**Madiha Aimon Tappal**

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

**Data Engineering Batch – 1**

**Day – 14 Assignment**

**PySpark**

PySpark RDDs and Pair RDDs. Both PySpark RDD and Pair RDDs consists of two types of operations namely, Transformations and Actions. We will learn more about them in the following lines. These operations are very useful and since these actions and transformations are in Python, one can get easily used to these methods.

**PySpark RDD Operations**

PySpark RDD (Resilient Distributed Dataset) is the foundational data structure in Apache Spark, a distributed computing framework for large-scale data processing. RDDs are immutable, partitioned collections of objects that can be processed in parallel across a cluster of machines. Here are some key theoretical aspects of PySpark RDD operations:

**1. Immutability:**

* RDDs are immutable, meaning their contents cannot be changed once created. If you want to transform an RDD, you create a new RDD with the desired transformations.
* Immutability ensures fault tolerance by allowing the recreation of lost data by recomputing transformations.

**2. Resilience:**

* The term "Resilient" in RDD stands for fault tolerance. RDDs automatically recover from node failures by recomputing lost partitions.
* Lineage information is stored for each RDD, allowing Spark to reconstruct lost data by recomputing from the original data source.

**3. Parallel Processing:**

* RDDs support parallel processing across a cluster of machines. Spark automatically distributes the data and computation across nodes in the cluster.
* Each partition of an RDD can be processed independently, enabling parallel execution of transformations.

**4. Transformations and Actions:**

* RDD operations can be broadly categorized into transformations and actions.
* **Transformations** are operations that create a new RDD from an existing one, like **map**, **filter**, **union**, **groupByKey**, etc.
* **Actions** are operations that return a value to the driver program or write data to an external storage system, such as **collect**, **count**, **saveAsTextFile**, etc.

**5. Lazy Evaluation:**

* Spark uses lazy evaluation, which means that transformations on RDDs are not executed immediately. Instead, they are recorded as a sequence of transformations to be applied later.
* Spark only computes the result when an action is triggered. This optimizes the execution plan and reduces unnecessary computation.

**6. Narrow and Wide Transformations:**

* Transformations are categorized into narrow (e.g., **map**, **filter**) and wide (e.g., **groupByKey**, **reduceByKey**) transformations.
* Narrow transformations result in dependencies between a single parent partition and its child partitions, while wide transformations involve dependencies between multiple partitions.

**7. Persistence (Caching):**

* RDDs can be persisted in memory or on disk to avoid recomputation, especially for iterative algorithms.
* The **cache** or **persist** methods can be used to store intermediate results, improving the performance of iterative algorithms.

**8. Partitioning:**

* RDDs are divided into partitions, which are the basic units of parallelism.
* The number of partitions affects parallelism and performance. Partitioning can be controlled using the **repartition** and **coalesce** methods.

**9. Broadcast Variables:**

* Broadcast variables are read-only variables cached on each worker node, reducing the need to transfer large read-only data sets over the network.

Resilient Distributed Dataset or RDD in a PySpark is a core data structure of PySpark. PySpark RDD’s is a low-level object and are highly efficient in performing distributed tasks. This article will not involve the basics of PySpark such as the creation of PySpark RDDs and PySpark DataFrames.

PySpark RDD has a set of operations to accomplish any task. These operations are of two types:

1. Transformations

2. Actions

**Transformations**are a kind of operation that takes an RDD as input and produces another RDD as output. Once a transformation is applied to an RDD, it returns a new RDD, the original RDD remains the same and thus are immutable. After applying the transformation, it creates a Directed Acyclic Graph or DAG for computations and ends after applying any actions on it. This is the reason they are called lazy evaluation processes.

**Actions**are a kind of operation which are applied on an RDD to produce a single value. These methods are applied on a resultant RDD and produces a non-RDD value, thus removing the laziness of the transformation of RDD.

To conclude in Layman’s Terms, Transformations are applied on an RDD to give another RDD. While Actions are performed on an RDD to give a non-RDD value.

!pip install pyspark

Next, we will initialize a SparkContext to perform the operations:

from pyspark import SparkContext

sc = SparkContext.getOrCreate()

. Now, we have SparkContext ready with us, we get to perform all the Actions and Transformations coming next.

**Actions in PySpark RDDs**

In PySpark RDDs, Actions are a kind of operation that returns a value on being applied to an RDD.

Following are some of the essential PySpark RDD Operations widely used.

**1. The .collect() Action**

The **.collect()** action on an RDD returns a list of all the elements of the RDD. It’s a great asset for displaying all the contents of our RDD. Let’s understand this with an example:

collect\_rdd = sc.parallelize([1,2,3,4,5])

print(collect\_rdd.collect())

On executing this code, we get:

Output

Here we first created an RDD, **collect\_rdd**, using the **.parallelize()** method of SparkContext. Then we used the **.collect()** method on our RDD which returns the list of all the elements from **collect\_rdd**.

