


Built-in Functions and Looping in Pyspark Dataframes

Table of Contents (you can navigate directly by clicking on this points)

- [Importing libraries](#)
- [Splitting text into Meaningful Columns and Type Casting](#)
- [Filtering Invalid dates](#)
- [Getting current datetime & loading timestamp data and type casting it](#)
- [Getting date, year, month, day, time, hour, minutes, seconds from the timestamp data](#)
- [Working on some Mathematical functions](#)
 - [Min, Max, Mean, Standard Deviation](#)
 - [Round, Absolute, Ceil, Floor](#)
 - [covariance and correlation](#)
 - [describe](#)
 - [expr](#)
 - [approx count distinct](#)
 - [collect list](#)
 - [collect set](#)
 - [count distinct](#)
 - [first](#)
 - [last](#)
 - [kurtosis](#)
 - [skewness](#)
 - [stddev_samp](#)
 - [stddev_pop](#)
 - [sum](#)
 - [sumDistinct](#)
 - [variance, var_samp, var_pop](#)
- [Type Casting and Interpreting various results](#)
- [To check if all the values of a column are integers only or not?](#)
- [For Loop, If condition, Foreach command](#)
 - [Printing data using for loop](#)
 - [For loop and if condition](#)
 - [Foreach command](#)
 - [Creating dataframe in for loop](#)

Note → Arrow  is used to go back to “table of contents”.

Importing libraries



```
from pyspark.sql.types import *  
from pyspark.sql.functions import *
```

Splitting text into Meaningful Columns and Type Casting



Let's first load our data -

```
#!/FileStore/tables/validating_date_data.txt  
data = spark.read.text("dbfs:/FileStore/tables/validating_date_data.txt")  
display(data)
```

▶ (1) Spark Jobs
▶ data: pyspark.sql.dataframe.DataFrame = [value: string]

	value
1	459tet20210112
2	999tew20203112
3	342tre20200112
4	765try20200312

Showing all 4 rows.



Command took 0.45 seconds -- by nidhi.mantri@infosys.com at 12/29/2020, 7:27:44 PM on Mantri

Here, our data is : -

- First 3 characters is number
- Next 3 characters is text
- Last 8 characters is a date(yyyymmdd)

So, let's split the actual data using substring function and type casting the columns

```
import datetime  
splitted_data = (data  
    .withColumn("Number", substring(col("value"), 1,3).cast(IntegerType()))  
    .withColumn("Text", substring(col("value"), 4, 3).cast(StringType()))
```

```
.withColumn("Date", to_date(substring(col("value"), 7, 8), "yyyyMMdd"))
)
```

```
display(splitted_data)
```

substring(col("value"), 1, 3) → first parameter is the column, second parameter is the position (1 based indexing) from which we want to read, and third parameter is number of characters to be read.

to_date(column, format) → first parameter is the column, which we want to type cast and second parameter is the format (in which the date is stored). It will automatically replace dates with wrong month, day by null.

The screenshot shows the Databricks interface. On the left is a sidebar with icons for Data, Clusters, Jobs, and Search. The main area displays a Spark job with the command: `splitted_data: pyspark.sql.dataframe.DataFrame = [value: string, Number: integer ... 2 more fields]`. Below this is a table view of the data:

	value	Number	Text	Date
1	459tet20210112	459	tet	2021-01-12
2	999tew20203112	999	tew	null
3	342tre20200112	342	tre	2020-01-12
4	765try20200312	765	try	2020-03-12

Showing all 4 rows.

Command took 0.47 seconds -- by nidhi.mantri@infosys.com at 12/29/2020, 7:28:38 PM on Mantri

Another Method –

```
splitted_data = (data.withColumn('Id', col('value').substr(1, 3).cast(IntegerType()))
    .withColumn('Name', col('value').substr(4, 3))
    .withColumn('Date', date_format(to_date(col('text').substr(7, 8), "yyyyMMdd"), "yyyy-MM-dd"))
    .drop('value'))
display(splitted_data)
```

The screenshot shows a table view of the data with columns Id, Name, and Date:

	Id	Name	Date
1	459	tet	2021-01-12
2	999	tew	null
3	342	tre	2020-01-12
4	765	try	2020-03-12

Showing all 4 rows.

