CREATION OF RDD AND DATAFRAME IN PYSPARK

Create RDD

- ***Resilient Distributed Datasets (RDD)*** - "Resilient Distributed Datasets" (RDD) is the fundamental "Data Structure" of Apache Spark. RDDs are "Immutable Distributed Collection of Objects" and "Fault-Tolerant" in nature. Each "Dataset" in RDD is divided into "Logical Partitions", which may be "Computed" across multiple "Nodes" in a "Cluster". Operations can be performed in "Parallel" on RDDs.
  RDDs are called "Resilient" because, these have the ability to always "re-compute" an RDD, when a "Node" fails.
- Following are the different ways to create a "PySpark RDD" -
  - ***Create Spark RDD from Sequence or List Using parallelize ()*** - RDDs are generally created by "Parallelized Collection", i.e., by taking an existing "Collection" from "Driver Program" ("Scala", "Python" etc.), and, passing it to SparkContext's "parralelize ()" method.
    This method is used only for testing, but, not in real time, as the entire Data will reside on one "Node", which is not ideal for "Production".
    - ✓ ***parallelize ()*** - PySpark "parallelize ()" is a "Method" in "SparkContext", and, is used to create an RDD from a "List" collection.

```
employeeColumns = ["Employee_Id", "First_Name", "Last_Name", "House_No", "Address", "City", "Pincode"]

employeeList = [\
                (1001, "Oindrila", "CHakraborty", "118/H", "Narikel Danga North Road", "Kolkata", 700011),\
                (1002, "Soumyajyoti", "Bagchi", "38", "Dhakuria East Road", "Kolkata", 700078),\
                (1003, "Oishi", "Bhattacharyya", "28B", "M.G Road", "Pune", 411009),\
                (1004, "Sabarni", "Chakraborty", "109A", "Ramkrishna Road", "Kolkata", 700105)\
             ]
```

```
employeeRdd = spark.sparkContext.parallelize(employeeList)
print(type(employeeRdd))
```

Output -

```
<class 'pyspark.rdd.RDD'>
```

The "parallelize () Method" also has another "Signature", which additionally takes "Integer Argument" to specify the "Number of Partitions". "Partitions" are "Basic Units of Parallelism" in "PySpark".

```
employeeWithPartitionRdd = spark.sparkContext.parallelize(employeeList, 6)
print("Number of Partitions: " + str(employeeWithPartitionRdd.getNumPartitions()))
```

Output -

```
Number of Partitions: 6
```

Sometimes an “Empty RDD” needs to be created. An “Empty RDD” can be created in the following two ways -

- The “parallelize () Method” in “SparkContext” can be used in order to create an “Empty RDD”. An “Empty List” needs to be passed to the “parallelize () Function”.

```
emptyRddParallelize = spark.sparkContext.parallelize([])
print("Is 'emptyRddParallelize' Empty: " + str(emptyRddParallelize.isEmpty()))
```

Output -

```
Is 'emptyRddParallelize' Empty: True
```

- The “emptyRDD () Method” in “SparkContext” can be used in order to create an “Empty RDD”.

```
emptyRdd = spark.sparkContext.emptyRDD()
print("Is 'emptyRdd' Empty: " + str(emptyRdd.isEmpty()))
```

Output -

```
Is 'emptyRdd' Empty: True
```

➢ ***Create Spark RDD from Text File*** - Mostly for production systems, RDDs are created from “Files”. Suppose following is the content of a “Text File” -

```
Hi All,
This is Oindrila Chakraborty Bagchi.
My address is - 111/T ABC Road, 2nd Floor, Kolkata - 700001.
I work at Cognizant Technology Solutions.
My Office is at Sector 5, Salt Lake City, Kolkata.
```

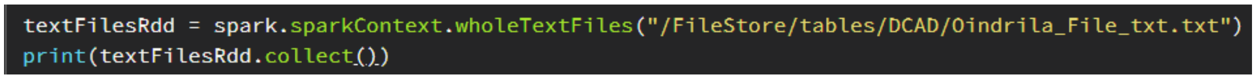
✓ ***textFile ()*** - The “textFile () Method” of “SparkContext” creates an RDD for which each “Record” represents a “Line” in a “File”.

```
textFileRdd = spark.sparkContext.textFile("/FileStore/tables/DCAD/Oindrila_File_txt.txt")
for text in textFileRdd.collect():
  print(text)
```

Output -

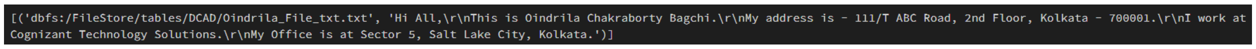
```
Hi All,
This is Oindrila Chakraborty Bagchi.
My address is - 111/T ABC Road, 2nd Floor, Kolkata - 700001.
I work at Cognizant Technology Solutions.
My Office is at Sector 5, Salt Lake City, Kolkata.
```

✓ ***wholeTextFiles ()*** - To read the entire content of a “File” as a “Single Record”, the “wholeTextFiles () method” of “SparkContext” is used.