**2. The .count() Action**

The **.count()** action on an RDD is an operation that returns the number of elements of our RDD. This helps in verifying if a correct number of elements are being added in an RDD. Let’s understand this with an example:

from pyspark import SparkContext

sc = SparkContext.getOrCreate()

count\_rdd = sc.parallelize([1,2,3,4,5,5,6,7,8,9])

print(count\_rdd.count())

On executing this code, we get:

Output

Here, we first created an RDD, **count\_rdd**, using the **.parallelize()** method of SparkContext. Then we applied the **.count()** method on our RDD which returned the number of elements present in our RDD.

**3. The .first() Action**

The **.first()** action on an RDD returns the first element from our RDD. This can be helpful when we want to verify if the exact kind of data has been loaded in our RDD as per the requirements. For example, if wanted an RDD with the first 10 natural numbers. We can verify this by checking the first element from our RDD i.e. 1. Let’s understand this with an example:

from pyspark import SparkContext

sc = SparkContext.getOrCreate()

count\_rdd = sc.parallelize([1,2,3,4,5,5,6,7,8,9])

print(count\_rdd.count())

first\_rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])

print(first\_rdd.first())

On executing this code, we get:

Output

Here, we first created an RDD, **first\_rdd**using the **.parallelize()** method of SparkContext having the first ten natural numbers. Then, we applied the **.first()** operation on first\_rdd. This returned the first element from first\_rdd, i.e. 1.

**4. The .take() Action**

The **.take(n)** action on an RDD returns n number of elements from the RDD. The ‘n’ argument takes an integer which refers to the number of elements we want to extract from the RDD. Let’s understand this with an example:

take\_rdd = sc.parallelize([1,2,3,4,5])

print(take\_rdd.take(3))

On executing this code, we get:

Output

Here, we first created an RDD, **take\_rdd**, using the **.parallelize()** method of SparkContext. Then we applied the **.take(3)** method on our RDD **take\_rdd**. This returned the first 3 elements in a list from the RDD.

**5. The .reduce() Action**

The **.reduce()** Actiontakes two elements from the given RDD and operates. This operation is performed using an anonymous function or lambda. For example, if we want to add all the elements from the given RDD, we can use the .reduce() action.

from pyspark import SparkContext

sc = SparkContext.getOrCreate()

reduce\_rdd = sc.parallelize([1,3,4,6])

print(reduce\_rdd.reduce(lambda x, y : x + y))

On executing this code, we get:

Output

Here, we created an RDD, **reduce\_rdd**using **.parallelize()** method of SparkContext. We used the .reduce action on **reduce\_rdd**with an enclosed anonymous function or lambda. Here, the lambda adds all the elements of the given RDD and prints the sum.

**6. The .saveAsTextFile() Action**

The **.saveAsTextFile()** Action is used to serve the resultant RDD as a text file. We can also specify the path to which file needed to be saved. This helps in saving our results especially when we are working with a large amount of data.

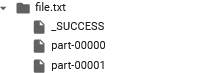
from pyspark import SparkContext

sc = SparkContext.getOrCreate()