Here, we use the `substr()` function, it is same as `substring` function. The only difference is the column we provide to it.

When we use `date_format()` function, "Date" is of string type.

Note – check the date of second row(actual data), month is 31 (wrong date), and also the current date is 29/12/2020 that means date of first row is also invalid.

Let's filter the invalid dates.

Filtering invalid dates



Checking –

- Date is not null
- Date is less than or equal to current date (or a particular date)

```
import datetime
```

```
current_date = datetime.date(2020,12,29) #datetime.date.today()
```

```
validated_data = splitted_data.filter((col("Date").isNotNull()) & (col("Date") <= current_date))
```

```
display(validated_data)
```

	value	Number	Text	Date
1	342tre20200112	342	tre	2020-01-12
2	765try20200312	765	try	2020-03-12

Another method –

```
split3=splitted_data.filter(splitted_data.Date!="null") # Here "Date" must be of string type.
```

```
# or splitted_data.dropna()
```

```
current_date=datetime.date.today()
```

```
split4=split3[split3["Date"]<current_date]
```

```
display(split4)
```

	Id	Name	Date
1	342	tre	2020-01-12
2	765	try	2020-03-12

Showing all 2 rows.

Getting current datetime & Loading timestamp data and type casting it



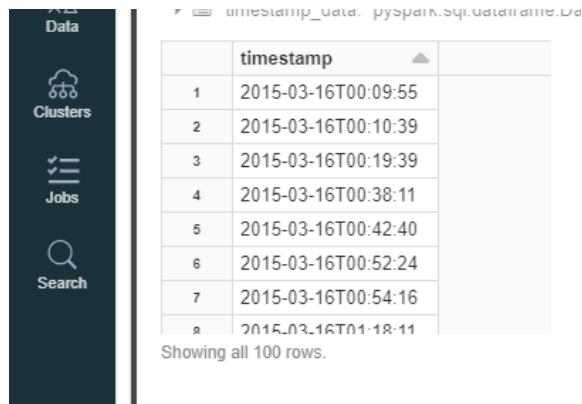
Getting current date and time using "from_utc_timestamp()" function -

```
cdate=spark.sql("select from_utc_timestamp(current_timestamp(),'GMT-5') AS  
your_local_datetime")  
cdate.show(truncate=False)
```

```
+-----+  
|your_local_datetime|  
+-----+  
|2020-12-29 06:16:27.018|  
+-----+
```

Loading a random timestamp data and selecting the timestamp column only.

```
timestamp_data =  
spark.read.csv("dbfs:/FileStore/tables/pageviews_by_second_example.tsv", sep="\t",  
header=True).select("timestamp")  
display(timestamp_data)
```



The screenshot shows the Databricks Data View interface. On the left is a sidebar with icons for Data, Clusters, Jobs, and Search. The main area displays a table with the following data:

	timestamp
1	2015-03-16T00:09:55
2	2015-03-16T00:10:39
3	2015-03-16T00:19:39
4	2015-03-16T00:38:11
5	2015-03-16T00:42:40
6	2015-03-16T00:52:24
7	2015-03-16T00:54:16
8	2015-03-16T01:18:11

Showing all 100 rows.

Type casting the column to timestamp type.

```
timestamp_data = timestamp_data.withColumn("timestamp",  
col("timestamp").cast("timestamp"))  
display(timestamp_data)
```

	timestamp
1	2015-03-16T00:09:55.000+0000
2	2015-03-16T00:10:39.000+0000
3	2015-03-16T00:19:39.000+0000
4	2015-03-16T00:38:11.000+0000
5	2015-03-16T00:42:40.000+0000
6	2015-03-16T00:52:24.000+0000
7	2015-03-16T00:54:16.000+0000
8	2015-03-16T01:18:11.000+0000

Showing all 100 rows.

