**Madiha Aimon Tappal**

[madihaaimon@gmail.com](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

**PySpark Coding**

**1. Manipulating PySpark DataFrames:**

* **Transformations:**
  + PySpark DataFrames are immutable, meaning any operation results in a new DataFrame.
  + Transformations, like **select**, **withColumn**, and **alias**, create a new DataFrame with the desired modifications.
* **Expressions:**
  + Operations on columns using expressions, leveraging PySpark's SQL-like syntax.
* **User-Defined Functions (UDFs):**
  + Applying custom functions to columns using **udf** for complex transformations.

**2. Dropping Columns or Rows in PySpark:**

* **Column Dropping:**
  + Using the **drop** method to remove specific columns.
* **Row Dropping:**
  + Filtering rows based on conditions using **filter** or **where**.

**3. Sorting PySpark DataFrames:**

* **Sort vs. OrderBy:**
  + PySpark offers both **sort** and **orderBy** methods for sorting.
  + The primary difference is that **orderBy** is an alias for **sort**.
* **Sorting on Multiple Columns:**
  + Specifying multiple columns for sorting, which defines the hierarchical order.
* **Global Sorting vs. Local Sorting:**
  + Understanding the implications of sorting across the entire DataFrame vs. within partitions.

**4. Aggregations in PySpark:**

* **GroupBy Operations:**
  + Grouping data using **groupBy** and applying aggregate functions like **sum**, **avg**, **min**, **max**.
* **Window Functions:**
  + Performing aggregations over a specific window or range of rows using **window** functions.
* **Pivot and CrossTab:**
  + Reshaping data using **pivot** and **crosstab** for more advanced aggregation scenarios.
* **Aggregating on Multiple Levels:**
  + Utilizing multiple levels of grouping for complex analyses.

**5. User-Defined Aggregations (UDAFs):**

* **Definition:**
  + Creating custom aggregate functions to perform complex aggregations not covered by built-in functions.
* **Registration and Usage:**
  + Registering UDAFs and incorporating them into PySpark DataFrame operations.

**GroupBy and Aggregate function: -**

The theoretical concepts behind the **groupBy** and **agg** functions in PySpark.

**1. groupBy Function:**

The **groupBy** operation is a fundamental concept in data processing and analytics. In the context of PySpark:

* **Grouping Rows:** The **groupBy** function is used to group rows of a DataFrame based on one or more specified columns.
* **Data Partitioning:** It divides the data into groups based on the unique values in the specified columns, forming subsets or partitions of the DataFrame.
* **GroupedData Object:** The result of a **groupBy** operation is a **GroupedData** object, which acts as an intermediary for performing aggregate operations on the grouped data.
* **Parallel Processing:** Under the hood, PySpark performs this grouping in a distributed and parallelized manner, which is essential for handling large-scale datasets across a cluster of machines.

**2. agg (Aggregate) Function:**

The **agg** function is used to perform aggregation operations on grouped data. Key theoretical aspects include:

* **Aggregate Functions:** Aggregation involves applying functions that summarize or compute statistics on the values within each group.
* **Dictionary Syntax:** The **agg** function typically takes a dictionary where keys are column names, and values are the aggregate functions to be applied to those columns.
* **Column Aliases:** It allows renaming the resulting columns by providing aliases, making the output more readable and user-friendly.
* **Flexible Aggregations:** PySpark supports a wide range of built-in and user-defined aggregate functions, providing flexibility in computing various statistics.
* **Distributed Computation:** Similar to the **groupBy** operation, the **agg** function operates in a distributed fashion, allowing it to efficiently handle large datasets across a cluster.

In summary, **groupBy** and **agg** functions in PySpark enable efficient and parallelized data grouping and aggregation, which is crucial for processing and analyzing massive datasets in distributed computing environments. They form the foundation for performing complex analytics and computations on Big Data using the PySpark framework.

Similar to SQL GROUP BY clause, PySpark groupBy() function is used to collect the identical data into groups on DataFrame and perform count, sum, avg, min, and max functions on the grouped data.

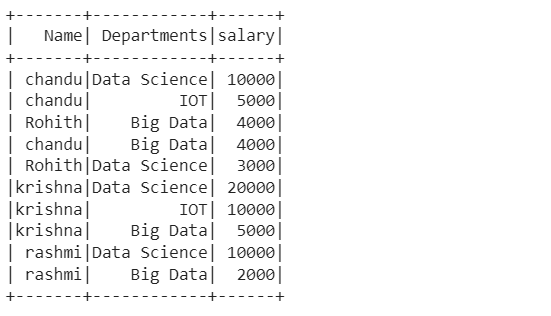
Before starting, let's create a simple DataFrame to work with. The CSV file used can be found here.

from pyspark.sql import SparkSession

spark =SparkSession.builder.appName("Practice").getOrCreate()

df\_pyspark= spark.read.csv("test2.csv",header=True,inferSchema=True)  
df\_pyspark.show()

df\_pyspark.groupBy("Departments").sum("salary").show()

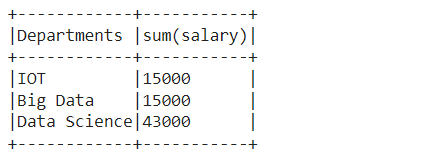


groupBy():

groupBy() on Departments column of DataFrame and then find the sum of salary for each department using sum() function.

#group by Departments which gives summation of salaries

df\_pyspark.groupBy("Departments").sum("salary").show()



Similarly, we can perform min, max, mean, avg, and count using the groupBy function.

df\_pyspark.groupBy("Departments").min("salary").show()  
df\_pyspark.groupBy("Departments").max("salary").show()  
df\_pyspark.groupBy("Departments").avg("salary").show()  
df\_pyspark.groupBy("Departments").mean("salary").show()

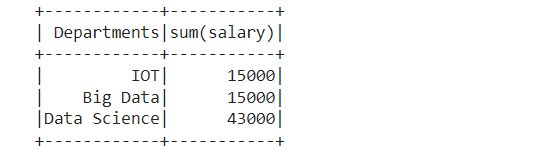
df\_pyspark.groupBy("Departments").count().show() #count of number of people in each Department

groupBy() using multiple columns

df\_pyspark.groupBy("Name","Departments").sum("salary").show()

groupBy() and agg() function

df\_pyspark.groupBy("Departments").agg(({"salary":"sum"})).show()

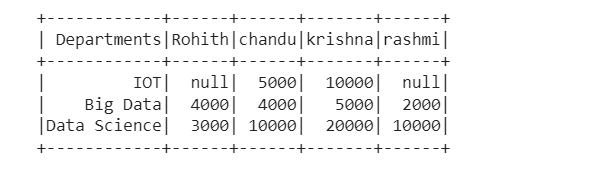


We can also perform agg() function on entire DataFrame without groupBy()

df\_pyspark.agg(({"salary":"sum"})).show()+-----------+   
|sum(salary)|   
+-----------+   
| 73000|   
+-----------+

Using Pivot/ UnPivot — Spark SQL provides pivot() function to rotate the data from one column into multiple columns (transpose row to column). It is an aggregation where one of the grouping column values is transposed into individual columns with distinct data. Similary UnPivot can be used.