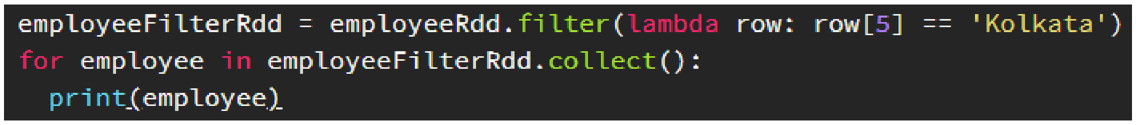
```
textFilesRdd = spark.sparkContext.wholeTextFiles("/FileStore/tables/DCAD/Oindrila_File_txt.txt")
print(textFilesRdd.collect())
```

Output -



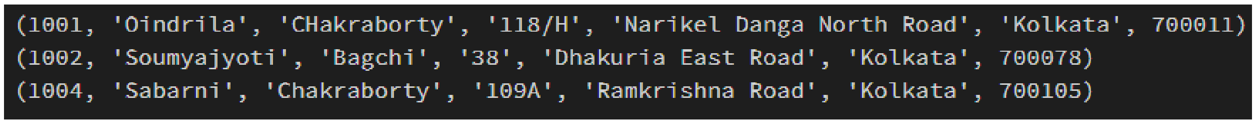
```
[('dbfs:/FileStore/tables/DCAD/Oindrila_File_txt.txt', 'Hi All,\r\nThis is Oindrila Chakraborty Bagchi.\r\nMy address is - 111/T ABC Road, 2nd Floor, Kolkata - 700001.\r\nI work at
Cognizant Technology Solutions.\r\nMy Office is at Sector 5, Salt Lake City, Kolkata.')]
```

➢ ***Create Spark RDD from Another Spark RDD*** - “Transformation Methods” can be used to create a new RDD from an existing one.



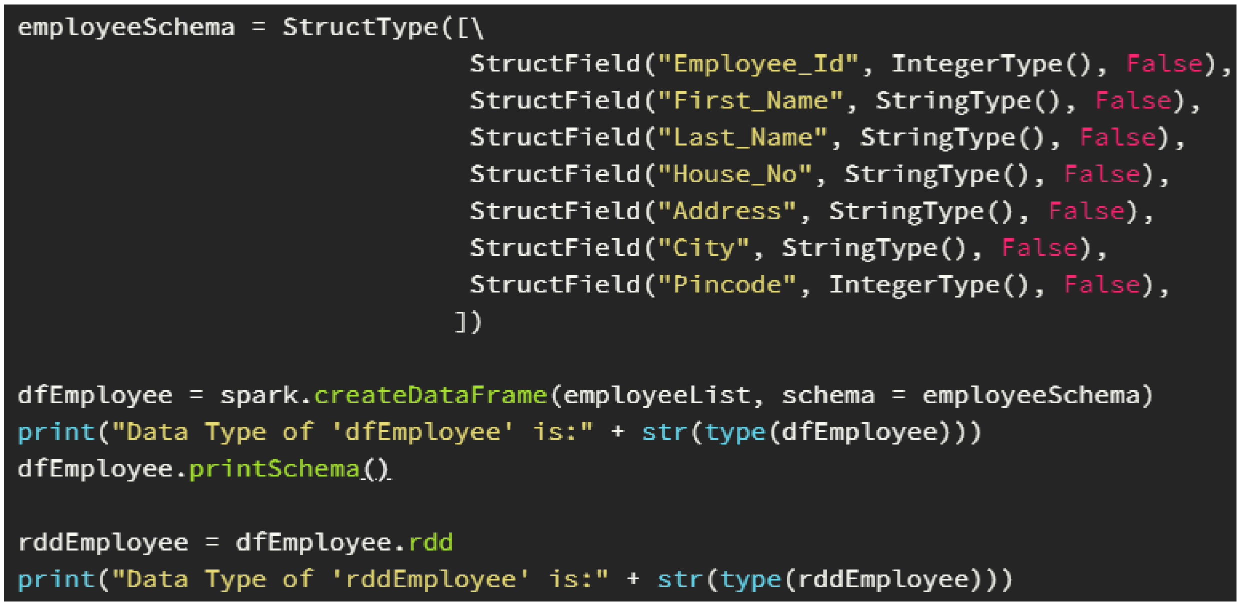
```
employeeFilterRdd = employeeRdd.filter(lambda row: row[5] == 'Kolkata')
for employee in employeeFilterRdd.collect():
  print(employee)
```

Output -



```
(1001, 'Oindrila', 'CHakraborty', '118/H', 'Narikel Danga North Road', 'Kolkata', 700011)
(1002, 'Soumyajyoti', 'Bagchi', '38', 'Dhakuria East Road', 'Kolkata', 700078)
(1004, 'Sabarni', 'Chakraborty', '109A', 'Ramkrishna Road', 'Kolkata', 700105)
```

