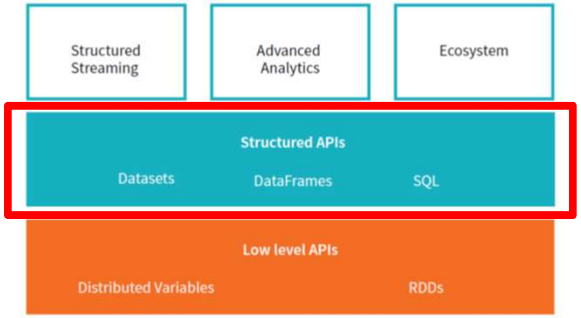
SQL

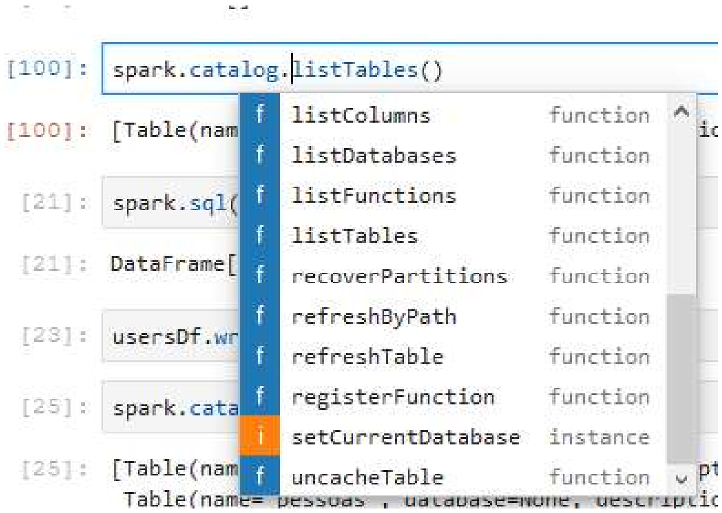

Structured Streaming
Advanced Analytics
Ecosystem
Structured APIs
Datasets
DataFrames
SQL
Low level APIs
Distributed Variables
RDDs

DATABASES, VIEWS, AND TABLES

DATABASES, VIEWS, AND TABLES

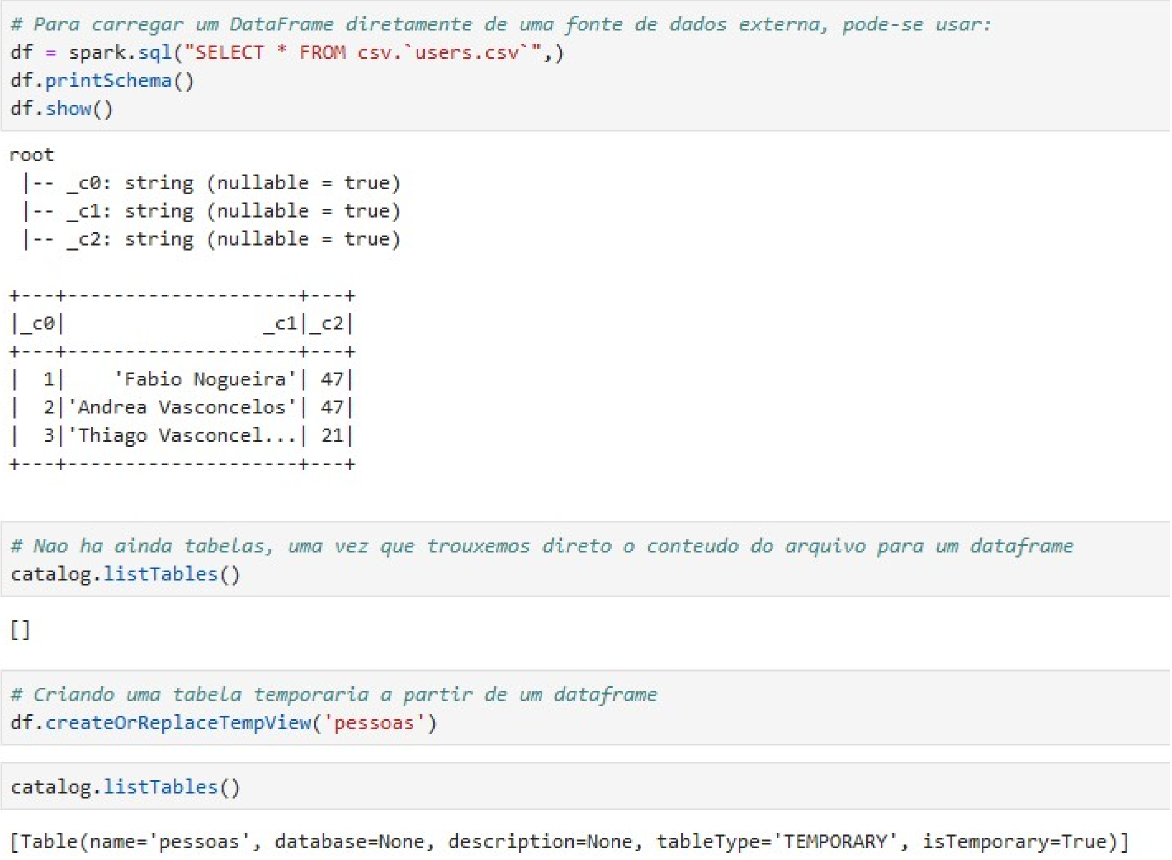
Besides the DataFrame API, spark implements the concepts of databases, functions, views and tables. The idea is to enable us to use standard SQL to process data

Spark stores metadata concerning those elements in an internal catalog, which is accessible from a SparkSession



```
[100]: spark.catalog.listTables()
[100]: [Table(nam
 [21]: spark.sql(
 [21]: DataFrame[
 [23]: usersDf.wr
 [25]: spark.cata
 [25]: [Table(nam
        Table(name='pessoas', database=None, descriptic
```

CREATING A VIEW



```
# Para carregar um DataFrame diretamente de uma fonte de dados externa, pode-se usar:
df = spark.sql("SELECT * FROM csv.`users.csv`",)
df.printSchema()
df.show()
```

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)

+---+--------------------+---+
|_c0|                 _c1|_c2|
+---+--------------------+---+
|  1|    'Fabio Nogueira'| 47|
|  2|'Andrea Vasconcelos'| 47|
|  3|'Thiago Vasconcel...| 21|
+---+--------------------+---+
```