Getting date, year, month, day, time, hour, minutes, seconds from the timestamp data

```
result_data = (timestamp_data

.withColumn("Date", to_date(col("timestamp"), "yyyy-MM-dd"))

.withColumn("Year", year(col("timestamp")))

.withColumn("Month", month(col("timestamp")))

.withColumn("Day", dayofmonth(col("timestamp")))

.withColumn("Time", concat(hour(col("timestamp")), lit(":"),
minute(col("timestamp")), lit(":"), second(col("timestamp"))))

.withColumn("Hour", hour(col("timestamp")))

.withColumn("Minute", minute(col("timestamp")))

.withColumn("Second", second(col("timestamp")))

)

display(result_data)
```

concat – used to concatenate multiple columns' data

lit – column of literal value

	timestamp	Date	Year	Month	Day	Time	Hour	Minute	Second
1	2015-03-16T00:09:55.000+0000	2015-03-16	2015	3	16	0:9:55	0	9	55
2	2015-03-16T00:10:39.000+0000	2015-03-16	2015	3	16	0:10:39	0	10	39
3	2015-03-16T00:19:39.000+0000	2015-03-16	2015	3	16	0:19:39	0	19	39
4	2015-03-16T00:38:11.000+0000	2015-03-16	2015	3	16	0:38:11	0	38	11
5	2015-03-16T00:42:40.000+0000	2015-03-16	2015	3	16	0:42:40	0	42	40
6	2015-03-16T00:52:24.000+0000	2015-03-16	2015	3	16	0:52:24	0	52	24
7	2015-03-16T00:54:16.000+0000	2015-03-16	2015	3	16	0:54:16	0	54	16
8	2015-03-16T01:18:11.000+0000	2015-03-16	2015	3	16	1:18:11	1	18	11

Showing all 100 rows.

Working on some mathematical functions

Creating a random data →



```
data = sqlContext.range(0, 10)

data_df = data.select("id", rand(seed=0).alias("uniform_data(ud)"),
randn(seed=0).alias("normal_data(nd)"))

display(data_df)
```

seed – to have same data every time.

▶ (3) Spark Jobs

▶ data_df: pyspark.sql.dataframe.DataFrame = [id: long, uniform_data(ud): double ... 1 more fields]

	id	uniform_data(ud)	normal_data(nd)
1	0	0.7604953758285915	1.6034991609278433
2	1	0.6363787615254752	1.6845611254444919
3	2	0.5311207224659675	0.2637682686300013
4	3	0.25738143505962285	-1.1854930781734352
5	4	0.6698885713796182	0.8301167121353836
6	5	0.9531453492357947	-1.1081822375859998
7	6	0.02390696427502892	1.5298496477243015
8	7	0.07039836716341774	-0.24587658470592705

Showing all 10 rows.

Min, Max, Mean and Standard Deviation of the data -



```
#ud --> uniform_data(ud)
#nd --> normal_data(nd)

print("Min, Max, Mean and Standard Deviation - ")

data_df.select(min('uniform_data(ud)').alias("min(ud)"),
               max("normal_data(nd)").alias("max(nd)"),
               mean("uniform_data(ud)").alias("mean(ud)"),
               stddev("normal_data(nd)").alias("standardDeviation(nd)"),
               ).show()
```

▶ (2) Spark Jobs

Min, Max, Mean and Standard Deviation -

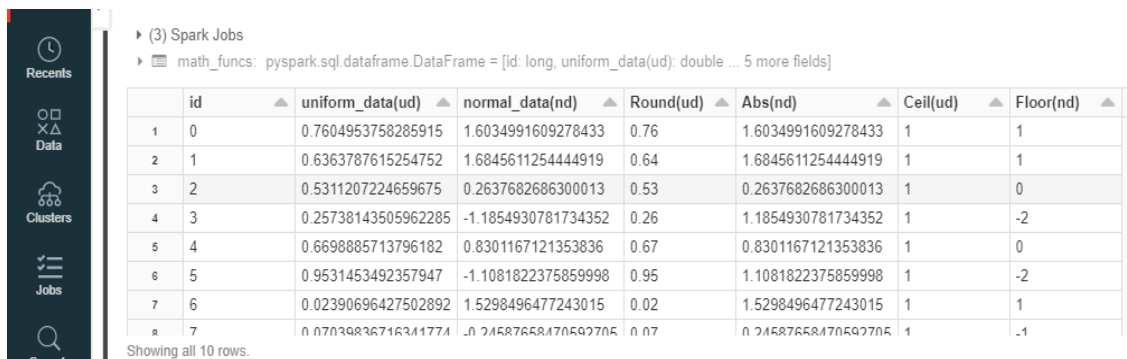
min(ud)	max(nd)	mean(ud)	standardDeviation(nd)
0.02390696427502892	1.6845611254444919	0.46445849521329385	1.2601578153233755