➢ ***Create Spark RDD from Existing Spark DataFrame / Dataset*** - To convert a “DataFrame”, or, a “Dataset” to an RDD, the “rdd () Method” needs to be used on any of these “Data Types”.



```
employeeSchema = StructType([\
                              StructField("Employee_Id", IntegerType(), False),
                              StructField("First_Name", StringType(), False),
                              StructField("Last_Name", StringType(), False),
                              StructField("House_No", StringType(), False),
                              StructField("Address", StringType(), False),
                              StructField("City", StringType(), False),
                              StructField("Pincode", IntegerType(), False),
                             ])

dfEmployee = spark.createDataFrame(employeeList, schema = employeeSchema)
print("Data Type of 'dfEmployee' is:" + str(type(dfEmployee)))
dfEmployee.printSchema()

rddEmployee = dfEmployee.rdd
print("Data Type of 'rddEmployee' is:" + str(type(rddEmployee)))
```

Output -

```
Data Type of 'dfEmployee' is:<class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- Employee_Id: integer (nullable = false)
 |-- First_Name: string (nullable = false)
 |-- Last_Name: string (nullable = false)
 |-- House_No: string (nullable = false)
 |-- Address: string (nullable = false)
 |-- City: string (nullable = false)
 |-- Pincode: integer (nullable = false)

Data Type of 'rddEmployee' is:<class 'pyspark.rdd.RDD'>
```

Create PySpark DataFrame

- Following are the different ways to create a "PySpark DataFrame" -
  - ***Create DataFrame from RDD*** - One easy way to manually create "PySpark DataFrame" is from an existing RDD.
    - ✓ ***toDF ()*** - The "toDF () Method" of "PySpark RDD" is used to create a "DataFrame" from an existing RDD. Since, RDD doesn't have "Columns", the "DataFrame" is created with default "Column Names", like "_1", "_2" etc.

```
employeeDf = employeeRdd.toDF()
employeeDf.printSchema()
```

Output -

```
root
 |-- _1: long (nullable = true)
 |-- _2: string (nullable = true)
 |-- _3: string (nullable = true)
 |-- _4: string (nullable = true)
 |-- _5: string (nullable = true)
 |-- _6: string (nullable = true)
 |-- _7: long (nullable = true)
```

To create a "DataFrame" with specified "Column Names", a "List" containing the "Column Names" can be provided to the "schema" Property the of the "toDF () Method".

```
employeeColumns = ["Employee_Id", "First_Name", "Last_Name", "House_No", "Address", "City", "Pincode"]
```

```
employeeColumnNamesDf = employeeRdd.toDF(schema = employeeColumns)
employeeColumnNamesDf.printSchema()
```

Output -

```
root
 |-- Employee_Id: long (nullable = true)
 |-- First_Name: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- House_No: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Pincode: long (nullable = true)
```

To create a “DataFrame” with a “Schema”, i.e., “Column Names along with corresponding Data Types”, a “StructType Object of Schema” can be provided to the “schema” Property the of the “toDF () Method”.

```
employeeSchema = StructType([\
                             StructField("Employee_Id", IntegerType(), False),
                             StructField("First_Name", StringType(), False),
                             StructField("Last_Name", StringType(), False),
                             StructField("House_No", StringType(), False),
                             StructField("Address", StringType(), False),
                             StructField("City", StringType(), False),
                             StructField("Pincode", IntegerType(), False),
                            ])

employeeSchemaDf = employeeRdd.toDF(schema = employeeSchema)
employeeSchemaDf.printSchema()
```

Output -

```
root
 |-- Employee_Id: integer (nullable = false)
 |-- First_Name: string (nullable = false)
 |-- Last_Name: string (nullable = false)
 |-- House_No: string (nullable = false)
 |-- Address: string (nullable = false)
 |-- City: string (nullable = false)
 |-- Pincode: integer (nullable = false)
```

- ✓ ***createDataFrame ()*** - The "createDataFrame () Method" from "SparkSession" is another way to create "DataFrame" manually, and, it takes "RDD Object" as an "Argument". Since, RDD doesn't have "Columns", the "DataFrame" is created with default "Column Names", like "_1", "_2" etc.

```
employeeDf = spark.createDataFrame(employeeRdd)
employeeDf.printSchema()
```

Output -

```
root
 |-- _1: long (nullable = true)
 |-- _2: string (nullable = true)
 |-- _3: string (nullable = true)
 |-- _4: string (nullable = true)
 |-- _5: string (nullable = true)
 |-- _6: string (nullable = true)
 |-- _7: long (nullable = true)
```

To create a "DataFrame" with specified "Column Names", a "List" containing the "Column Names" can be provided to the "schema" Property the of the "createDataFrame () Method".

```
employeeColumnNamesDf = spark.createDataFrame(employeeRdd, schema = employeeColumns)
employeeColumnNamesDf.printSchema()
```