```
# Nao ha ainda tabelas, uma vez que trouxemos direto o conteudo do arquivo para um dataframe
catalog.listTables()
```

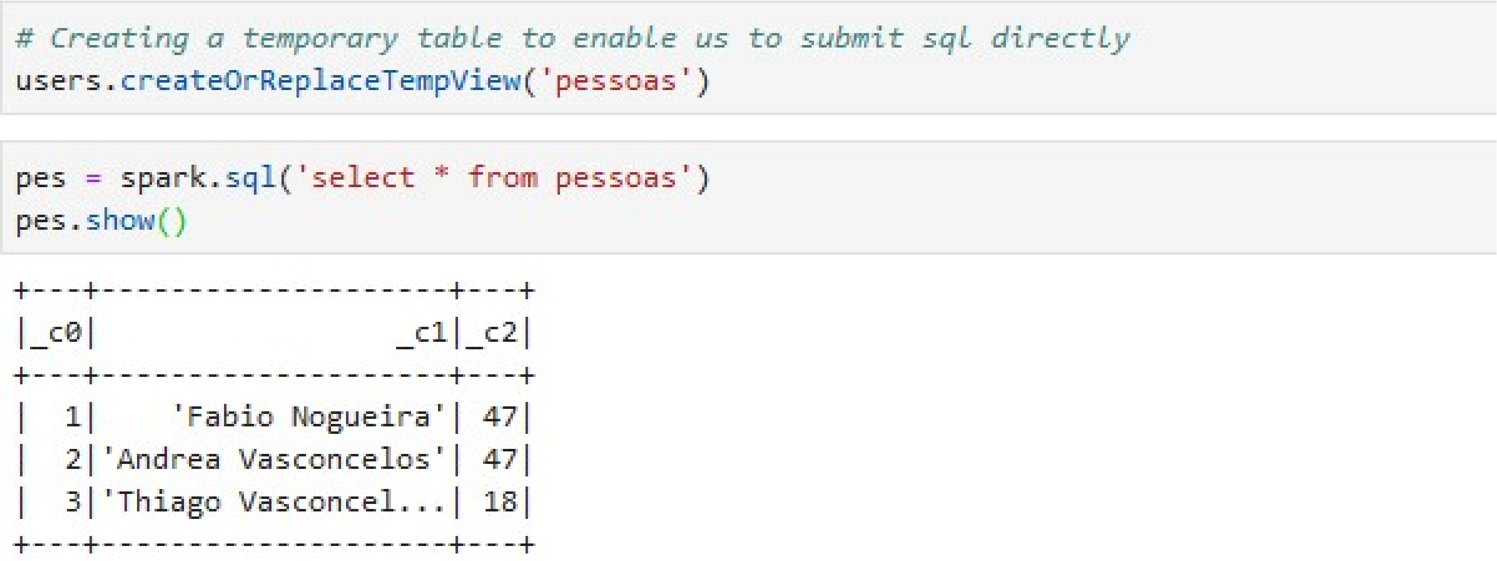
```
[]
```

```
# Criando uma tabela temporaria a partir de um dataframe
df.createOrReplaceTempView('pessoas')
```

```
catalog.listTables()
```

```
[Table(name='pessoas', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

QUERYING VIEWS USING SQL



```
# Creating a temporary table to enable us to submit sql directly
users.createOrReplaceTempView('pessoas')
```

```
pes = spark.sql('select * from pessoas')
pes.show()
```

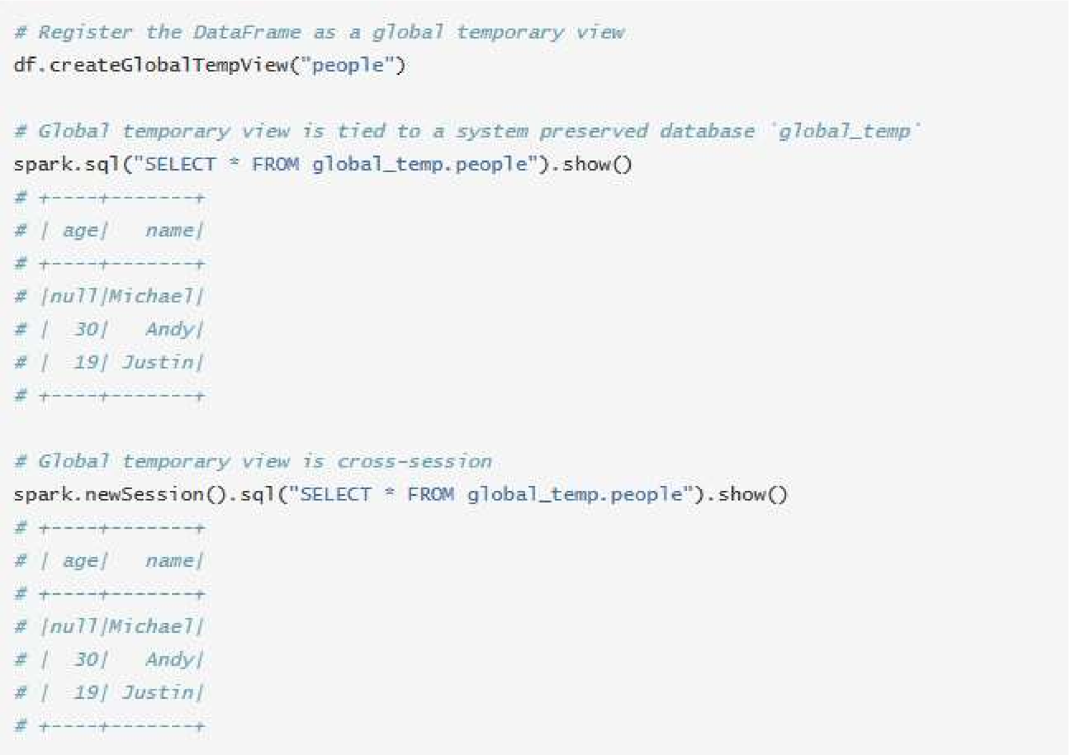
```
+---+--------------------+---+
|_c0|                 _c1|_c2|
+---+--------------------+---+
|  1|    'Fabio Nogueira'| 47|
|  2|'Andrea Vasconcelos'| 47|
|  3|'Thiago Vasconcel...| 18|
+---+--------------------+---+
```

