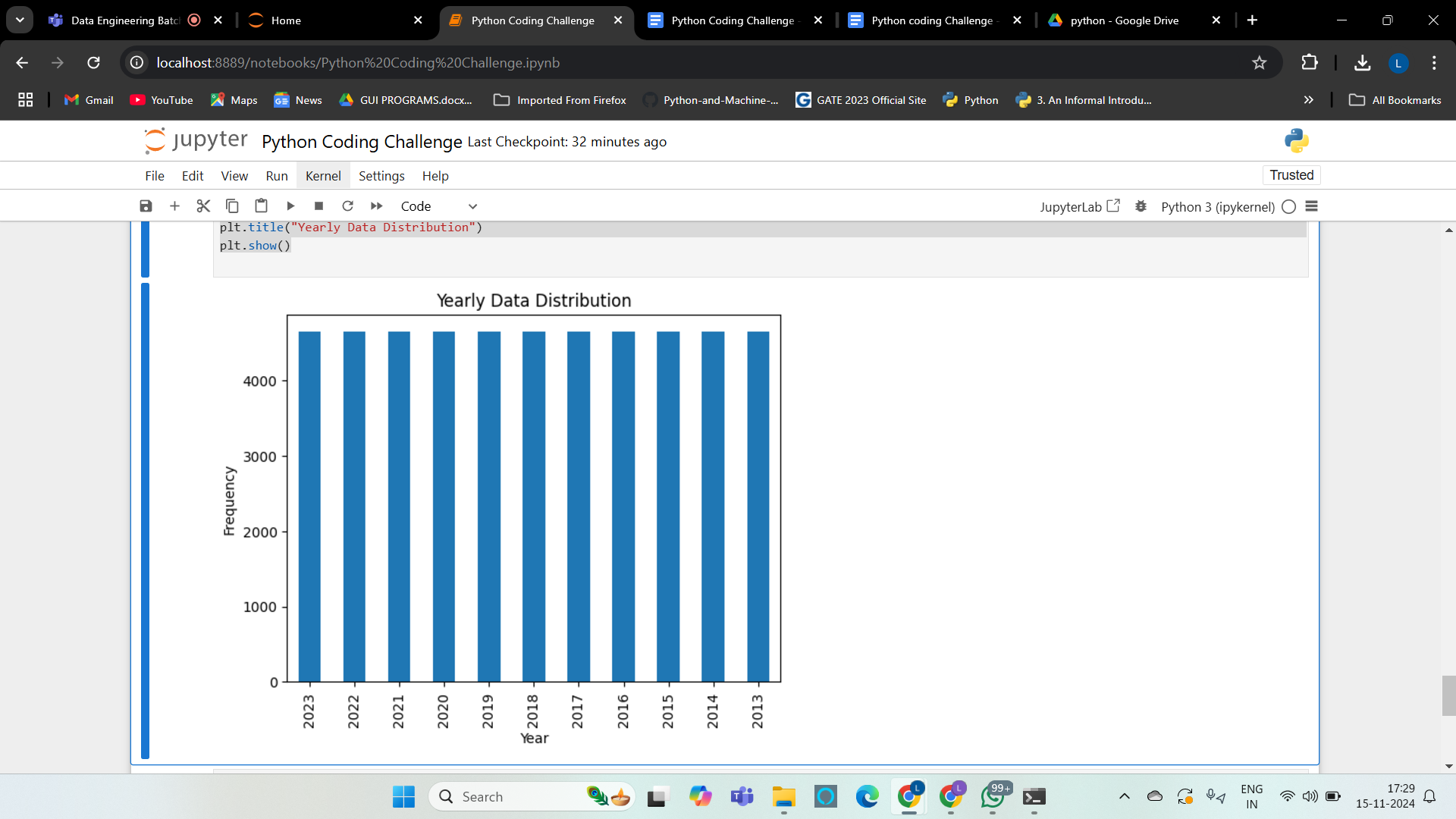
df['Year'].value\_counts().plot(kind='bar')

plt.xlabel("Year")

plt.ylabel("Frequency")

plt.title("Yearly Data Distribution")

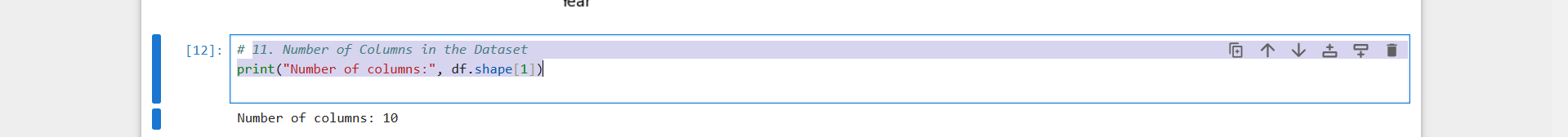
plt.show()



**11. Number of Columns in the Dataset :**

* **df.shape[1]** provides the count of columns in the DataFrame.
* It helps in quickly assessing the DataFrame’s dimensionality.
* Useful for verifying dataset structure against expectations.

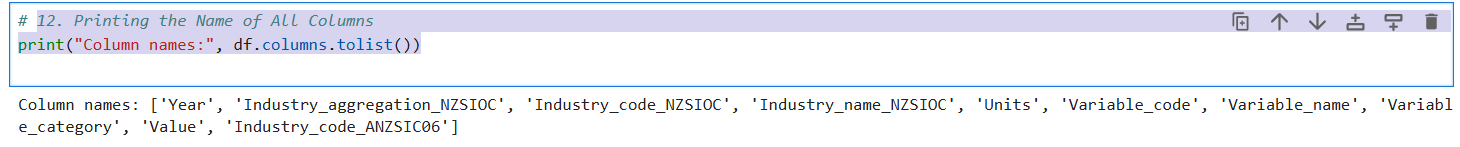
**Code :**

print("Number of columns:", df.shape[1])

**12. Printing the Name of All Columns :**

* **df.columns.tolist()** lists all column names in the DataFrame.
* Ensures all expected columns are present and correctly named.
* Useful for further analysis or feature selection.

**Code :**

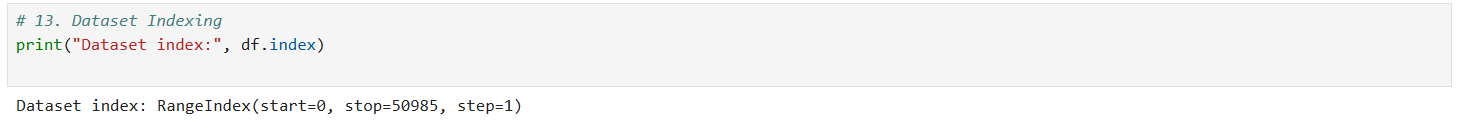
print("Column names:", df.columns.tolist())

**13. Dataset Indexing :**

* **df.index** provides information on the DataFrame index.
* Knowing the index type (RangeIndex, etc.) aids in understanding data access patterns.
* Useful for aligning data or troubleshooting mismatches.

**Code :**

print("Dataset index:", df.index)



**14. Number of Observations in the Dataset :**

* **df.shape[0]** gives the row count or number of observations.
* This count is essential for sample size assessment and analysis.
* Ensures data adequacy for statistical and machine learning purposes.

**Code :**

print("Number of observations:", df.shape[0])