Round, Absolute, Ceil, Floor -



```
math_funcs = (data_df
    .withColumn("Round(ud)", round(col("uniform_data(ud)"), 2))
    .withColumn("Abs(nd)", abs(col("normal_data(nd)")))
    .withColumn("Ceil(ud)", ceil(col("uniform_data(ud)")))
    .withColumn("Floor(nd)", floor(col("normal_data(nd)")))
)

display(math_funcs)
```



(3) Spark Jobs

math_funcs: pyspark.sql.dataframe.DataFrame = [id: long, uniform_data(ud): double ... 5 more fields]

	id	uniform_data(ud)	normal_data(nd)	Round(ud)	Abs(nd)	Ceil(ud)	Floor(nd)
1	0	0.7604953758285915	1.6034991609278433	0.76	1.6034991609278433	1	1
2	1	0.6363787615254752	1.6845611254444919	0.64	1.6845611254444919	1	1
3	2	0.5311207224659675	0.2637682686300013	0.53	0.2637682686300013	1	0
4	3	0.25738143505962285	-1.1854930781734352	0.26	1.1854930781734352	1	-2
5	4	0.6698885713796182	0.8301167121353836	0.67	0.8301167121353836	1	0
6	5	0.9531453492357947	-1.1081822375859998	0.95	1.1081822375859998	1	-2
7	6	0.02390696427502892	1.5298496477243015	0.02	1.5298496477243015	1	1
8	7	0.07034836716341774	-0.24587658470592705	0.07	0.24587658470592705	1	-1

Showing all 10 rows.

One more example of round function -

```
df = spark.createDataFrame(
    [(0.0, 0.2, 3.45631),
     (0.4, 1.4, 2.82945),
     (0.5, 1.9, 7.76261),
     (0.6, 0.9, 2.76790),
     (1.2, 1.0, 9.87984)],
    ["col1", "col2", "col3"])

df.show()
```



```
(7, 'Maggi', 77, 42, 34, 78),]
, ['id', 'Name', 'English', 'Maths', 'Science', 'Computer'])
```

```
markDf.show()
```

```
+-----+-----+-----+-----+
| id| Name|English|Maths|Science|Computer|
+-----+-----+-----+-----+
| 1| Abin| 88| 76| 82| 91|
| 2| Annu| 86| 94| 89| 90|
| 3| Don| 75| 83| 93| 65|
| 4| Tessy| 63| 33| 74| 83|
| 5| Steev| 82| 54| 97| 65|
| 6| Alan| 82| 89| 90| 90|
| 7| Maggi| 77| 42| 34| 78|
+-----+-----+-----+-----+
```

describe() function –



```
markDf.describe('English', 'Maths', 'Science', 'Computer').show()
```

```
+-----+-----+-----+-----+
|summary| English| Maths| Science| Computer|
+-----+-----+-----+-----+
| count| 7| 7| 7| 7|
| mean| 79.0| 67.28571428571429| 79.85714285714286| 80.28571428571429|
| stddev| 8.406346808612328| 24.150322880617875| 21.582620874433296| 11.426785574754984|
| min| 63| 33| 34| 65|
| max| 88| 94| 97| 91|
+-----+-----+-----+-----+
```