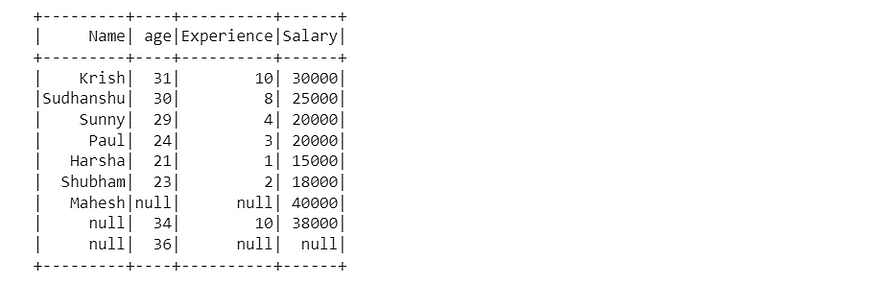
df\_pyspark.groupBy("Departments").pivot("Name").sum("salary").show()



Handling Missing Values Pyspark

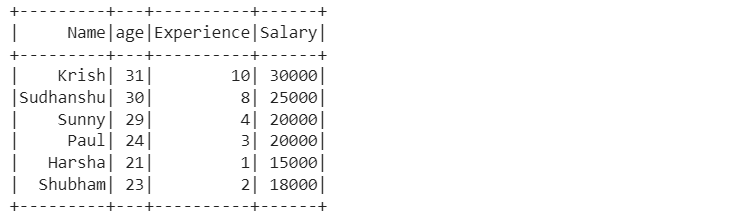
For this task, we will be using this CSV file

df\_pyspark1=spark.read.csv("test3.csv",header=True,inferSchema=True)  
df\_pyspark1.show()



Dropping rows based on null values

df\_pyspark1.na.drop().show()



drop() has the following parameters — how, thresh, and subset

1. df\_pyspark1.na.drop(how="all").show()

# if all values in rows are null then drop # default any

1. df\_pyspark1.na.drop(how="any",thresh=2).show()

#atleast 2 non null values should be present. df\_pyspark1.na.drop(how="any",subset=["salary"]).show()

# only in that column rows get deleted

Filling missing values — Single Value

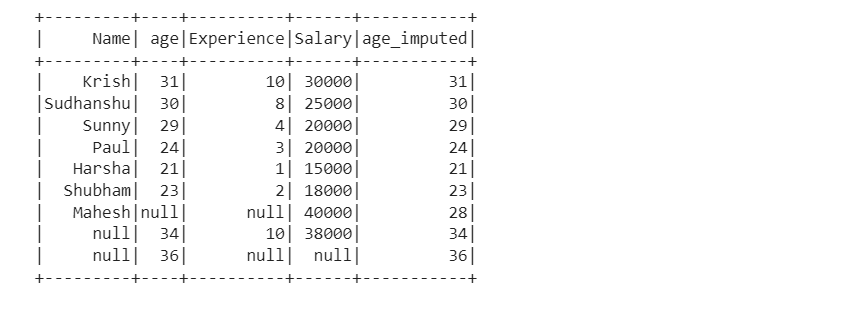
df\_pyspark1.na.fill('Missing Values').show() #string values will get replaced as string is given as inputdf\_pyspark1.na.fill(0).show() #integer values will get replaced as integer is given as input

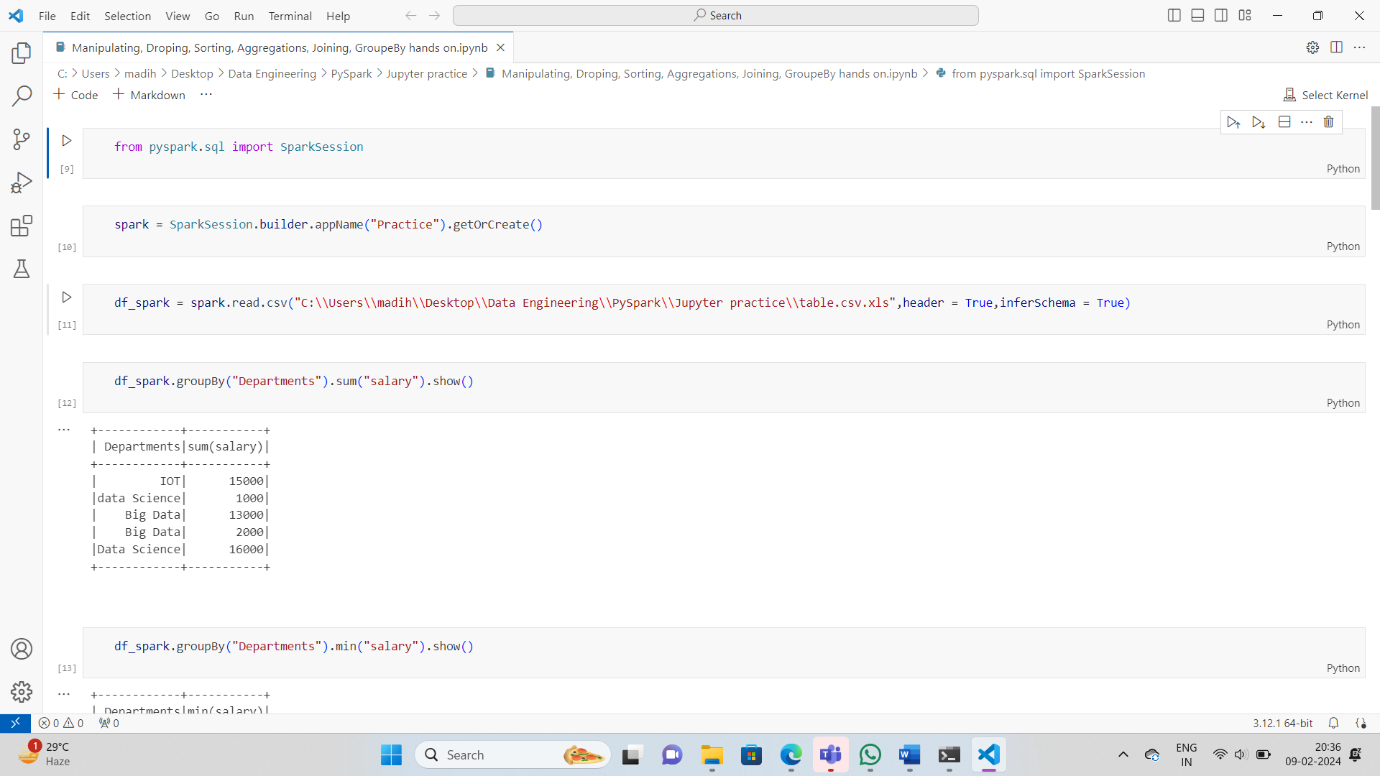
Filling missing values using Mean, Median, or Mode with help of the Imputer function