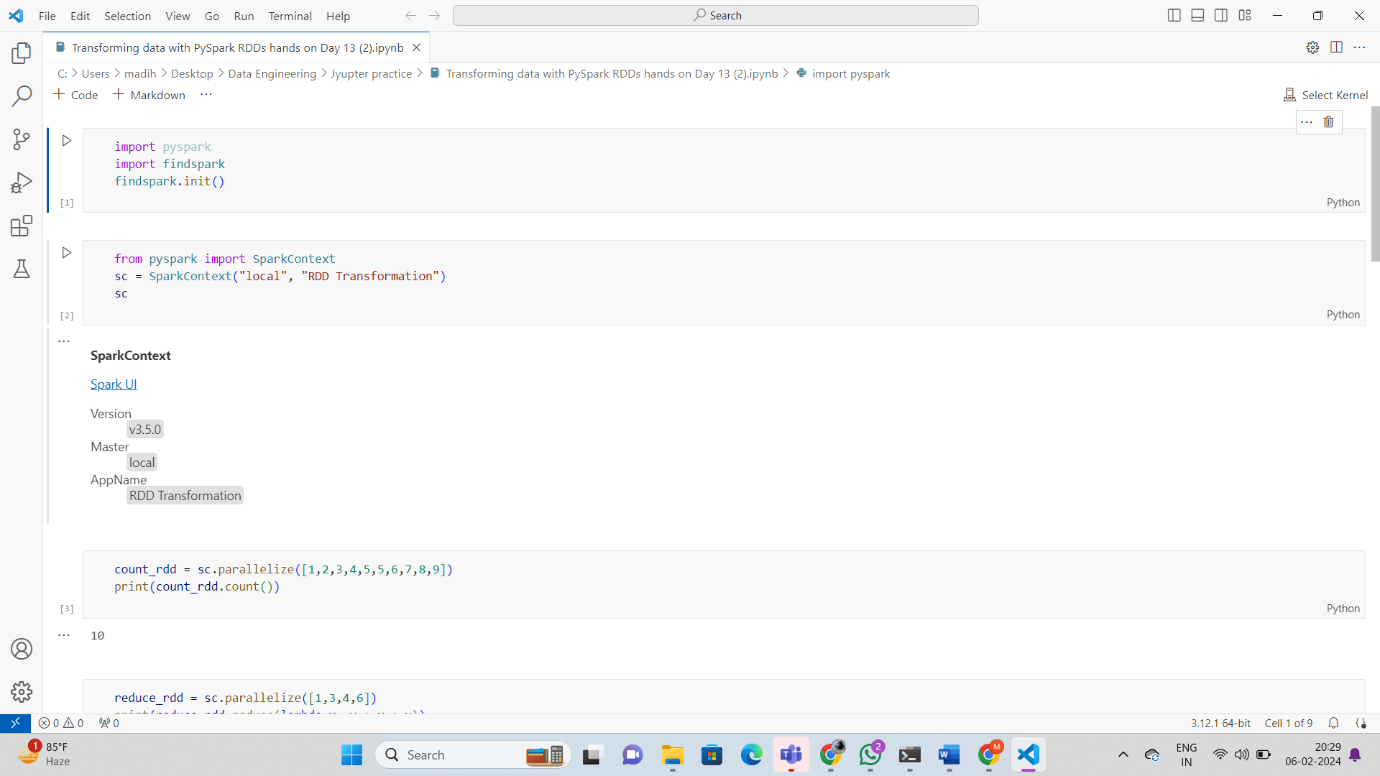
save\_rdd = sc.parallelize([1,2,3,4,5,6])

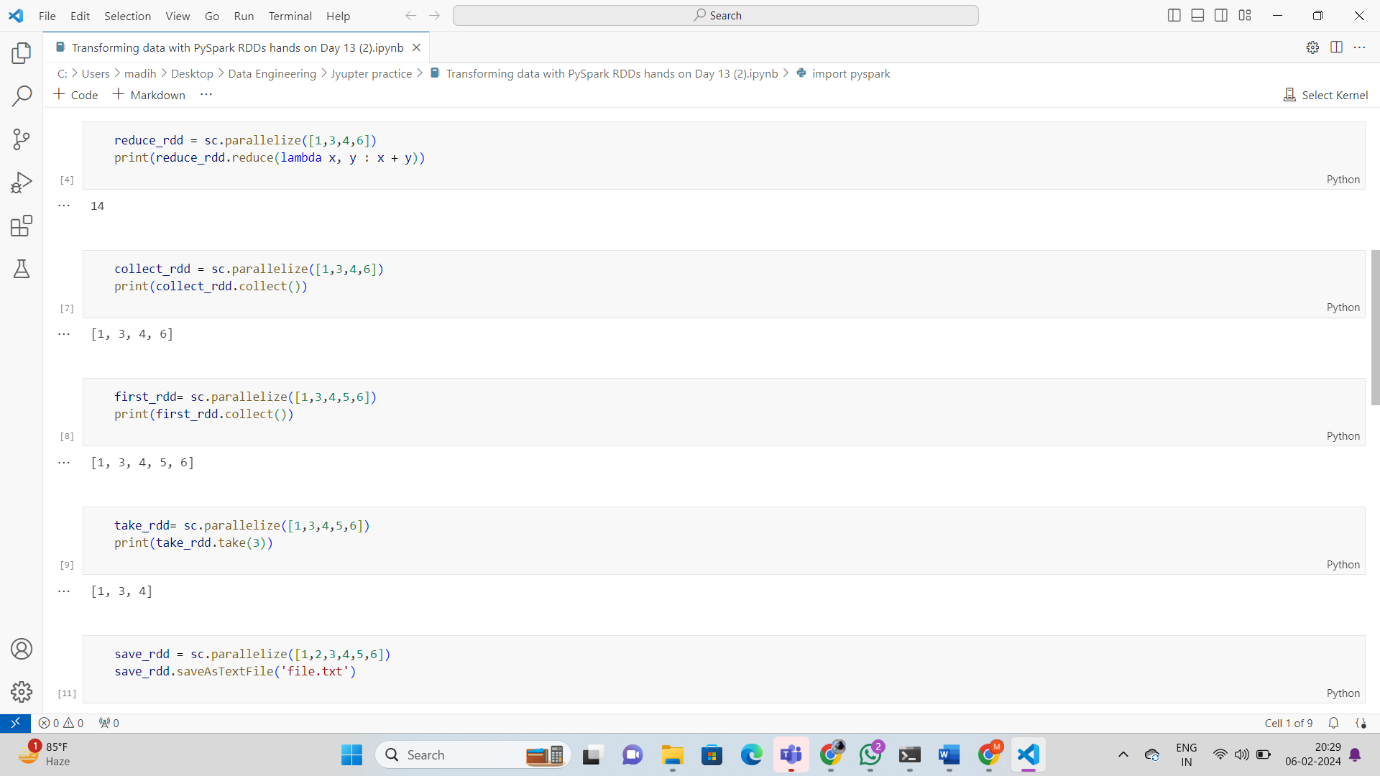
save\_rdd.saveAsTextFile('file.txt')