TEMPORARY VIEWS

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates

If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view

TEMPORARY VIEWS



```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

PERSISTING DATAFRAMES

DataFrames can also be saved as persistent tables into Hive metastore using the saveAsTable command

These tables can be organized in Databases

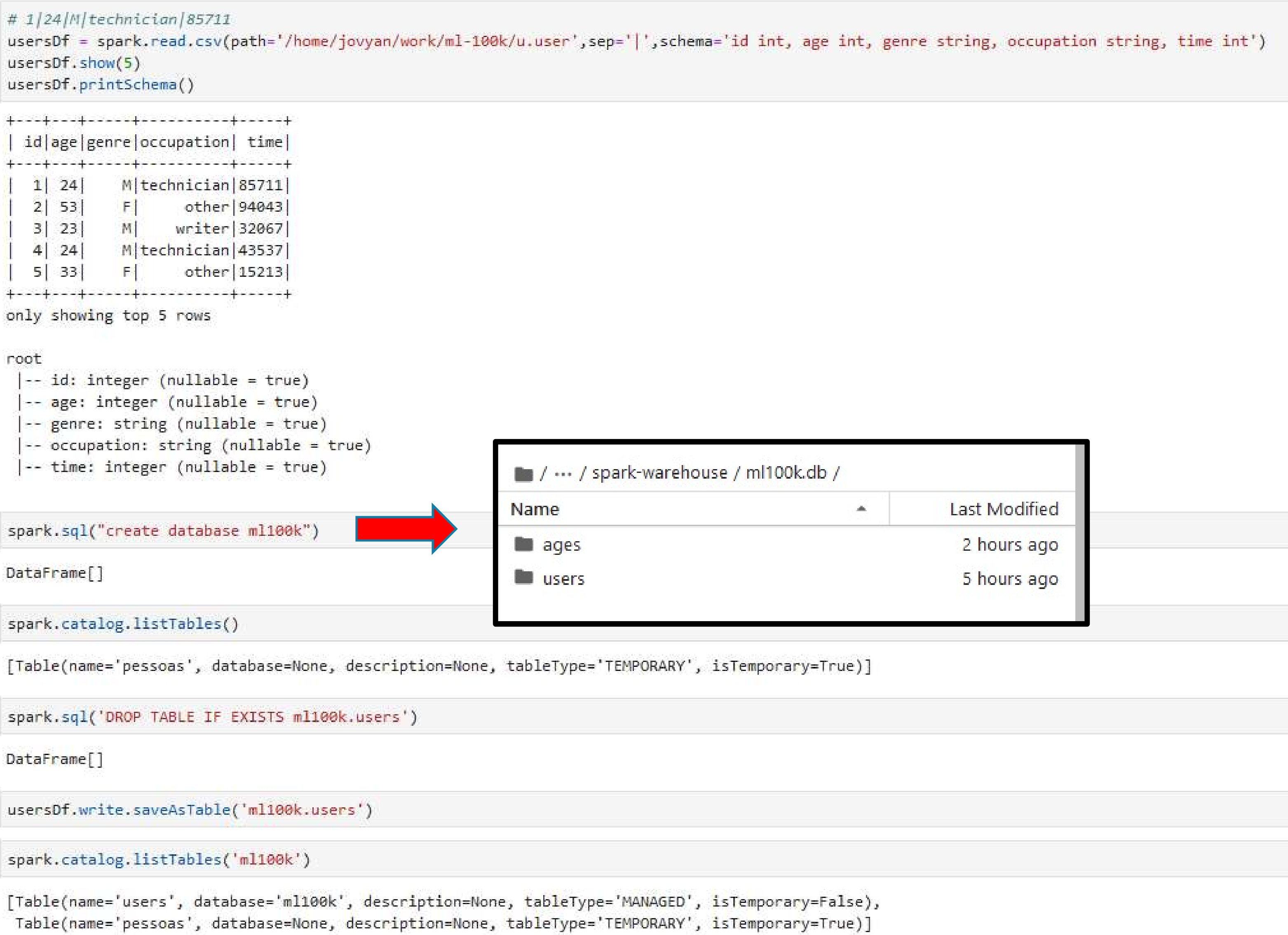
Unlike the createOrReplaceTempView command, saveAsTable will materialize the contents of the DataFrame and create a pointer to the data in the Hive metastore.

A DataFrame for a persistent table can be created by calling the table method on a SparkSession with the name of the table.

PERSISTING DATAFRAMES

For file-based data source, e.g. text, parquet, json, etc. you can specify a custom table path via the path option, e.g. df.write.option("path", "/some/path").saveAsTable("t"). Table persisted that way are referred to as “External”, this means that when the table is dropped, the custom table path will not be removed and the table data is still there. If no custom table path is specified, we say the table is “Managed” by Spark. So, it will write data to a default table path under the warehouse directory. When the table is dropped, the default table path will be removed too.

Persistent datasource tables have per-partition metadata stored in the Hive metastore.



```
# 1|24|M|technician|85711
usersDf = spark.read.csv(path='/home/jovyan/work/ml-100k/u.user',sep='|',schema='id int, age int, genre string, occupation string, time int')
usersDf.show(5)
usersDf.printSchema()
```

```
+---+---+-----+----------+-----+
| id|age|genre|occupation| time|
+---+---+-----+----------+-----+
|  1| 24|    M|technician|85711|
|  2| 53|    F|     other|94043|
|  3| 23|    M|    writer|32067|
|  4| 24|    M|technician|43537|
|  5| 33|    F|     other|15213|
+---+---+-----+----------+-----+
only showing top 5 rows

root
 |-- id: integer (nullable = true)
 |-- age: integer (nullable = true)
 |-- genre: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- time: integer (nullable = true)
```

```
spark.sql("create database ml100k")
```

```
DataFrame[]
```

/ ··· / spark-warehouse / ml100k.db /

Name	Last Modified
ages	2 hours ago
users	5 hours ago

```
spark.catalog.listTables()
```

```
[Table(name='pessoas', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

```
spark.sql('DROP TABLE IF EXISTS ml100k.users')
```

```
DataFrame[]
```

```
usersDf.write.saveAsTable('ml100k.users')
```

```
spark.catalog.listTables('ml100k')
```

```
[Table(name='users', database='ml100k', description=None, tableType='MANAGED', isTemporary=False),
 Table(name='pessoas', database=None, description=None, tableType='TEMPORARY', isTemporary=True)]
```

FUNCTIONS

FUNCTIONS

Structured APIs includes lots of built-in functions that can be applied to columns according to their types

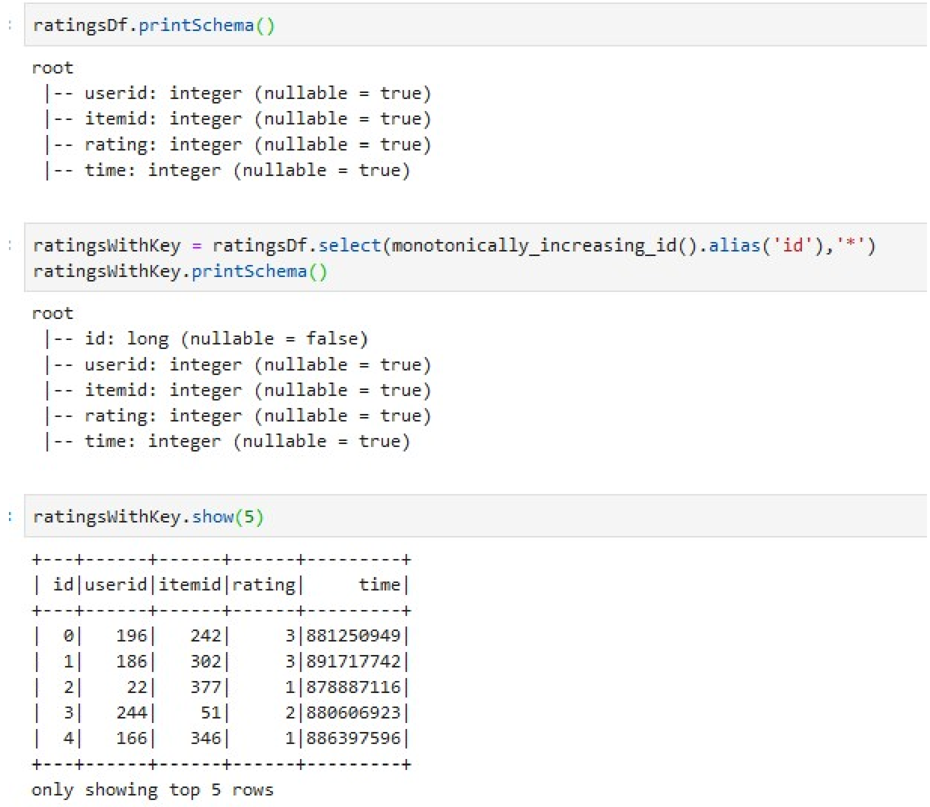
We will show some of them, others can be found here:

- https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions

Moreover, Spark allows users to define their own functions, which can be registered and used in their queries

NUMERICAL

GENERATING IDS



```
ratingsDf.printSchema()
```

```
root
 |-- userid: integer (nullable = true)
 |-- itemid: integer (nullable = true)
 |-- rating: integer (nullable = true)
 |-- time: integer (nullable = true)
```

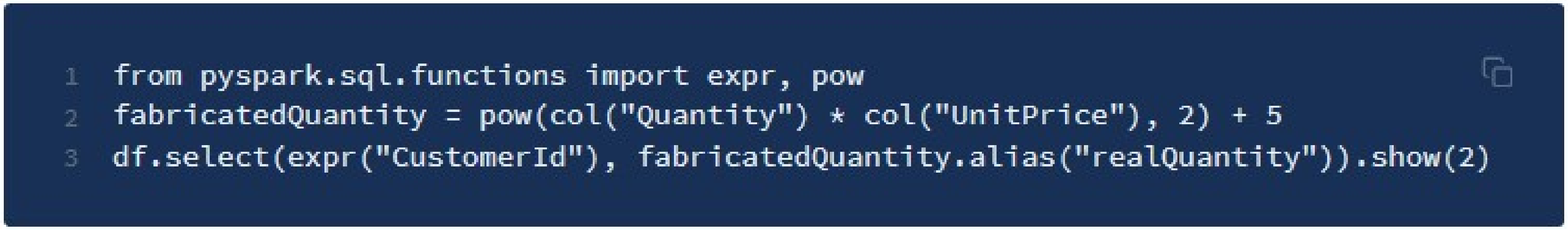
```
ratingsWithKey = ratingsDf.select(monotonically_increasing_id().alias('id'),'*')
ratingsWithKey.printSchema()
```

```
root
 |-- id: long (nullable = false)
 |-- userid: integer (nullable = true)
 |-- itemid: integer (nullable = true)
 |-- rating: integer (nullable = true)
 |-- time: integer (nullable = true)
```

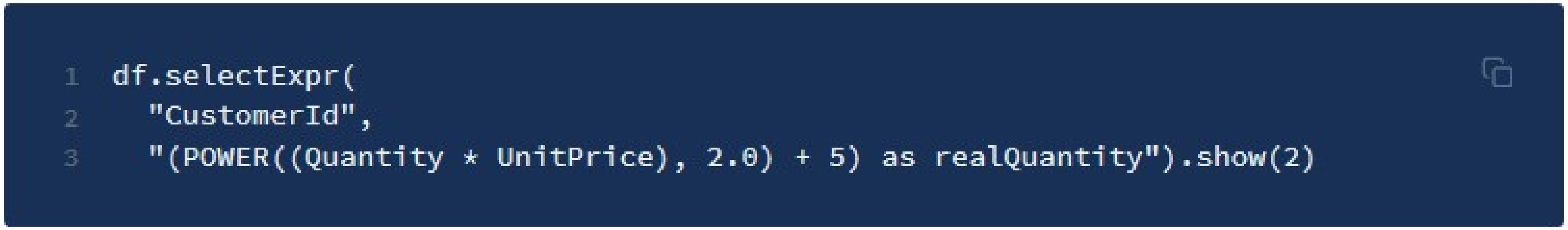
```
ratingsWithKey.show(5)
```

```
+---+------+------+------+---------+
| id|userid|itemid|rating|     time|
+---+------+------+------+---------+
|  0|   196|   242|     3|881250949|
|  1|   186|   302|     3|891717742|
|  2|    22|   377|     1|878887116|
|  3|   244|    51|     2|880606923|
|  4|   166|   346|     1|886397596|
+---+------+------+------+---------+
only showing top 5 rows
```

POW

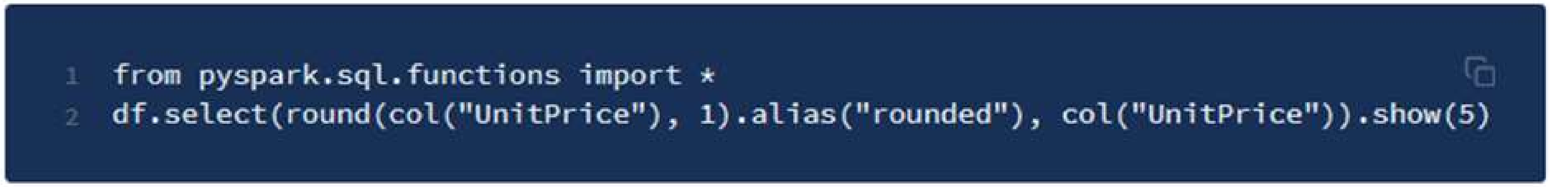


```
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)
```



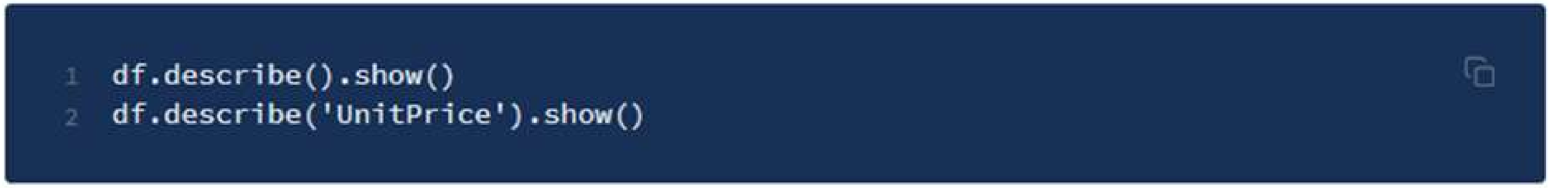
```
df.selectExpr(
  "CustomerId",
  "(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)
```

ROUND AND BROUND

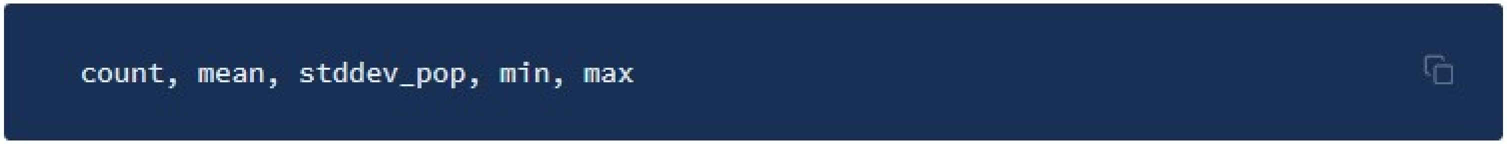


```
from pyspark.sql.functions import *
df.select(round(col("UnitPrice"), 1).alias("rounded"), col("UnitPrice")).show(5)
```