expr() function –



```
totDF = markDf.withColumn('Total', expr("English + Maths + Science + Computer"))
```

```
totDF.show()
```

```
+-----+-----+-----+-----+-----+
| id| Name|English|Maths|Science|Computer|Total|
+-----+-----+-----+-----+-----+
| 1| Abin| 88| 76| 82| 91| 337|
| 2| Annu| 86| 94| 89| 90| 359|
| 3| Don| 75| 83| 93| 65| 316|
| 4| Tessy| 63| 33| 74| 83| 253|
| 5| Steev| 82| 54| 97| 65| 298|
| 6| Alan| 82| 89| 90| 90| 351|
| 7| Maggi| 77| 42| 34| 78| 231|
+-----+-----+-----+-----+-----+
```

approx_count_distinct() function –



```
print("Approx_count_distinct: " + \
```

```
str(markDf.select(approx_count_distinct("English")).collect()[0][0]))
```

```
Approx_count_distinct: 6
```

collect_list() function –



```
markDf.select(collect_list("Name")).show(truncate=False)
```

```
+-----+
|collect_list(Name)      |
+-----+
|[Abin, Annu, Don, Tessy, Steev, Alan, Maggi]|
+-----+
```

collect_set() function –



```
markDf.select(collect_set("Maths")).show(truncate=False)
```

```
+-----+
|collect_set(Maths)      |
+-----+
|[33, 89, 83, 42, 54, 76, 94]|
+-----+
```

count_distinct() function –



```
newDf = markDf.select(countDistinct("English", "Science"))
```

```
newDf.show(truncate=False)
```

```
print("Distinct Count of English & Science: "+str(newDf.collect()[0][0]))
```

```
+-----+
|count(DISTINCT English, Science)|
+-----+
|7                                |
+-----+
```

```
Distinct Count of English & Science: 7
```

first() function –



```
markDf.select(first("Name")).show(truncate=False)
```

```
+-----+
|first(Name) |
+-----+
|Abin        |
+-----+
```

last() function –



```
markDf.select(last("Name")).show(truncate=False)
```

```
+-----+  
| last(Name) |  
+-----+  
| Maggi      |  
+-----+
```

kurtosis() function –



```
totDf.select(kurtosis("Total")).show(truncate=False)
```

```
+-----+  
| kurtosis(Total) |  
+-----+  
| -1.2090918177813672 |  
+-----+
```

skewness() function –



```
totDf.select(skewness("Total")).show(truncate=False)
```

```
+-----+  
| skewness(Total) |  
+-----+  
| -0.48880666999748196 |  
+-----+
```

stddev_samp() function – (sample standard deviation)



```
totDf.select(stddev_samp("Total")).show(truncate=False)
```

```
+-----+  
| stddev_samp(Total) |  
+-----+  
| 48.97569854140977 |  
+-----+
```

stddev_pop() function – (population standard deviation)



```
totDf.select(stddev_pop("Total")).show(truncate=False)
```

```
+-----+  
| stddev_pop(Total) |  
+-----+  
| 45.34268611003839 |  
+-----+
```

sum() function –



```
markDf.select(sum("English")).show(truncate=False)
```

```
+-----+
|sum(English)|
+-----+
|553         |
+-----+
```

sumDistinct() function –



```
markDf.select(sumDistinct("English")).show(truncate=False)
```

```
+-----+
|sum(DISTINCT English)|
+-----+
|471                   |
+-----+
```

variance(), var_samp() and var_pop() functions –



```
totDf.select(variance("Total"),var_samp("Total"),var_pop("Total")).show(truncate=False)
```

```
+-----+-----+-----+
|var_samp(Total)|var_samp(Total)|var_pop(Total)|
+-----+-----+-----+
|2398.619047619047|2398.619047619047|2055.9591836734685|
+-----+-----+-----+
```

Type Casting and Interpreting various results



(data_df

```
.withColumn("DecimalType(ud)", col("uniform_data(ud)").cast(DecimalType()))
.withColumn("FloatType(nd)", col("normal_data(nd)").cast(FloatType()))
.withColumn("IntType(ud)", col("uniform_data(ud)").cast(IntegerType()))).show()
```