On executing this code, we get:



Here, we created an RDD, **save\_rdd**using the **.parallelize()** method of SparkContext. We used the **.saveAsTextFile()** action on **save\_rdd**to save it into our directory with the name passed as an argument in it as a string type. The **.saveAsTextFile()** generates a directory with the given argument. Inside the directory, several parts of the file will be created based on the size of the file.



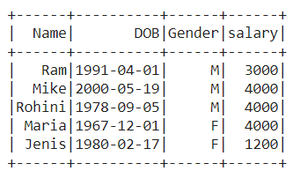


**Selecting, Renaming, Filtering Data in a Pandas DataFrame**

**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.*

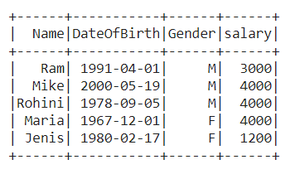
**Example 1:**Renaming the single column in the data frame

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:**

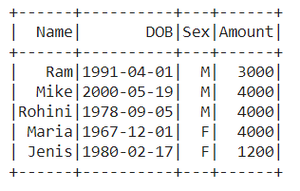


**Example 2:** Renaming multiple column names

**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 :***

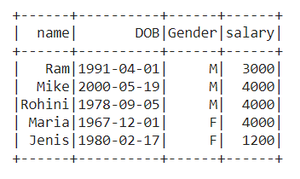
***expr :****It’s an**SQL expression.*

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.*

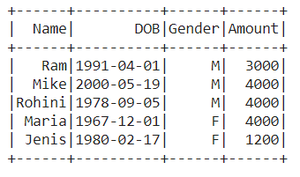
***Return type:****Selects the cols in the dataframe and returns a new DataFrame.*

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)*

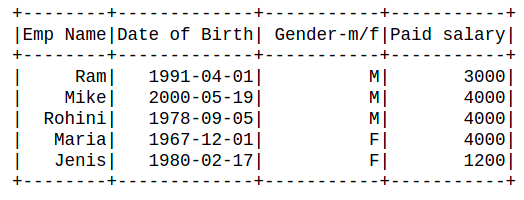
*Where, col is a new column name*

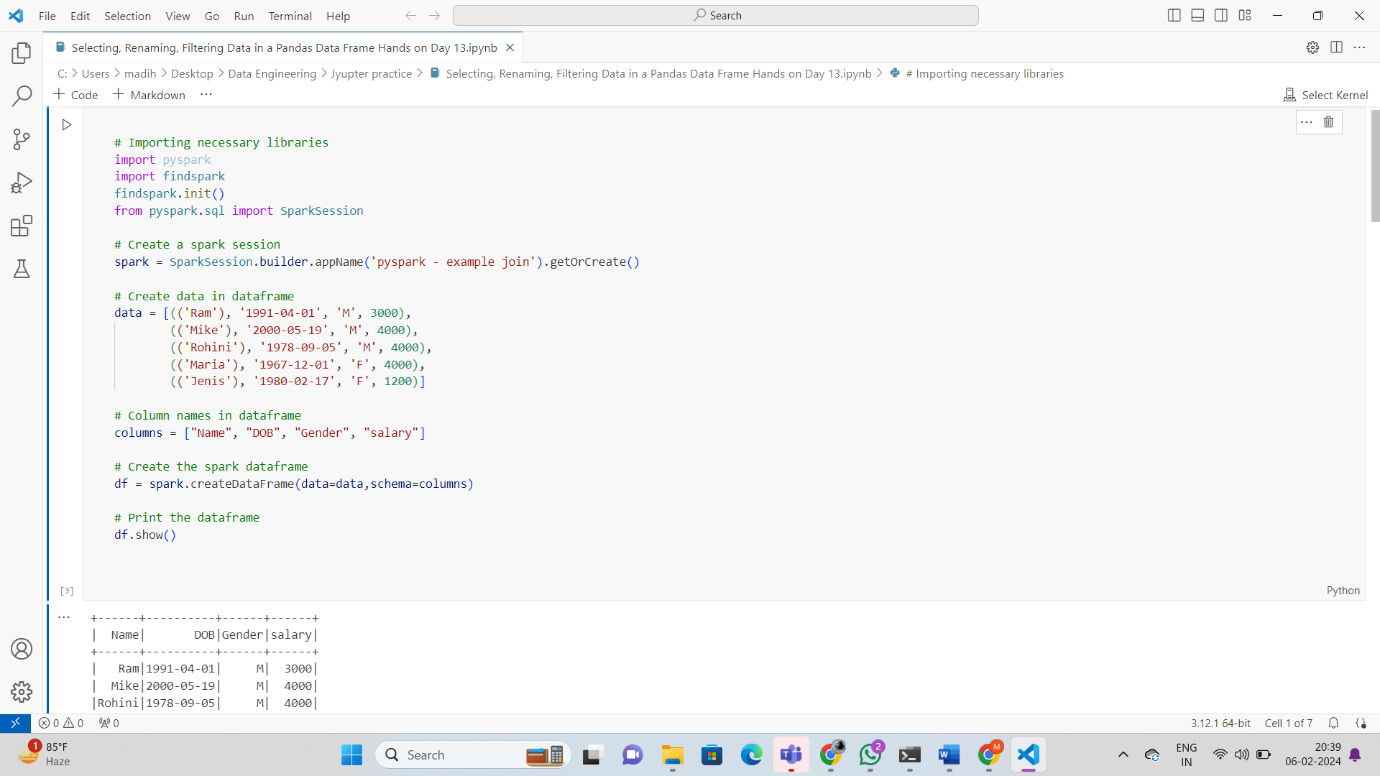
In this example, we will create an order list of new column names and pass it into toDF function