STATISTICS



```
df.describe().show()
df.describe('UnitPrice').show()
```

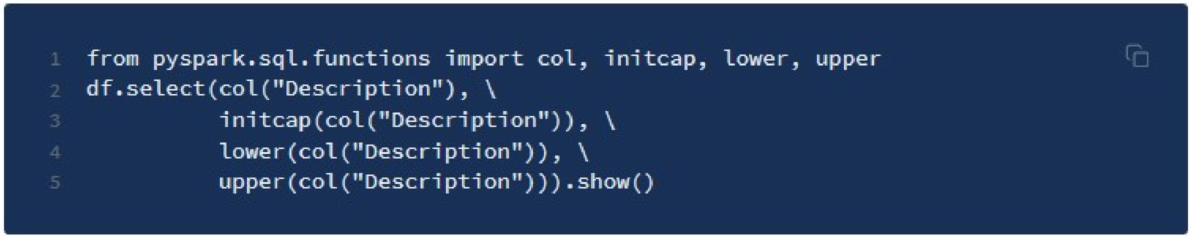


```
count, mean, stddev_pop, min, max
```

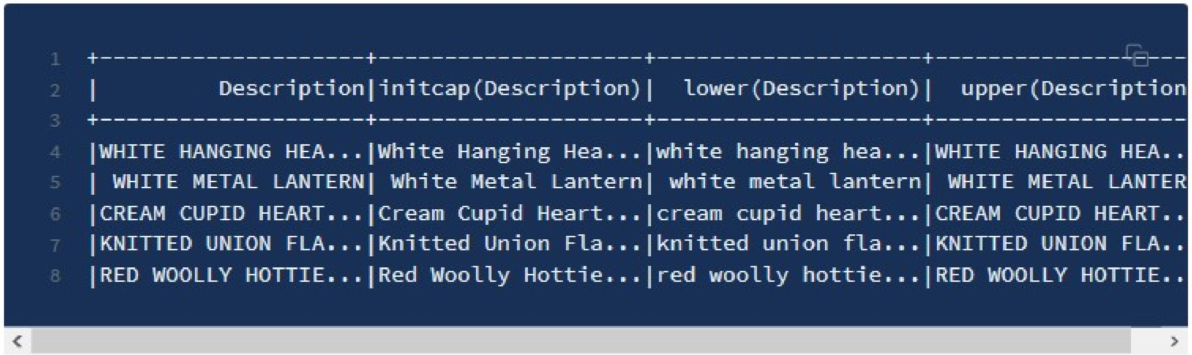
STRINGS

CASE

To modify a string column in order to change the case of its contents, we can use *initcap, lower,* and *upper* functions

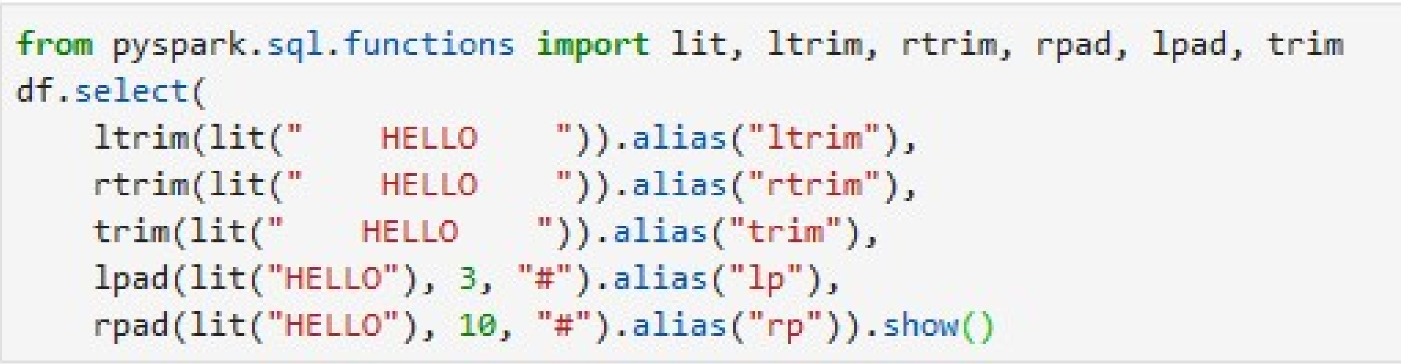


```
from pyspark.sql.functions import col, initcap, lower, upper
df.select(col("Description"), \
          initcap(col("Description")), \
          lower(col("Description")), \
          upper(col("Description"))).show()
```

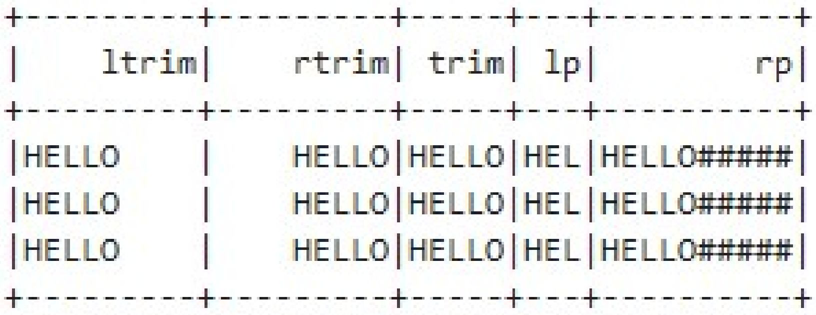


```
+--------------------+--------------------+--------------------+------------------
|         Description|initcap(Description)|  lower(Description)|  upper(Description
+--------------------+--------------------+--------------------+------------------
|WHITE HANGING HEA...|White Hanging Hea...|white hanging hea...|WHITE HANGING HEA..
| WHITE METAL LANTERN| White Metal Lantern| white metal lantern| WHITE METAL LANTER
|CREAM CUPID HEART...|Cream Cupid Heart...|cream cupid heart...|CREAM CUPID HEART..
|KNITTED UNION FLA...|Knitted Union Fla...|knitted union fla...|KNITTED UNION FLA..
|RED WOOLLY HOTTIE...|Red Woolly Hottie...|red woolly hottie...|RED WOOLLY HOTTIE..
```