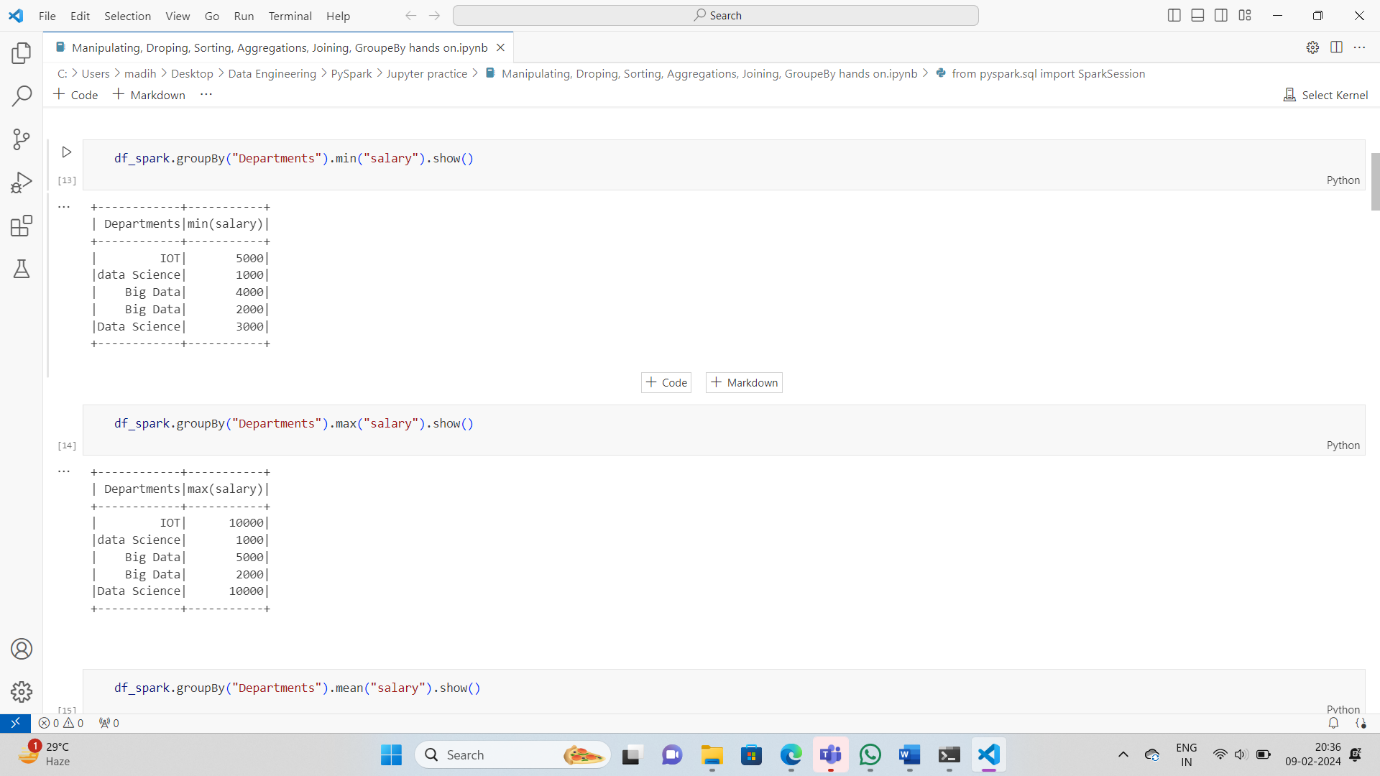
#filling with meanfrom pyspark.ml.feature import Imputerimputer = Imputer(inputCols=["age"],outputCols=["age\_imputed"]).setStrategy("mean")