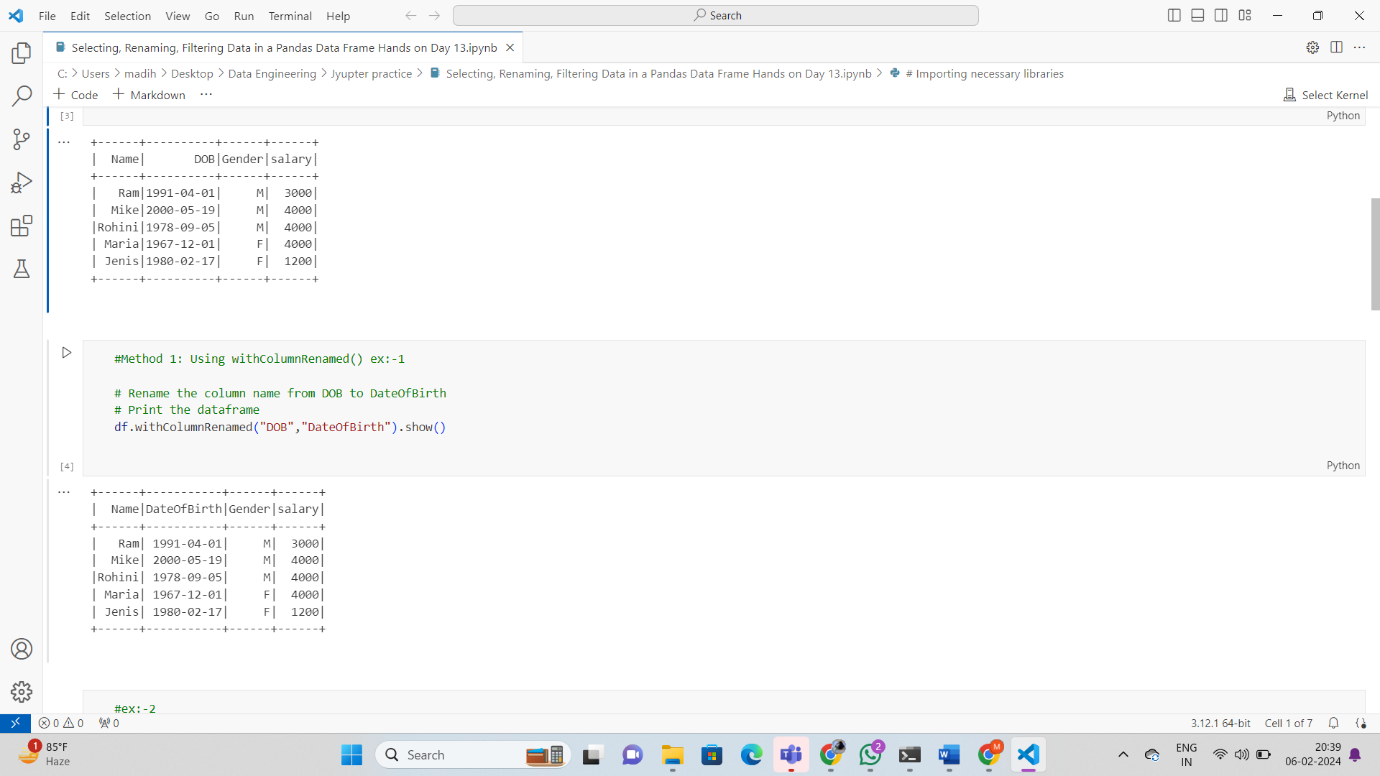
**Python3**

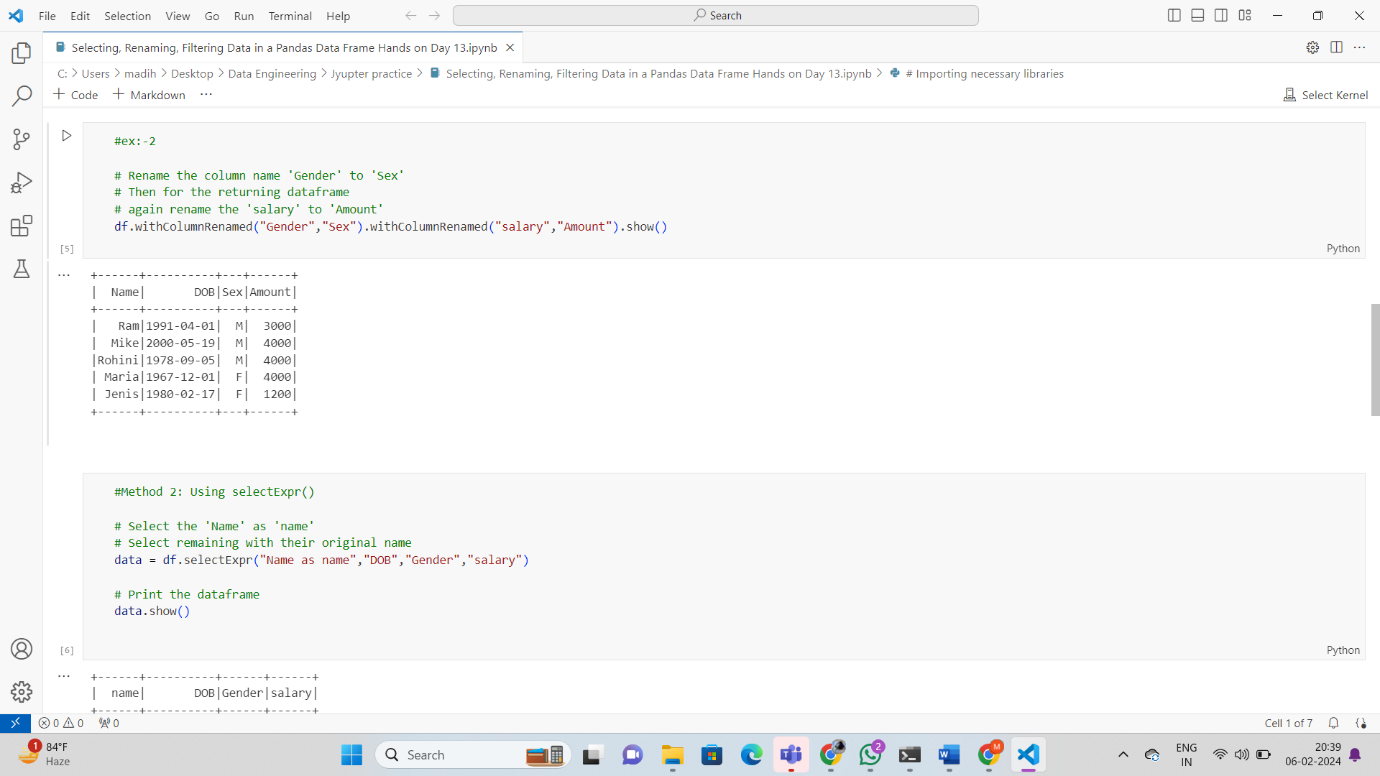
|  |
| --- |
| Data\_list **=** ["Emp Name","Date of Birth",               " Gender-m/f","Paid salary"]    new\_df **=** df.toDF(**\***Data\_list)  new\_df.show() |