DEALING WITH SPACES

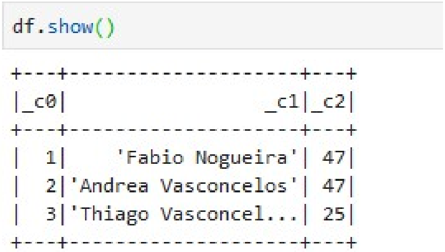


```
from pyspark.sql.functions import lit, ltrim, rtrim, rpad, lpad, trim
df.select(
    ltrim(lit("    HELLO    ")).alias("ltrim"),
    rtrim(lit("    HELLO    ")).alias("rtrim"),
    trim(lit("    HELLO    ")).alias("trim"),
    lpad(lit("HELLO"), 3, "#").alias("lp"),
    rpad(lit("HELLO"), 10, "#").alias("rp")).show()
```



```
+---------+---------+-----+---+----------+
|    ltrim|    rtrim| trim| lp|        rp|
+---------+---------+-----+---+----------+
|HELLO    |    HELLO|HELLO|HEL|HELLO#####|
|HELLO    |    HELLO|HELLO|HEL|HELLO#####|
|HELLO    |    HELLO|HELLO|HEL|HELLO#####|
+---------+---------+-----+---+----------+
```

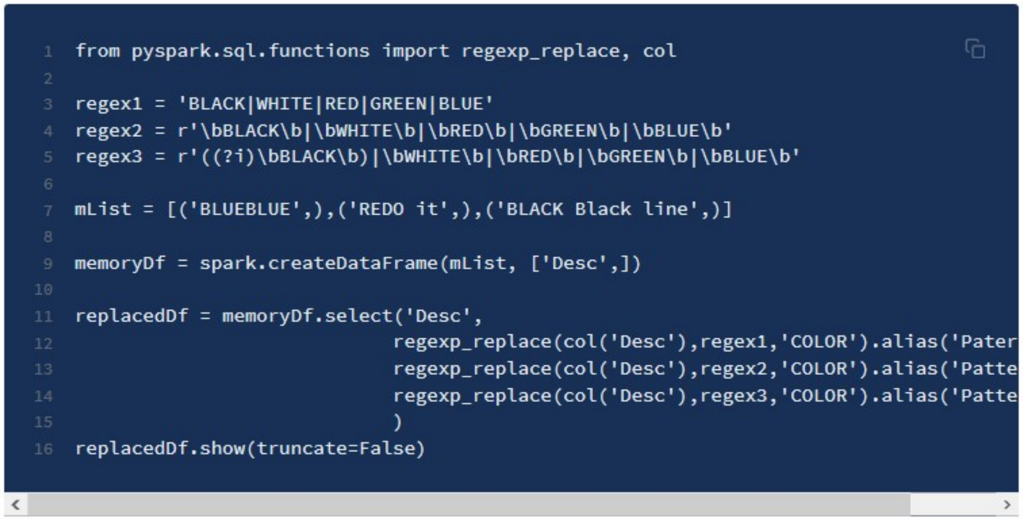
But, wait!!! Why do I see 3 lines here???



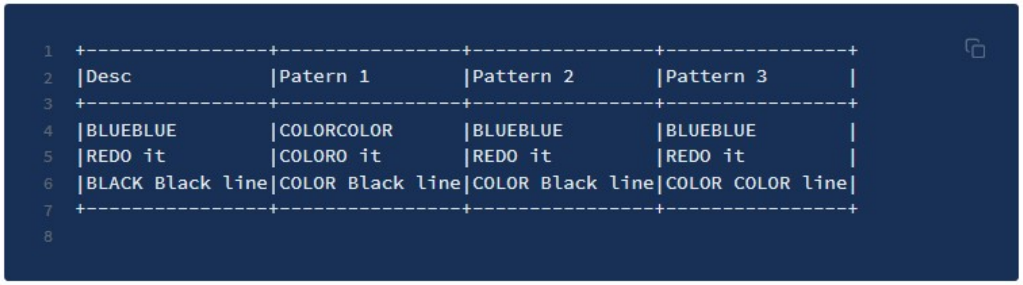
```
df.show()
```

```
+---+--------------------+---+
|_c0|                 _c1|_c2|
+---+--------------------+---+
|  1|    'Fabio Nogueira'| 47|
|  2|'Andrea Vasconcelos'| 47|
|  3|'Thiago Vasconcel...| 25|
+---+--------------------+---+
```

REPLACING STRINGS

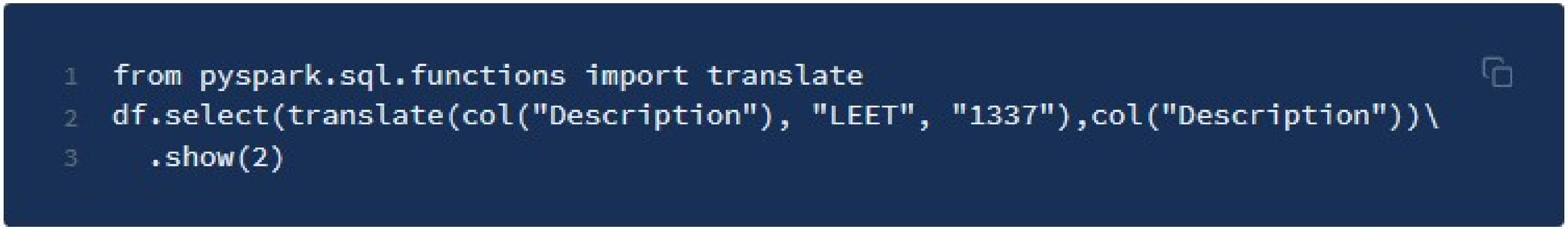


```python
from pyspark.sql.functions import regexp_replace, col

regex1 = 'BLACK|WHITE|RED|GREEN|BLUE'
regex2 = r'\bBLACK\b|\bWHITE\b|\bRED\b|\bGREEN\b|\bBLUE\b'
regex3 = r'((?i)\bBLACK\b)|\bWHITE\b|\bRED\b|\bGREEN\b|\bBLUE\b'

mList = [('BLUEBLUE',),('REDO it',),('BLACK Black line',)]

memoryDf = spark.createDataFrame(mList, ['Desc',])

replacedDf = memoryDf.select('Desc',
                            regexp_replace(col('Desc'),regex1,'COLOR').alias('Pater
                            regexp_replace(col('Desc'),regex2,'COLOR').alias('Patte
                            regexp_replace(col('Desc'),regex3,'COLOR').alias('Patte
                            )
replacedDf.show(truncate=False)
```