Output -

```
root
 |-- Employee_Id: long (nullable = true)
 |-- First_Name: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- House_No: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Pincode: long (nullable = true)
```

To create a "DataFrame" with a "Schema", i.e., "Column Names along with corresponding Data Types", a "StructType Object of Schema" can be provided to the "schema" Property the of the "createDataFrame () Method".

```
employeeSchema = StructType([\
                             StructField("Employee_Id", IntegerType(), False),
                             StructField("First_Name", StringType(), False),
                             StructField("Last_Name", StringType(), False),
                             StructField("House_No", StringType(), False),
                             StructField("Address", StringType(), False),
                             StructField("City", StringType(), False),
                             StructField("Pincode", IntegerType(), False),
                            ])

employeeSchemaDf = spark.createDataFrame(employeeRdd, schema = employeeSchema)
employeeSchemaDf.printSchema()
```

Output -

```
root
 |-- Employee_Id: integer (nullable = false)
 |-- First_Name: string (nullable = false)
 |-- Last_Name: string (nullable = false)
 |-- House_No: string (nullable = false)
 |-- Address: string (nullable = false)
 |-- City: string (nullable = false)
 |-- Pincode: integer (nullable = false)
```

➢ ***Create DataFrame from List Collection*** - “PySpark DataFrame” can be created from a “List Object”.

  ✓ ***createDataFrame ()*** - The “createDataFrame () Method” from “SparkSession” is another way to create “DataFrame” manually, and, it takes “List Object” as an “Argument”. Since, “List” doesn’t have “Columns”, the “DataFrame” is created with default “Column Names”, like “_1”, “_2” etc.

```
employeeDf = spark.createDataFrame(employeeList)
print("Type of 'employeeDf' is : " + str(type(employeeDf)))
employeeDf.printSchema()
```

Output -

```
Type of 'employeeDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- _1: long (nullable = true)
 |-- _2: string (nullable = true)
 |-- _3: string (nullable = true)
 |-- _4: string (nullable = true)
 |-- _5: string (nullable = true)
 |-- _6: string (nullable = true)
 |-- _7: long (nullable = true)
```

To create a “DataFrame” with specified “Column Names”, a “List” containing the “Column Names” can be provided to the “schema” Property the of the “createDataFrame () Method”.

```
employeeColumnNamesDf = spark.createDataFrame(employeeList, schema = employeeColumns)
print("Type of 'employeeColumnNamesDf' is : " + str(type(employeeColumnNamesDf)))
employeeColumnNamesDf.printSchema()
```

Output -

```
Type of 'employeeColumnNamesDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- Employee_Id: long (nullable = true)
 |-- First_Name: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- House_No: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Pincode: long (nullable = true)
```

To create a “DataFrame” with a “Schema”, i.e., “Column Names along with corresponding Data Types”, a “StructType Object of Schema” can be provided to the “schema” Property the of the “createDataFrame () Method”.

```
employeeSchema = StructType([\
                              StructField("Employee_Id", IntegerType(), False),
                              StructField("First_Name", StringType(), False),
                              StructField("Last_Name", StringType(), False),
                              StructField("House_No", StringType(), False),
                              StructField("Address", StringType(), False),
                              StructField("City", StringType(), False),
                              StructField("Pincode", IntegerType(), False),
                             ])

employeeSchemaDf = spark.createDataFrame(employeeList, schema = employeeSchema)
print("Type of 'employeeSchemaDf' is : " + str(type(employeeSchemaDf)))
employeeSchemaDf.printSchema()
```

Output -

```
Type of 'employeeSchemaDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- Employee_Id: integer (nullable = false)
 |-- First_Name: string (nullable = false)
 |-- Last_Name: string (nullable = false)
 |-- House_No: string (nullable = false)
 |-- Address: string (nullable = false)
 |-- City: string (nullable = false)
 |-- Pincode: integer (nullable = false)
```

✓ ***createDataFrame () with RowType*** - The “createDataFrame () Method” has another “Signature” in “PySpark”, which takes the “Collection of Row Type” as an “Argument”. First, the “List Object” needs to be converted into “List of Row Objects”. Since, “List” doesn’t have “Columns”, the “DataFrame” is created with default “Column Names”, like “_1”, “_2” etc.

```
employeeRow = map(lambda x: Row(*x), employeeList)
employeeDf = spark.createDataFrame(employeeRow)
employeeDf.printSchema()
```

Output -

```
root
 |-- _1: long (nullable = true)
 |-- _2: string (nullable = true)
 |-- _3: string (nullable = true)
 |-- _4: string (nullable = true)
 |-- _5: string (nullable = true)
 |-- _6: string (nullable = true)
 |-- _7: long (nullable = true)
```