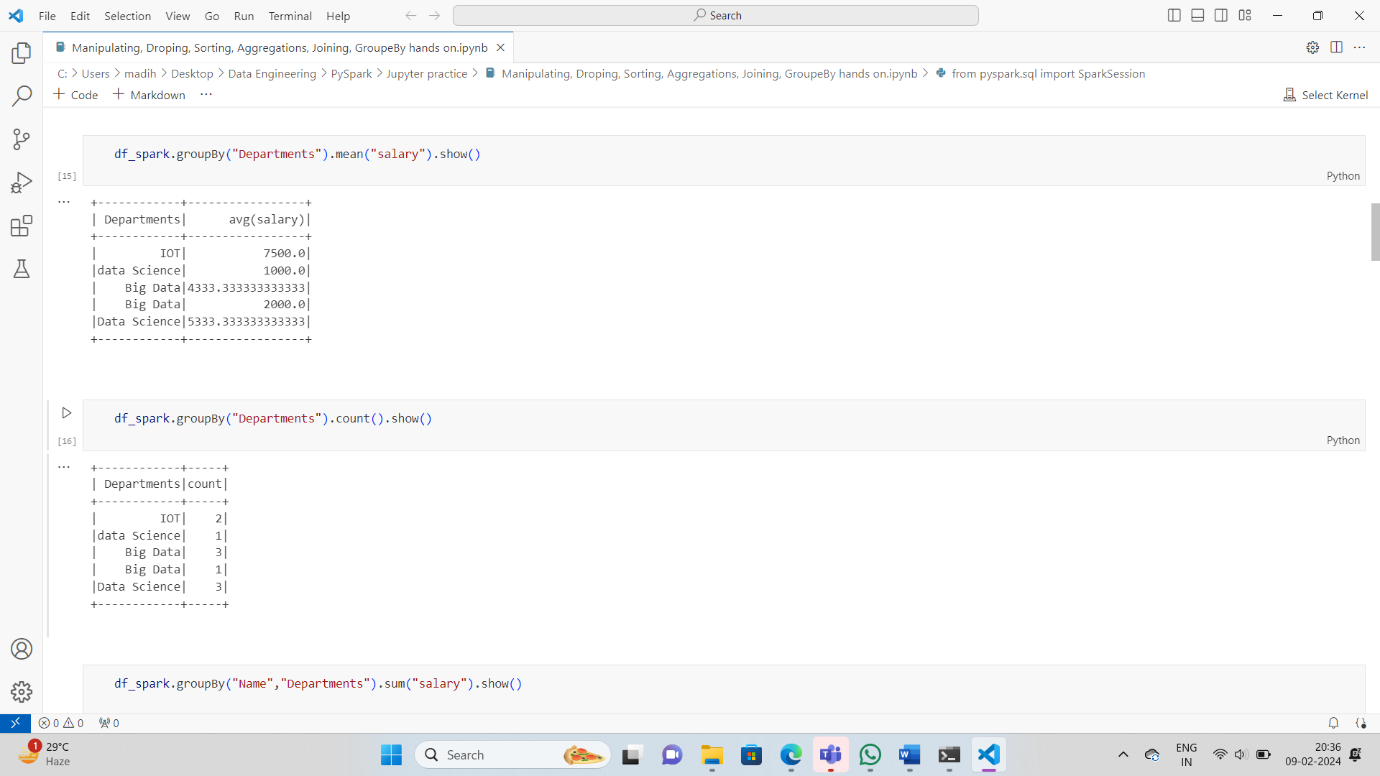
In setStrategy we can use mean, median, or mode.

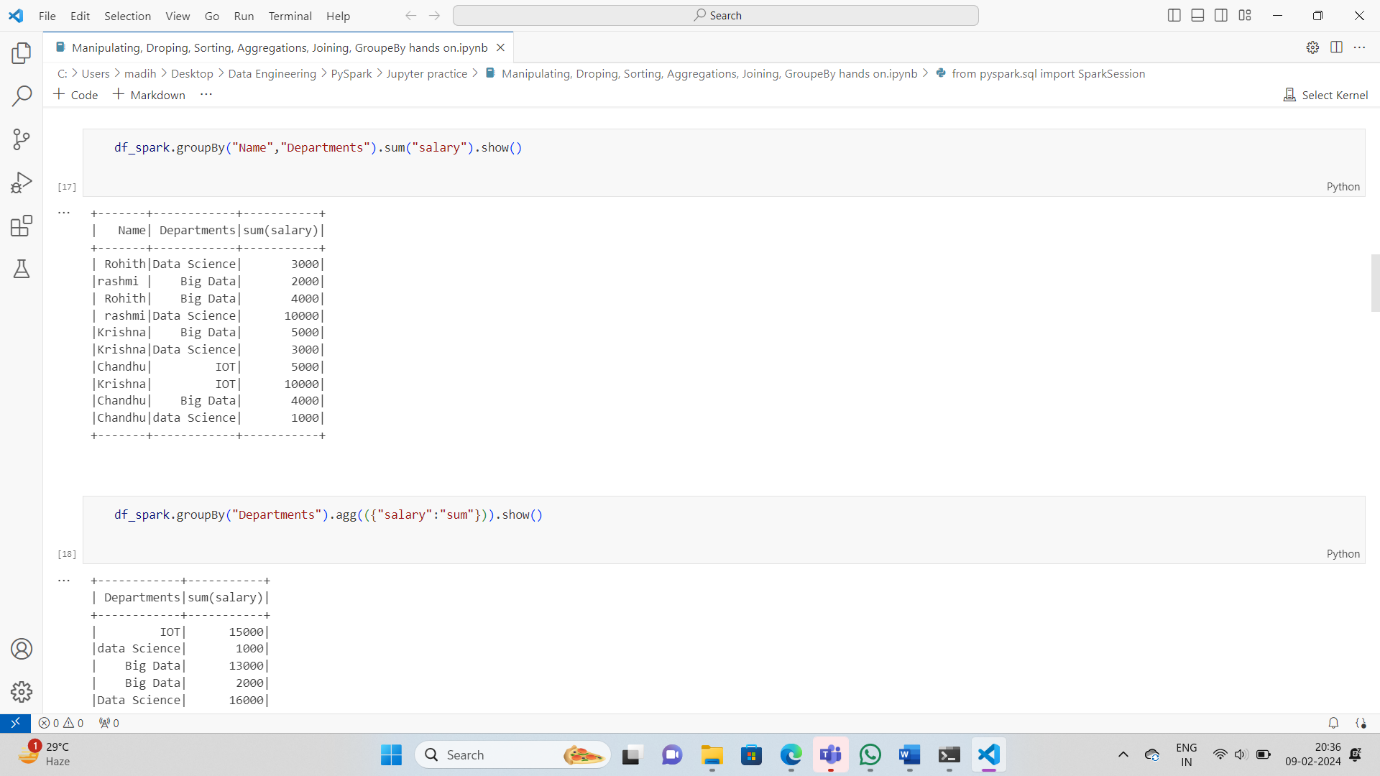
imputer.fit(df\_pyspark1).transform(df\_pyspark1).show()