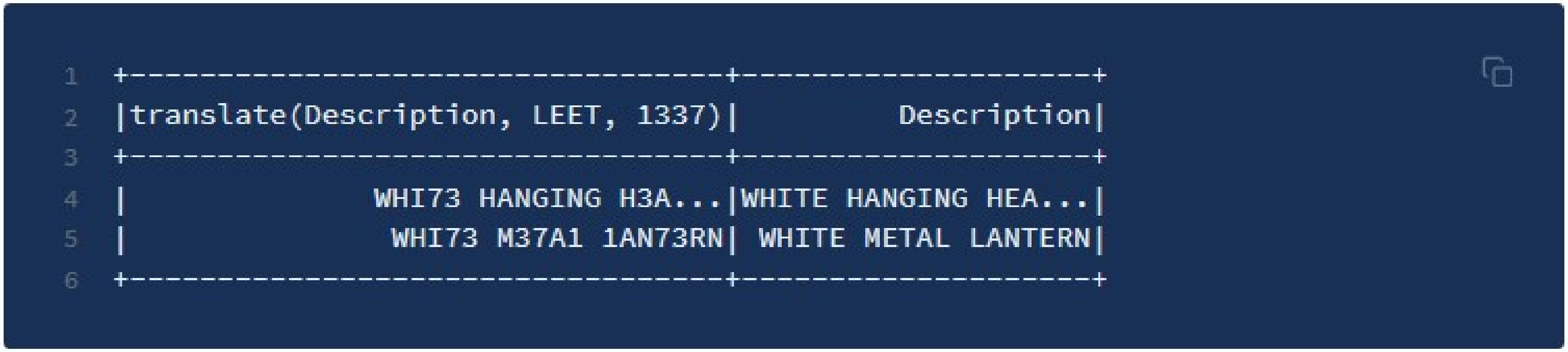


```
+----------------+----------------+----------------+----------------+
|Desc            |Patern 1        |Pattern 2       |Pattern 3       |
+----------------+----------------+----------------+----------------+
|BLUEBLUE        |COLORCOLOR      |BLUEBLUE        |BLUEBLUE        |
|REDO it         |COLORO it       |REDO it         |REDO it         |
|BLACK Black line|COLOR Black line|COLOR Black line|COLOR COLOR line|
+----------------+----------------+----------------+----------------+

```

MODIFYING INDIVIDUAL CHARACTERS

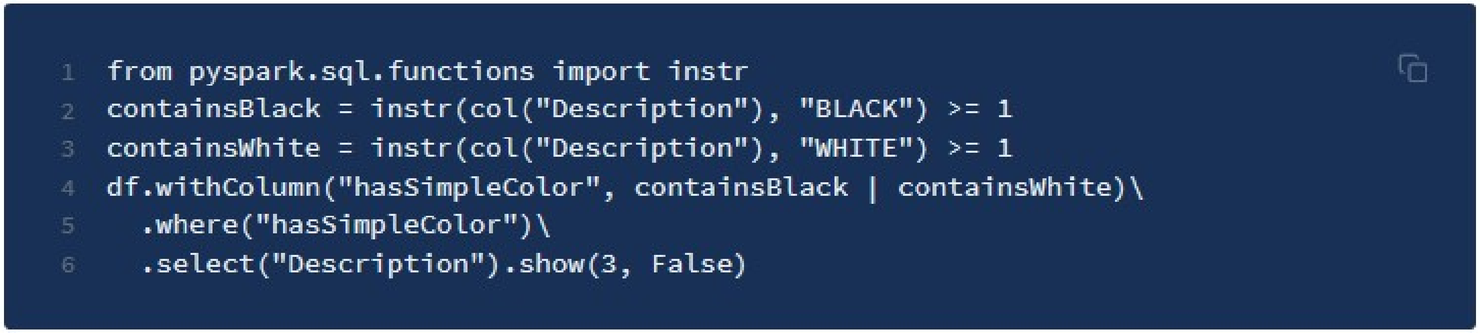


```
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\
  .show(2)
```

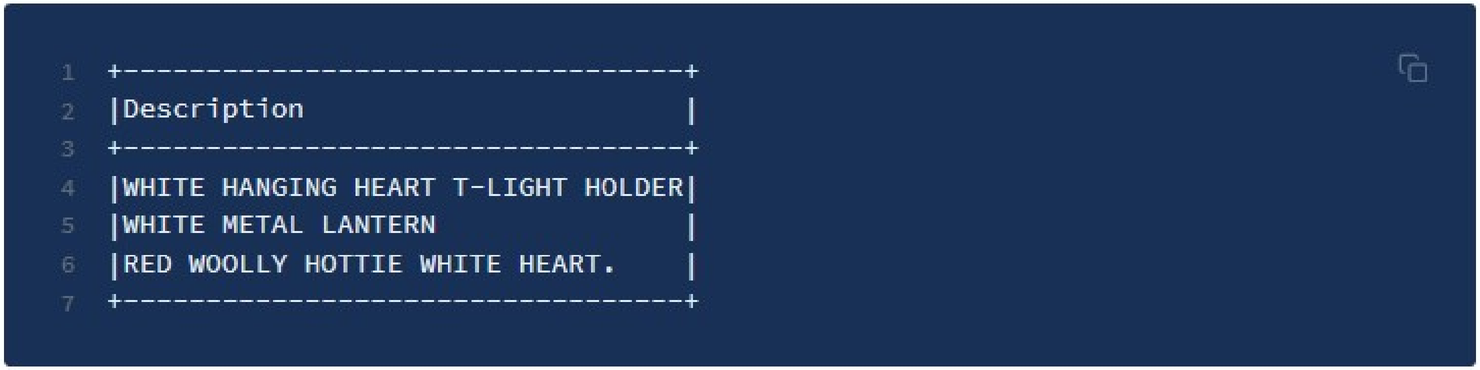


```
+----------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
+----------------------------------+--------------------+
|              WHI73 HANGING H3A...|WHITE HANGING HEA...|
|               WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
+----------------------------------+--------------------+
```

VERIFYING IF A STRING IS IN A COLUMN



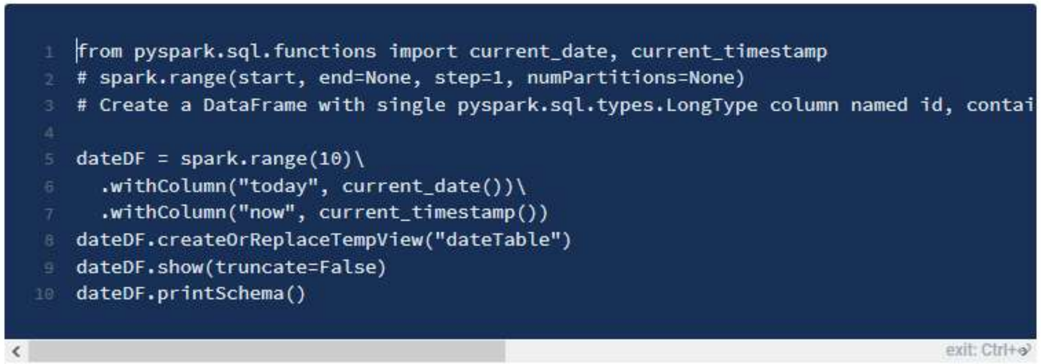
```
from pyspark.sql.functions import instr
containsBlack = instr(col("Description"), "BLACK") >= 1
containsWhite = instr(col("Description"), "WHITE") >= 1
df.withColumn("hasSimpleColor", containsBlack | containsWhite)\
  .where("hasSimpleColor")\
  .select("Description").show(3, False)
```