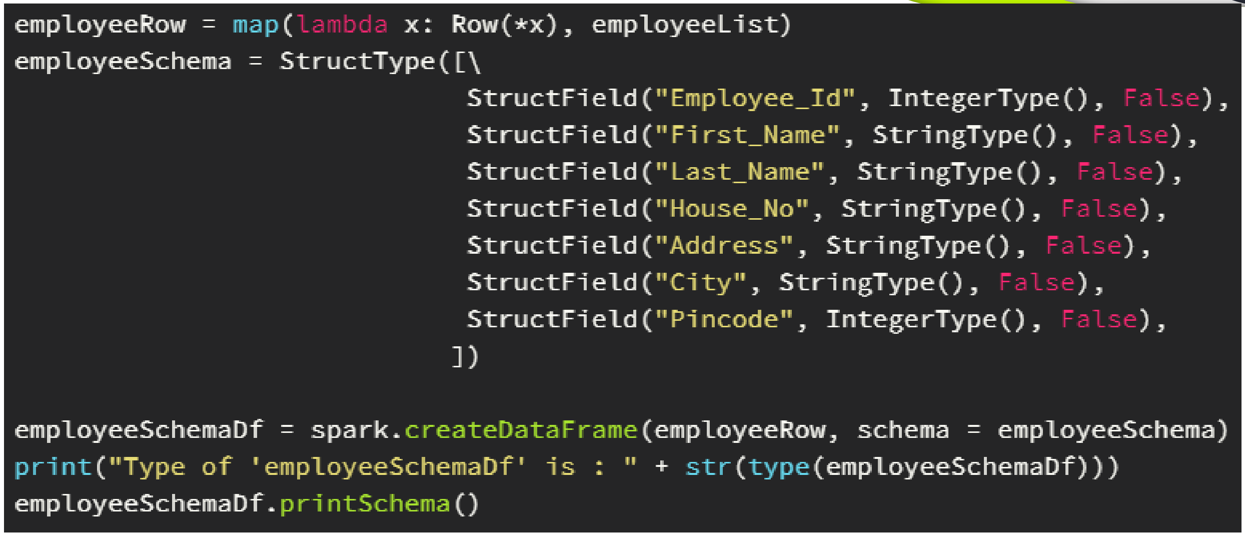
To create a “DataFrame” with specified “Column Names”, a “List” containing the “Column Names” can be provided to the “schema” Property the of the “createDataFrame () Method”.

```
employeeRow = map(lambda x: Row(*x), employeeList)
employeeColumnNamesDf = spark.createDataFrame(employeeRow, schema = employeeColumns)
print("Type of 'employeeColumnNamesDf' is : " + str(type(employeeColumnNamesDf)))
employeeColumnNamesDf.printSchema()
```

Output -

```
Type of 'employeeColumnNamesDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- Employee_Id: long (nullable = true)
 |-- First_Name: string (nullable = true)
 |-- Last_Name: string (nullable = true)
 |-- House_No: string (nullable = true)
 |-- Address: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Pincode: long (nullable = true)
```

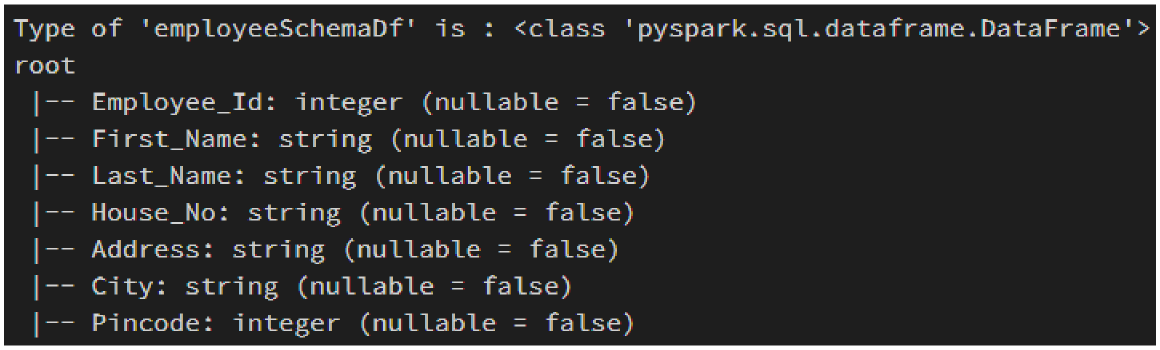
To create a “DataFrame” with a “Schema”, i.e., “Column Names along with corresponding Data Types”, a “StructType Object of Schema” can be provided to the “schema” Property the of the “createDataFrame () Method”.



```
employeeRow = map(lambda x: Row(*x), employeeList)
employeeSchema = StructType([\
                             StructField("Employee_Id", IntegerType(), False),
                             StructField("First_Name", StringType(), False),
                             StructField("Last_Name", StringType(), False),
                             StructField("House_No", StringType(), False),
                             StructField("Address", StringType(), False),
                             StructField("City", StringType(), False),
                             StructField("Pincode", IntegerType(), False),
                            ])

employeeSchemaDf = spark.createDataFrame(employeeRow, schema = employeeSchema)
print("Type of 'employeeSchemaDf' is : " + str(type(employeeSchemaDf)))
employeeSchemaDf.printSchema()
```