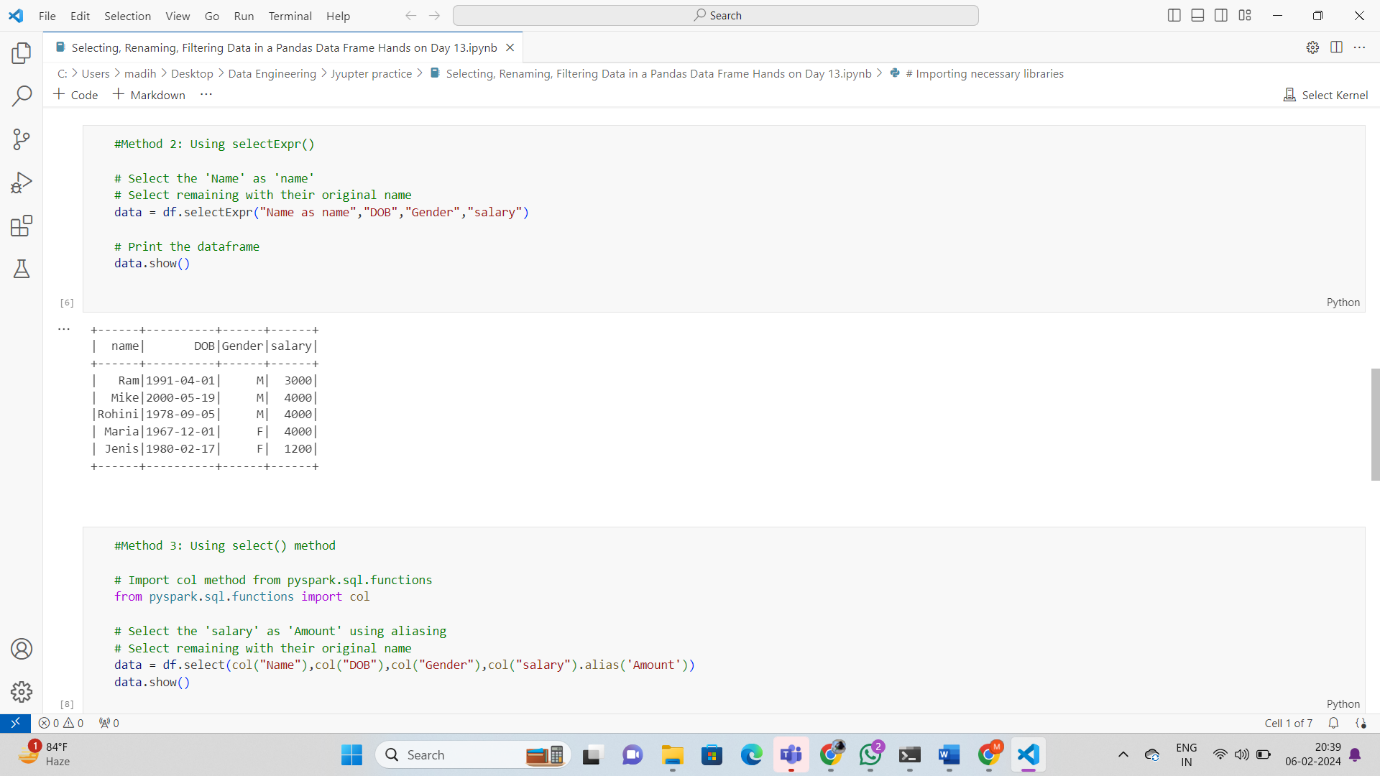
**Output:**

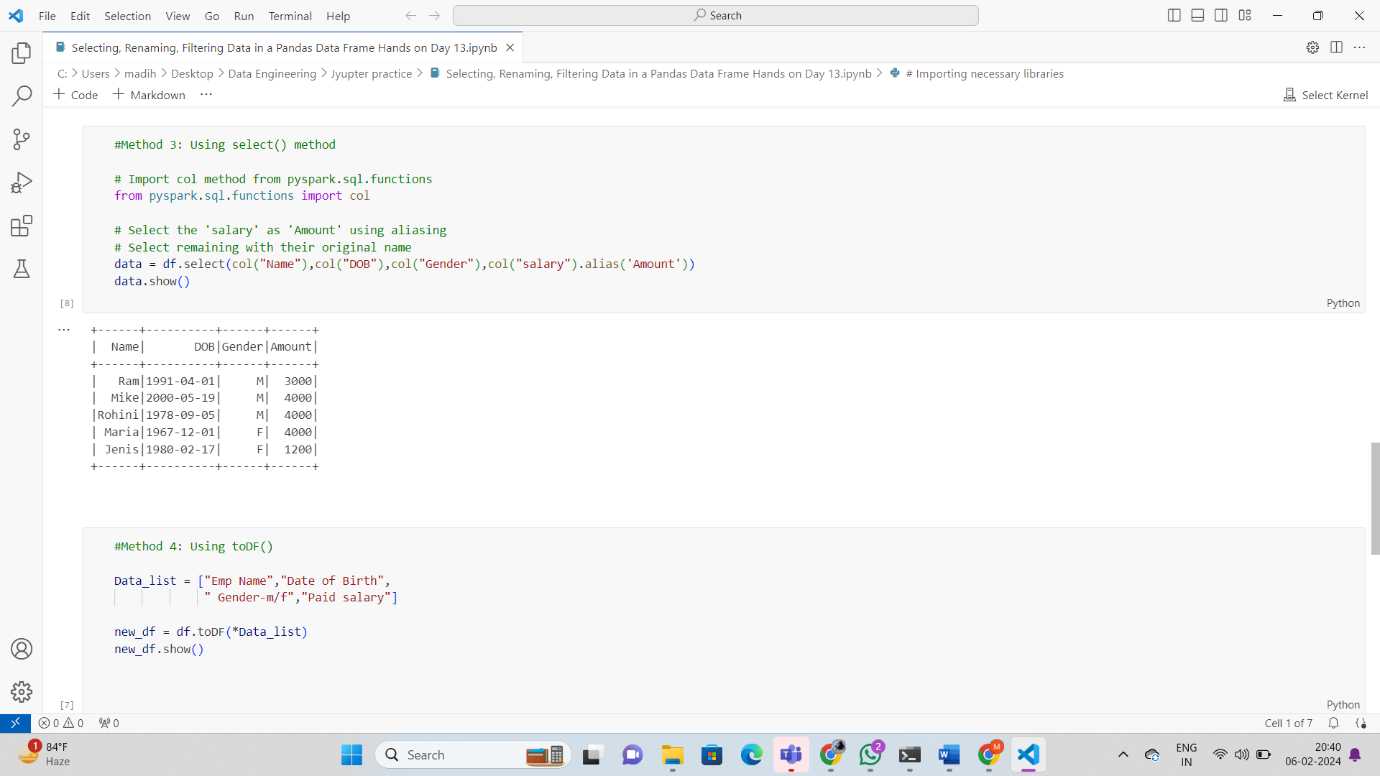