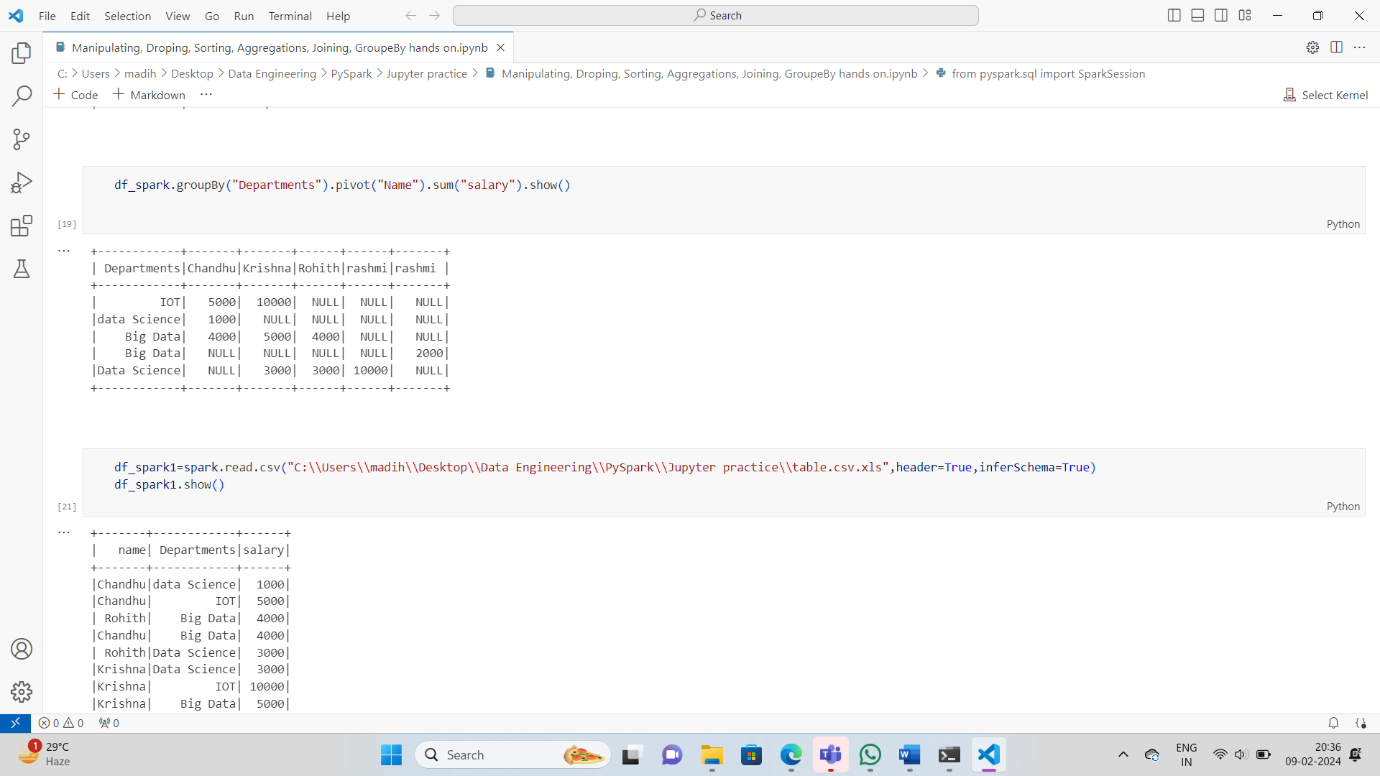


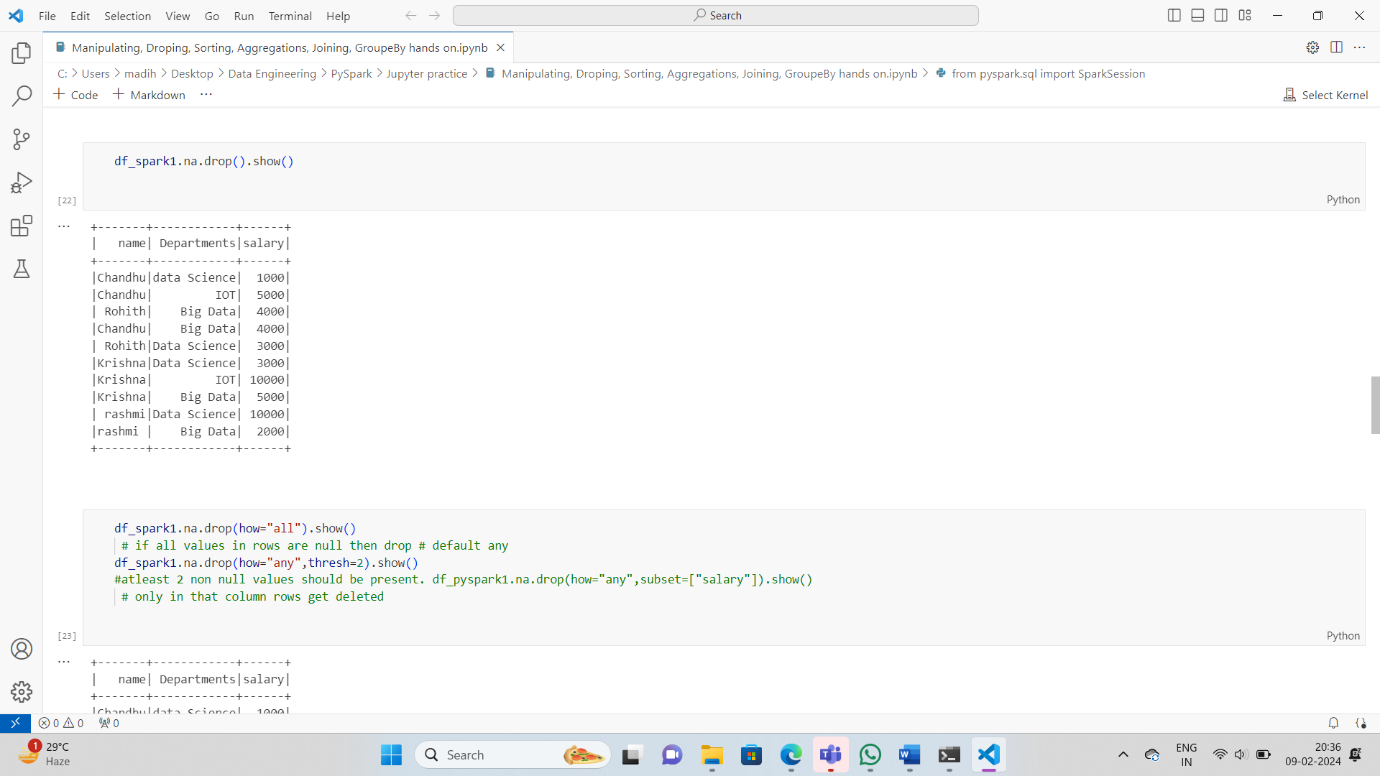
****

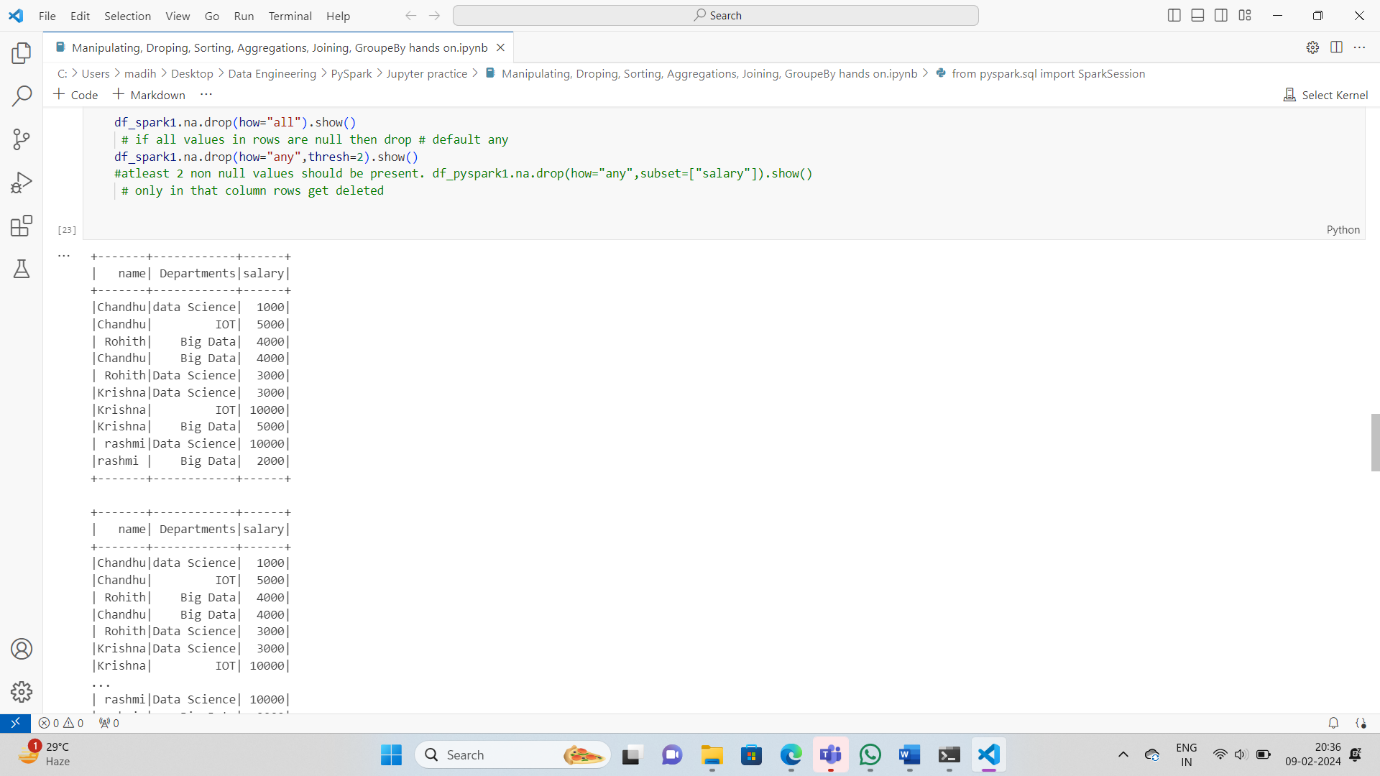
****

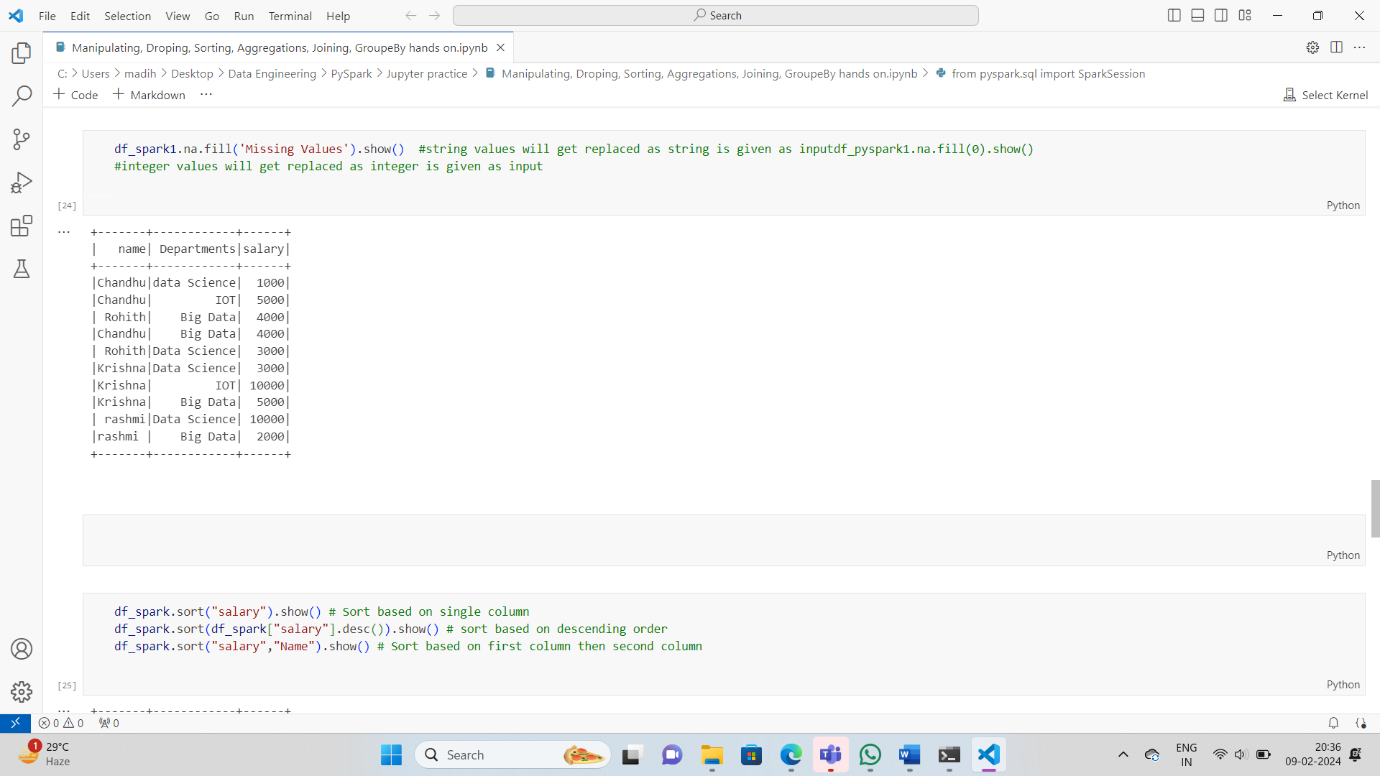
****

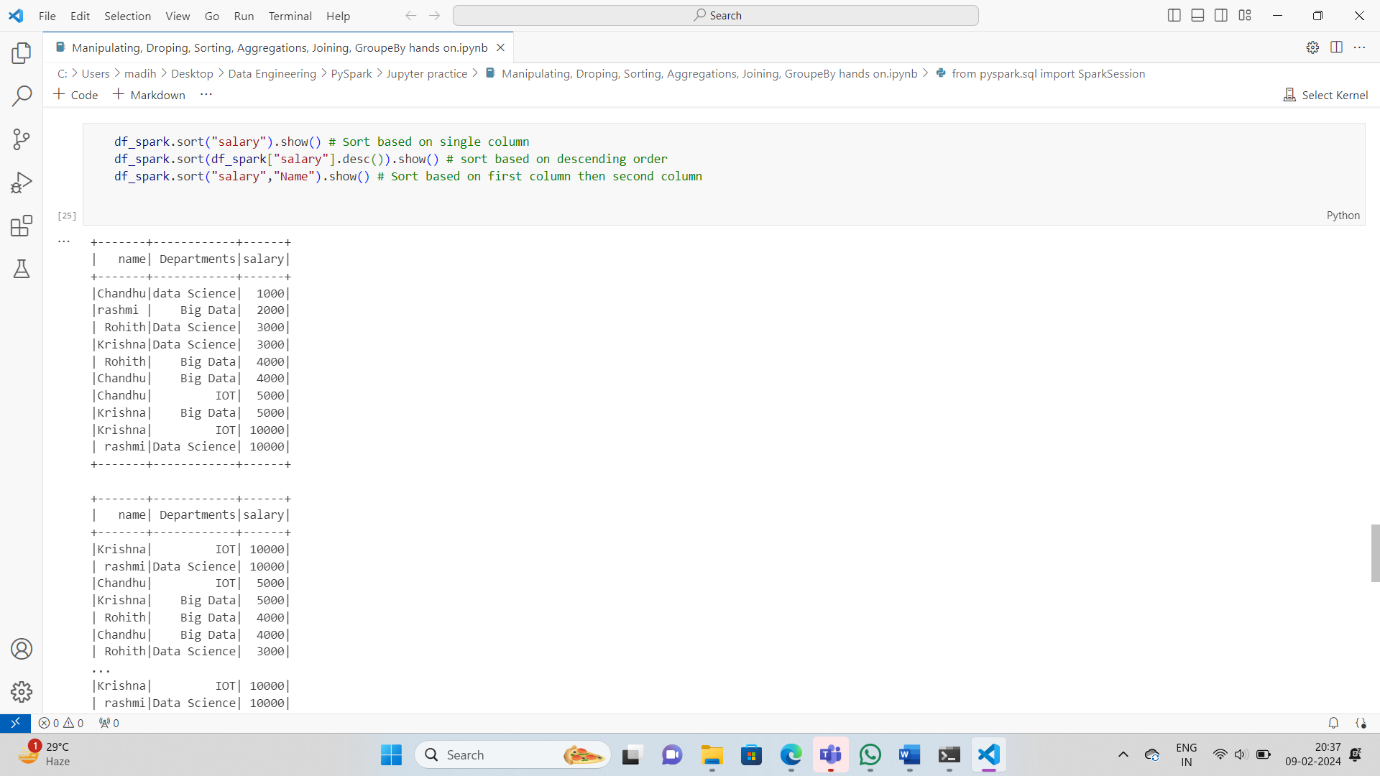
****

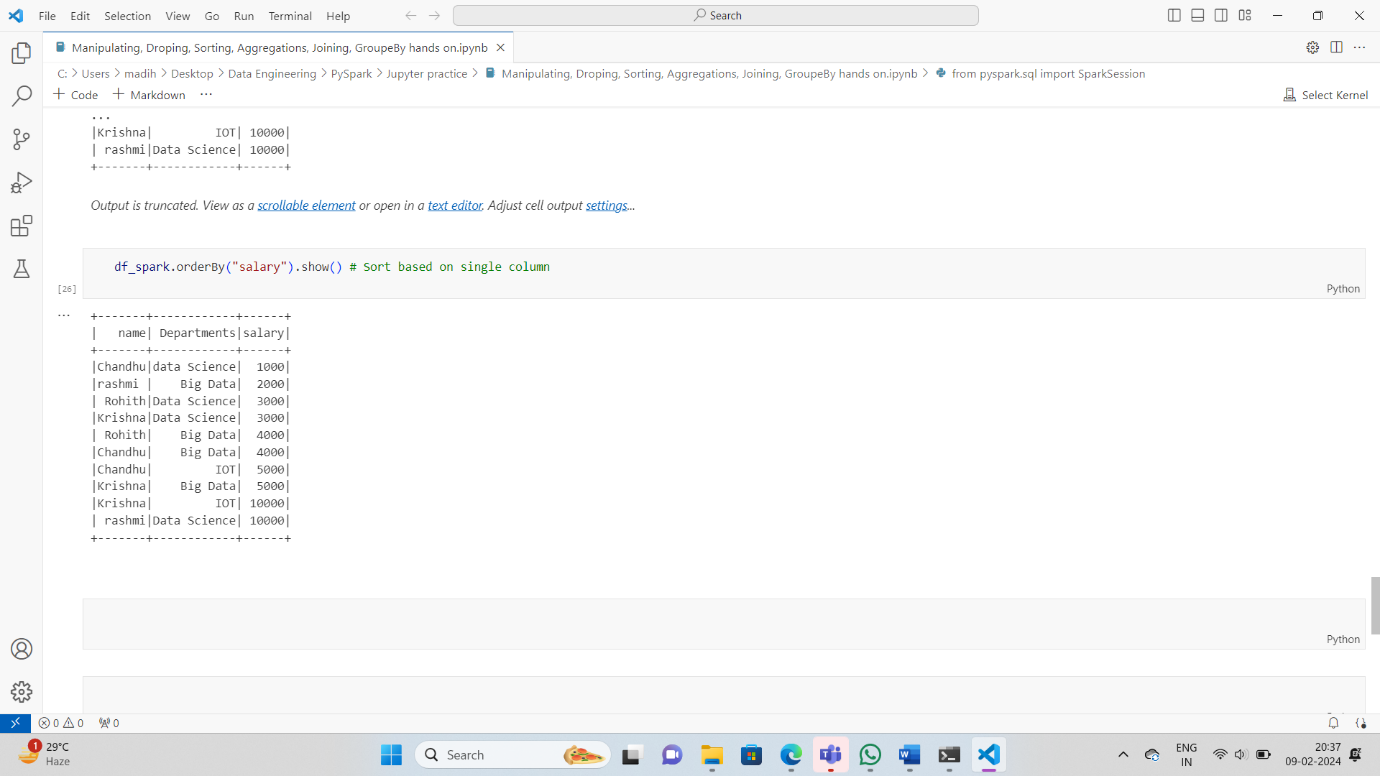
****

****

****

****

****

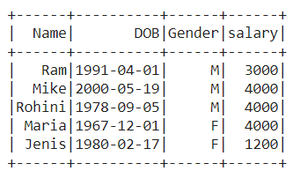
****

**Selecting, Renaming, Filtering Data in a Pandas DataFrame**

**Python3**

|  |
| --- |
| # Importing necessary libraries  **from** pyspark.sql **import** SparkSession    # Create a spark session  spark **=** SparkSession.builder.appName('pyspark - example join').getOrCreate()    # Create data in dataframe  data **=** [(('Ram'), '1991-04-01', 'M', 3000),          (('Mike'), '2000-05-19', 'M', 4000),          (('Rohini'), '1978-09-05', 'M', 4000),          (('Maria'), '1967-12-01', 'F', 4000),          (('Jenis'), '1980-02-17', 'F', 1200)]    # Column names in dataframe  columns **=** ["Name", "DOB", "Gender", "salary"]    # Create the spark dataframe  df **=** spark.createDataFrame(data**=**data,                             schema**=**columns)    # Print the dataframe  df.show() |

**Output:**



**Method 1: Using withColumnRenamed()**

We will use of withColumnRenamed() method to change the column names of pyspark data frame.

***Syntax:****DataFrame.withColumnRenamed(existing, new)*

***Parameters***

* ***existingstr:****Existing column name of data frame to rename.*
* ***newstr:****New column name.*
* ***Returns type:****Returns a data frame by renaming an existing column.*

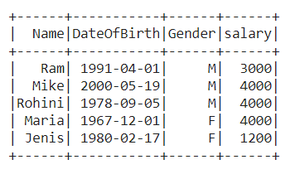
**Example 1:**Renaming the single column in the data frame

Here we’re Renaming the column name ‘DOB’ to ‘DateOfBirth’.

**Python3**

|  |
| --- |
| # Rename the column name from DOB to DateOfBirth  # Print the dataframe  df.withColumnRenamed("DOB","DateOfBirth").show() |

**Output:**

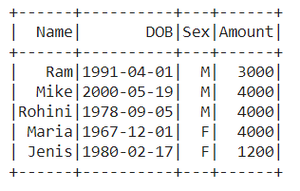


**Example 2:** Renaming multiple column names

**Python3**

|  |
| --- |
| # Rename the column name 'Gender' to 'Sex'  # Then for the returning dataframe  # again rename the 'salary' to 'Amount'  df.withColumnRenamed("Gender","Sex").  withColumnRenamed("salary","Amount").show() |

**Output:**



**Method 2: Using selectExpr()**

Renamingthe column names using**selectExpr()**method

***Syntax :****DataFrame.selectExpr(expr)*

***Parameters :***

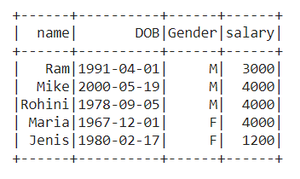
***expr :****It’s an**SQL expression.*

Here we are renaming Name as a name.

**Python3**

|  |
| --- |
| # Select the 'Name' as 'name'  # Select remaining with their original name  data **=** df.selectExpr("Name as name","DOB","Gender","salary")    # Print the dataframe  data.show() |

**Output:**



**Method 3: Using select() method**

***Syntax:****DataFrame.select(cols)*

***Parameters :***

***cols:****List of column names as strings.*

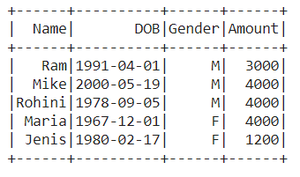
***Return type:****Selects the cols in the dataframe and returns a new DataFrame.*

Here we Rename the column name ‘salary’ to ‘Amount’

**Python3**

|  |
| --- |
| # Import col method from pyspark.sql.functions  **from** pyspark.sql.functions **import** col    # Select the 'salary' as 'Amount' using aliasing  # Select remaining with their original name  data **=** df.select(col("Name"),col("DOB"),                   col("Gender"),                   col("salary").alias('Amount'))    # Print the dataframe  data.show() |

**Output:**



**Method 4: Using toDF()**

This function returns a new DataFrame that with new specified column names.

***Syntax:****toDF(\*col)*

*Where, col is a new column name*

In this example, we will create an order list of new column names and pass it into toDF function

**Python3**

|  |
| --- |
| Data\_list **=** ["Emp Name","Date of Birth",               " Gender-m/f","Paid salary"]    new\_df **=** df.toDF(**\***Data\_list)  new\_df.show() |

**Output:**