Output -



```
Type of 'employeeSchemaDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- Employee_Id: integer (nullable = false)
 |-- First_Name: string (nullable = false)
 |-- Last_Name: string (nullable = false)
 |-- House_No: string (nullable = false)
 |-- Address: string (nullable = false)
 |-- City: string (nullable = false)
 |-- Pincode: integer (nullable = false)
```

➢ ***Create DataFrame from Data Sources*** - In real-time, most “PySpark DataFrames” are created from “Data Source Files”, like “CSV”, “Text”, “JSON”, “XML”, etc. “PySpark” by default supports many “Data Formats” out of the box without importing any “Libraries”, and, to create “DataFrame”, only the appropriate method available in the “DataFrameReader” Class needs to be used.

   ✓ ***csv ()*** - The “csv () Method” of the “DataFrameReader” Class is used to create a “DataFrame” from “CSV File”. Many options can be provided, like - what “Delimiter” to use, whether there is “Quoted Data” in the “CSV File”, “Date Formats” used in the “CSV File”, “infer schema”, and, many more.

```
customerCsvDf = spark.read.\
                          options(\
                                  sep = "|",\
                                  header = True,\
                                  inferSchema = True\
                                 ).\
                          csv("/FileStore/tables/retailer/data/customer.dat")

print("Type of 'customerCsvDf' is : " + str(type(customerCsvDf)))
customerCsvDf.printSchema()
```

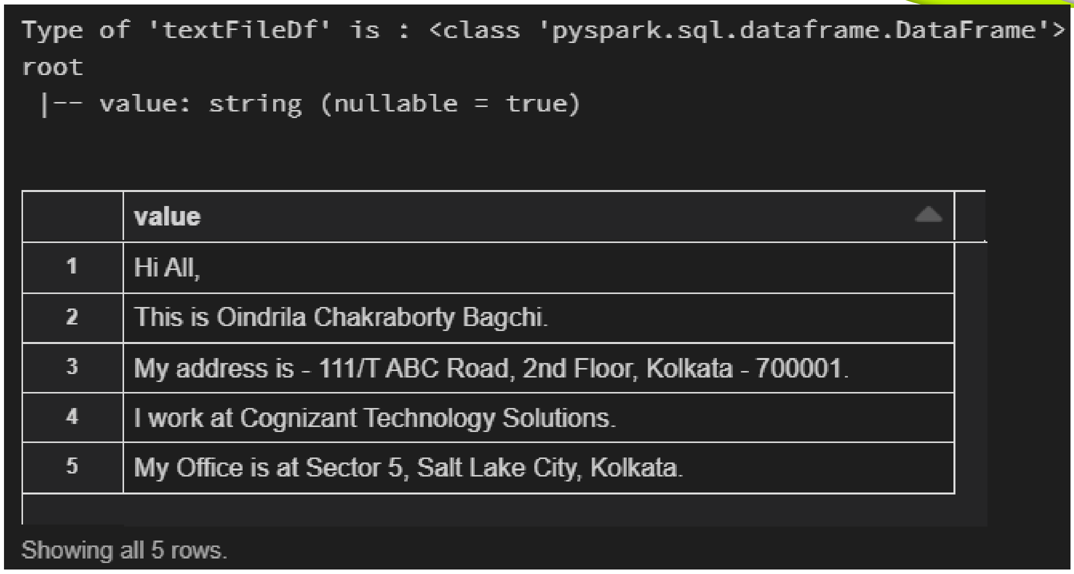
Output -

```
Type of 'customerCsvDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- c_customer_sk: integer (nullable = true)
 |-- c_customer_id: string (nullable = true)
 |-- c_current_cdemo_sk: integer (nullable = true)
 |-- c_current_hdemo_sk: integer (nullable = true)
 |-- c_current_addr_sk: integer (nullable = true)
 |-- c_first_shipto_date_sk: integer (nullable = true)
 |-- c_first_sales_date_sk: integer (nullable = true)
 |-- c_salutation: string (nullable = true)
 |-- c_first_name: string (nullable = true)
 |-- c_last_name: string (nullable = true)
 |-- c_preferred_cust_flag: string (nullable = true)
 |-- c_birth_day: integer (nullable = true)
 |-- c_birth_month: integer (nullable = true)
 |-- c_birth_year: integer (nullable = true)
 |-- c_birth_country: string (nullable = true)
 |-- c_login: string (nullable = true)
 |-- c_email_address: string (nullable = true)
 |-- c_last_review_date: double (nullable = true)
```