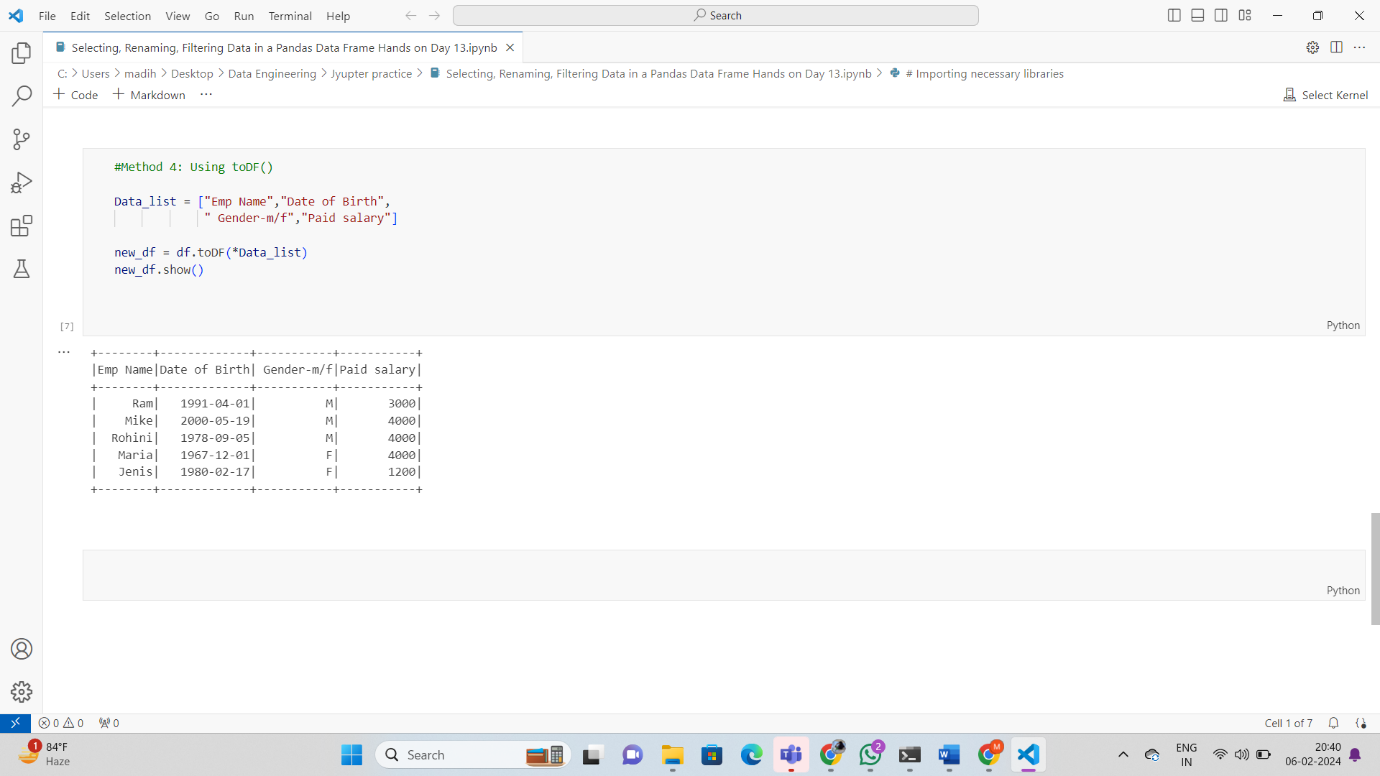












**Notes: -**