Recent						
○□ Data						
☁ Clusters						
✓ Jobs						
🔍 Search						

(3) Spark Jobs						
id	uniform_data(ud)	normal_data(nd)	DecimalType(ud)	FloatType(nd)	IntType(ud)	
0	0.7604953758285915	1.6034991609278433		1	1.6034992	0
1	0.6363787615254752	1.6845611254444919		1	1.6845611	0
2	0.5311207224659675	0.2637682686300013		1	0.26376826	0
3	0.25738143505962285	-1.1854930781734352		0	-1.1854931	0
4	0.6698885713796182	0.8301167121353836		1	0.8301167	0
5	0.9531453492357947	-1.1081822375859998		1	-1.1081822	0
6	0.02390696427502892	1.5298496477243015		0	1.5298496	0
7	0.07039836716341774	-0.24587658470592705		0	-0.24587658	0
8	0.28188121484885875	-1.773755556110447		0	-1.7737556	0
9	0.45998819035056326	-0.32537852999017897		0	-0.32537854	0

Command took 0.50 seconds -- by nidhi.mantri@infosys.com at 12/31/2020, 9:32:51 PM on Mantri

To check if all the values of a column are integers only or not?



Creating a dataframe –

```
string_int_data = spark.createDataFrame([("123nm",), ("12345",), ("789pr",), ("456jm",), ("56890",)], ["Id",])
display(string_int_data)
```

Recent	
○□ Data	
☁ Clusters	
✓ Jobs	
🔍 Search	

(3) Spark Jobs	
string_int_data: pyspark.sql.dataframe.DataFrame = [Id: string]	

	Id
1	123nm
2	12345
3	789pr
4	456jm
5	56890

Showing all 5 rows.

```
string_int_data.withColumn("Integer_Or_Not",
col("Id").cast(IntegerType()).isNull()).show()
```

(3) Spark Jobs	
Id	Integer_Or_Not
123nm	false
12345	true
789pr	false
456jm	false
56890	true

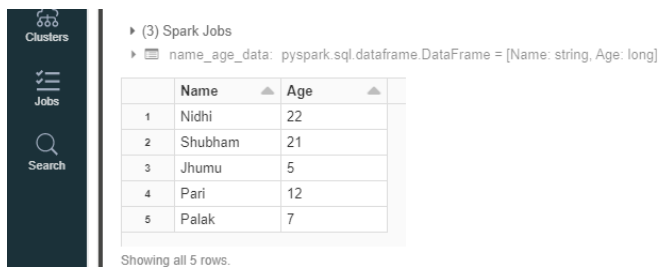
For Loop, If condition, Foreach command



Creating a dataframe

```
# Name_Age data
```

```
name_age_data = spark.createDataFrame(  
    [("Nidhi", 22), ("Shubham", 21),  
    ("Jhumu", 5), ("Pari", 12),  
    ("Palak", 7)],  
    ["Name", "Age"])  
  
display(name_age_data)
```



The screenshot shows the Databricks interface with a table containing 5 rows of data. The table has two columns: Name and Age. The rows are: 1 Nidhi 22, 2 Shubham 21, 3 Jhumu 5, 4 Pari 12, 5 Palak 7. The interface also shows a sidebar with 'Clusters', 'Jobs', and 'Search' options, and a status bar indicating 'Showing all 5 rows.'

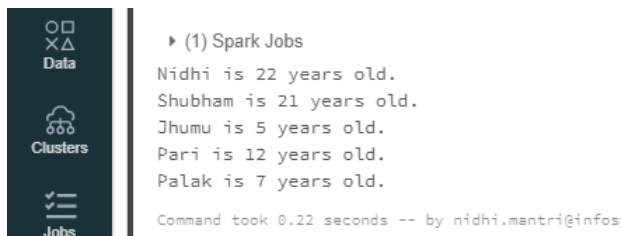
	Name	Age
1	Nidhi	22
2	Shubham	21
3	Jhumu	5
4	Pari	12
5	Palak	7

Printing data using for loop –



```
# for loop
```

```
for row in name_age_data.rdd.collect(): #name_age_data.collect() is also fine  
    print(row["Name"], "is", row["Age"], "years old.")
```