✓ ***text ()*** - The “text () Method” of the “DataFrameReader” Class is used to create a “DataFrame” from “Text File”.

```
textFileDf = spark.read.text("/FileStore/tables/DCAD/Oindrila_File_txt.txt")
print("Type of 'textFileDf' is : " + str(type(textFileDf)))
textFileDf.printSchema()
display(textFileDf)
```

Output -

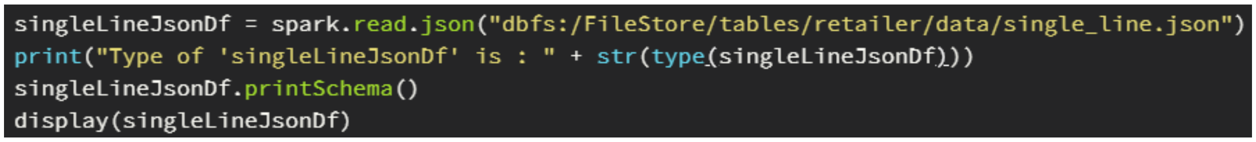


```
Type of 'textFileDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- value: string (nullable = true)
```

	value
1	Hi All,
2	This is Oindrila Chakraborty Bagchi.
3	My address is - 111/T ABC Road, 2nd Floor, Kolkata - 700001.
4	I work at Cognizant Technology Solutions.
5	My Office is at Sector 5, Salt Lake City, Kolkata.

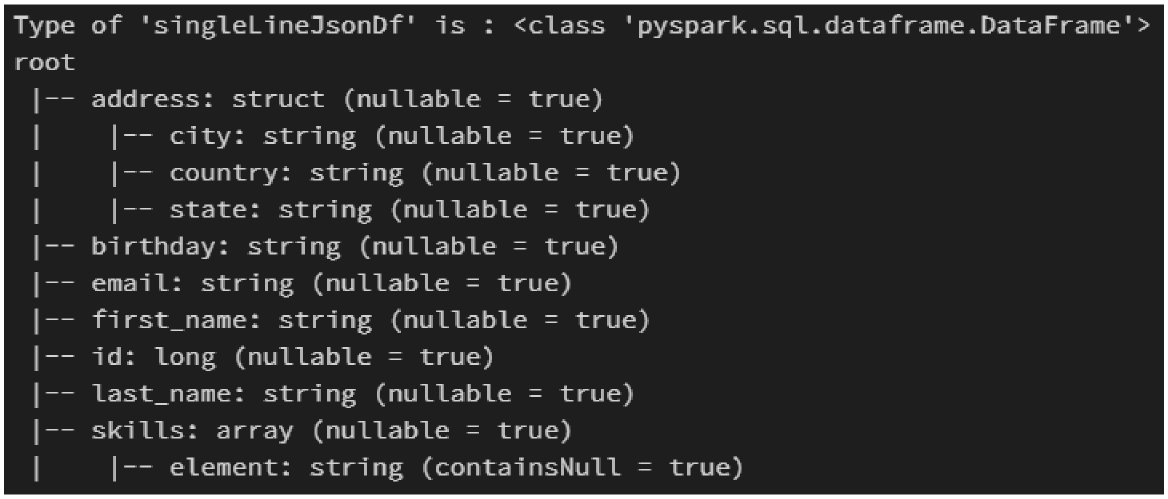
Showing all 5 rows.

- ✓ ***json ()*** - "PySpark" is also used to process "Semi-Structured Data Files", like "JSON Format". The "json () Method" of the "DataFrameReader" Class is used to create a "DataFrame" from "JSON File".

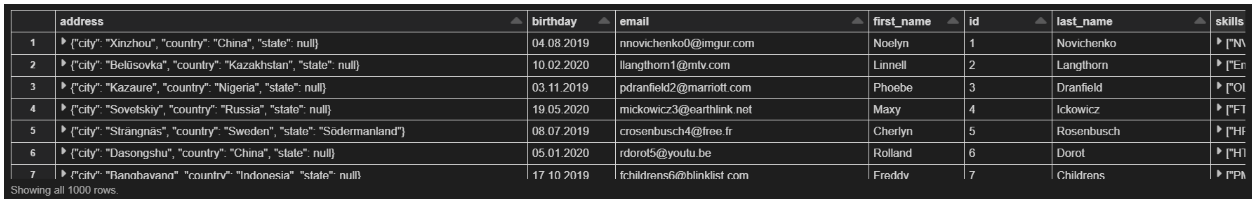


```
singleLineJsonDf = spark.read.json("dbfs:/FileStore/tables/retailer/data/single_line.json")
print("Type of 'singleLineJsonDf' is : " + str(type(singleLineJsonDf)))
singleLineJsonDf.printSchema()
display(singleLineJsonDf)
```

Output -



```
Type of 'singleLineJsonDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- country: string (nullable = true)
 |    |-- state: string (nullable = true)
 |-- birthday: string (nullable = true)
 |-- email: string (nullable = true)
 |-- first_name: string (nullable = true)
 |-- id: long (nullable = true)
 |-- last_name: string (nullable = true)
 |-- skills: array (nullable = true)
 |    |-- element: string (containsNull = true)
```



	address	birthday	email	first_name	id	last_name	skills
1	{"city": "Xinzhou", "country": "China", "state": null}	04.08.2019	nnovichenko0@imgur.com	Noelyn	1	Novichenko	["NV
2	{"city": "Belūsovka", "country": "Kazakhstan", "state": null}	10.02.2020	llangthorn1@mtv.com	Linnell	2	Langthorn	["En
3	{"city": "Kazaure", "country": "Nigeria", "state": null}	03.11.2019	pdranfield2@marriott.com	Phoebe	3	Dranfield	["OL
4	{"city": "Sovetskiy", "country": "Russia", "state": null}	19.05.2020	mickowicz3@earthlink.net	Maxy	4	Ickowicz	["FT
5	{"city": "Strängnäs", "country": "Sweden", "state": "Södermanland"}	08.07.2019	crosenbusch4@free.fr	Cherlyn	5	Rosenbusch	["HF
6	{"city": "Dasongshu", "country": "China", "state": null}	05.01.2020	rdorot5@youtu.be	Rolland	6	Dorot	["HT
7	{"city": "Bangbayang", "country": "Indonesia", "state": null}	17.10.2019	fchildrens6@blinklist.com	Freddy	7	Childrens	["PM

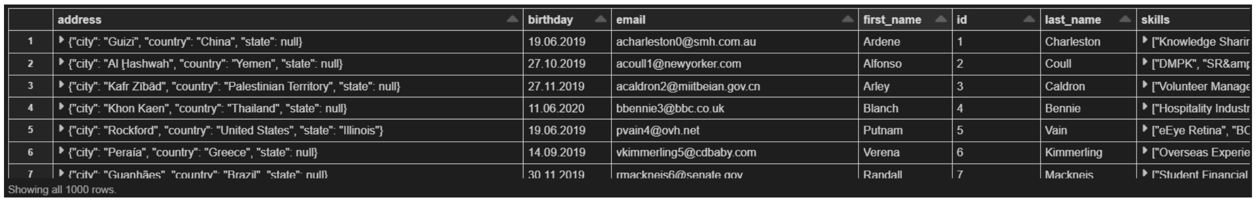
Showing all 1000 rows.

To read data from "Multiline JSON File", the "multiline" Property needs to added to "option", or, "options".

```
multiLineJsonDf = spark.read.option("multiLine", "true").json("dbfs:/FileStore/tables/retailer/data/multi_line.json")
print("Type of 'multiLineJsonDf' is : " + str(type(multiLineJsonDf)))
multiLineJsonDf.printSchema()
display(multiLineJsonDf)
```

Output -

```
Type of 'multiLineJsonDf' is : <class 'pyspark.sql.dataframe.DataFrame'>
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- country: string (nullable = true)
 |    |-- state: string (nullable = true)
 |-- birthday: string (nullable = true)
 |-- email: string (nullable = true)
 |-- first_name: string (nullable = true)
 |-- id: long (nullable = true)
 |-- last_name: string (nullable = true)
 |-- skills: array (nullable = true)
 |    |-- element: string (containsNull = true)
```



	address	birthday	email	first_name	id	last_name	skills
1	{"city": "Guizi", "country": "China", "state": null}	19.06.2019	acharleston0@smh.com.au	Ardene	1	Charleston	["Knowledge Sharir
2	{"city": "Al Ḩashwah", "country": "Yemen", "state": null}	27.10.2019	acoull1@newyorker.com	Alfonso	2	Coull	["DMPK", "SR&amp
3	{"city": "Kafr Zībād", "country": "Palestinian Territory", "state": null}	27.11.2019	acaldron2@miitbeian.gov.cn	Arley	3	Caldron	["Volunteer Manage
4	{"city": "Khon Kaen", "country": "Thailand", "state": null}	11.06.2020	bbennie3@bbc.co.uk	Blanch	4	Bennie	["Hospitality Industr
5	{"city": "Rockford", "country": "United States", "state": "Illinois"}	19.06.2019	pvain4@ovh.net	Putnam	5	Vain	["eEye Retina", "BC
6	{"city": "Peraía", "country": "Greece", "state": null}	14.09.2019	vkimmerling5@cdbaby.com	Verena	6	Kimmerling	["Overseas Experie
7	{"city": "Guanhães", "country": "Brazil", "state": null}	30.11.2019	rmackneis6@senate.gov	Randall	7	Mackneis	["Student Financial

Showing all 1000 rows.

✓ ***<u>Other Sources</u>*** - "DataFrame" can also be created by reading data from "Avro", "Parquet", "ORC", "Binary Files", "Kafka", or, by accessing "Hive", and "HBase" Tables etc.