The screenshot shows the Databricks interface with a job output window. The output displays the following text: Nidhi is 22 years old., Shubham is 21 years old., Jhumu is 5 years old., Pari is 12 years old., Palak is 7 years old. The command took 0.22 seconds and was executed by nidhi.mantri@infos.

```
► (1) Spark Jobs  
Nidhi is 22 years old.  
Shubham is 21 years old.  
Jhumu is 5 years old.  
Pari is 12 years old.  
Palak is 7 years old.  
  
Command took 0.22 seconds -- by nidhi.mantri@infos
```

For loop and if condition –



```
# for loop and if condition
```

```
for row in name_age_data.rdd.collect():  
    if row["Age"] > 18:  
        print(row["Name"], "is a young adult.")  
    else:  
        print(row["Name"], "is a child.")
```

Clusters
Jobs
Search

▶ (1) Spark Jobs

Nidhi is a young adult.
Shubham is a young adult.
Jhumu is a child.
Pari is a child.
Palak is a child.

Command took 0.33 seconds -- by nidhi.mantri@infosys.com at 12/31/2020, 9:20:27 PM on Mantri

Foreach command –

```
name_age_data.foreach(lambda row : print(row))
```

Check its output in driver log on clusters page.

Clusters /
Mantri
Edit
Clone
Restart
Terminate
Delete

Configuration
Notebooks
Libraries
Event Log
Spark UI
Driver Logs
Metrics
Apps
Spark Cluster UI - Master

Standard error

```

You should import from traitlets.config instead.
"You should import from traitlets.config instead.", ShimWarning)
/databricks/python/lib/python3.7/site-packages/IPython/nbconvert.py:13: ShimWarning: The 'IPython.nbconvert' package has been deprecated since IPython
4.0. You should import from nbconvert instead.
"You should import from nbconvert instead.", ShimWarning)
Fri Jan 1 09:31:42 2021 py4j imported
Fri Jan 1 09:31:42 2021 Python shell started with PID 1155 and guid 4287c5b2da7f4610b304ab09672579e4
Fri Jan 1 09:31:42 2021 Initialized gateway on port 37701
Fri Jan 1 09:31:44 2021 Python shell executor start
Row(Name='Pari', Age=12)
Row(Name='Palak', Age=7)
Row(Name='Nidhi', Age=22)
Row(Name='Jhumu', Age=5)
Row(Name='Shubham', Age=21)

```

Log4j output

```

21/01/01 09:32:07 INFO BlockManagerInfo: Removed broadcast_2_piece0 on 10.172.252.58:45141 in memory (size: 4.3 KiB, free: 3.9 GiB)
21/01/01 09:32:07 INFO BlockManagerInfo: Removed broadcast_7_piece0 on 10.172.252.58:45141 in memory (size: 5.8 KiB, free: 3.9 GiB)
21/01/01 09:32:07 INFO BlockManagerInfo: Removed broadcast_18_piece0 on 10.172.252.58:45141 in memory (size: 6.0 KiB, free: 3.9 GiB)

```

Creating dataframe in for loop –

```

import pandas as pd
import numpy as np
# creating dataframe from pandas dataframe
test = sqlContext.createDataFrame(pd.DataFrame({'Set': np.arange(1,11)}))
for i in np.arange(2,6).tolist():
    test = test.withColumn('Set' + str(i), lit(i ** 2) + test.Set)
test.show()

```

```

+---+---+---+---+
| Set| Set2| Set3| Set4| Set5|
+---+---+---+---+
|  1|   5|  10|  17|  26|
|  2|   6|  11|  18|  27|
|  3|   7|  12|  19|  28|
|  4|   8|  13|  20|  29|
|  5|   9|  14|  21|  30|
|  6|  10|  15|  22|  31|
|  7|  11|  16|  23|  32|
|  8|  12|  17|  24|  33|
|  9|  13|  18|  25|  34|
| 10|  14|  19|  26|  35|
+---+---+---+---+

```

So, Yeahhh!!! With this we have successfully learnt about different built-in functions and loops.

Keep Learning!!!

- Nidhi Mantri

Specialist Programmer – Big Data