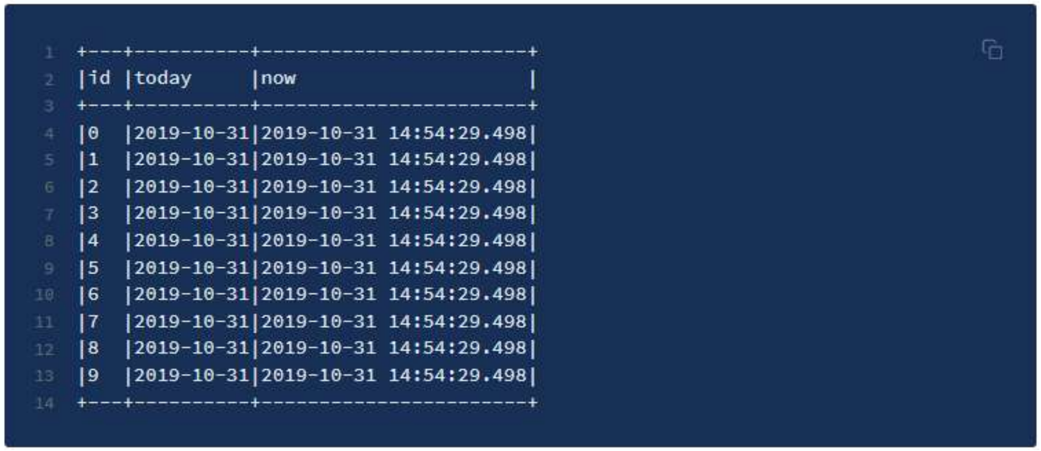
```
+----------------------------------+
|Description                       |
+----------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN               |
|RED WOOLLY HOTTIE WHITE HEART.    |
+----------------------------------+
```

DATES AND TIME

CURRENT DATE AND TIMESTAMP

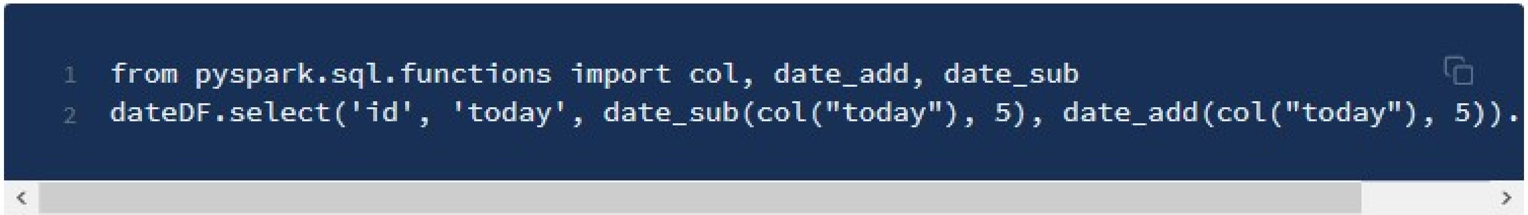


```
from pyspark.sql.functions import current_date, current_timestamp
# spark.range(start, end=None, step=1, numPartitions=None)
# Create a DataFrame with single pyspark.sql.types.LongType column named id, contai

dateDF = spark.range(10)\
  .withColumn("today", current_date())\
  .withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")
dateDF.show(truncate=False)
dateDF.printSchema()
```

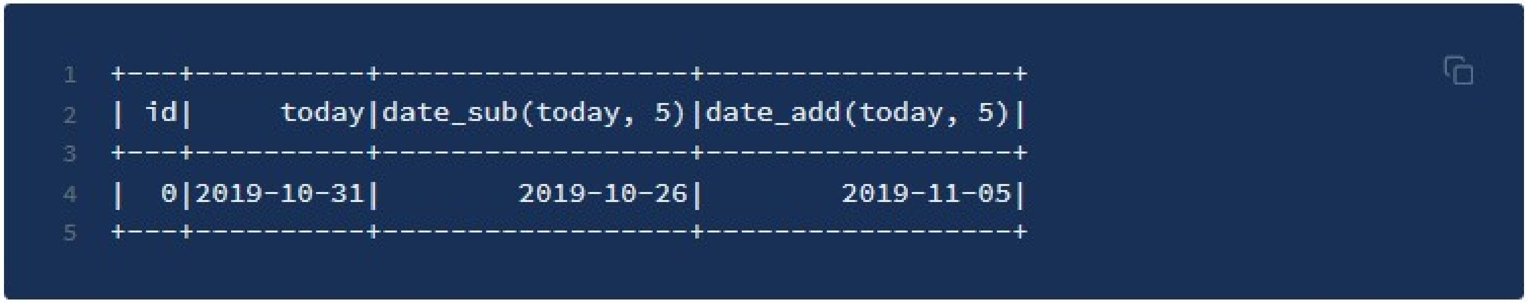


```
+---+----------+-----------------------+
|id |today     |now                    |
+---+----------+-----------------------+
|0  |2019-10-31|2019-10-31 14:54:29.498|
|1  |2019-10-31|2019-10-31 14:54:29.498|
|2  |2019-10-31|2019-10-31 14:54:29.498|
|3  |2019-10-31|2019-10-31 14:54:29.498|
|4  |2019-10-31|2019-10-31 14:54:29.498|
|5  |2019-10-31|2019-10-31 14:54:29.498|
|6  |2019-10-31|2019-10-31 14:54:29.498|
|7  |2019-10-31|2019-10-31 14:54:29.498|
|8  |2019-10-31|2019-10-31 14:54:29.498|
|9  |2019-10-31|2019-10-31 14:54:29.498|
+---+----------+-----------------------+
```

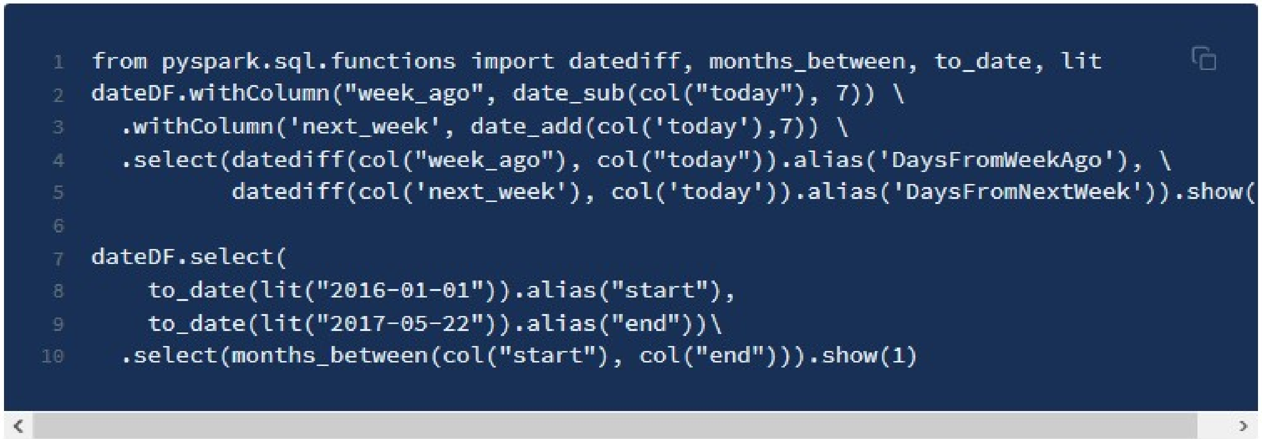
ADD AND SUB DATES



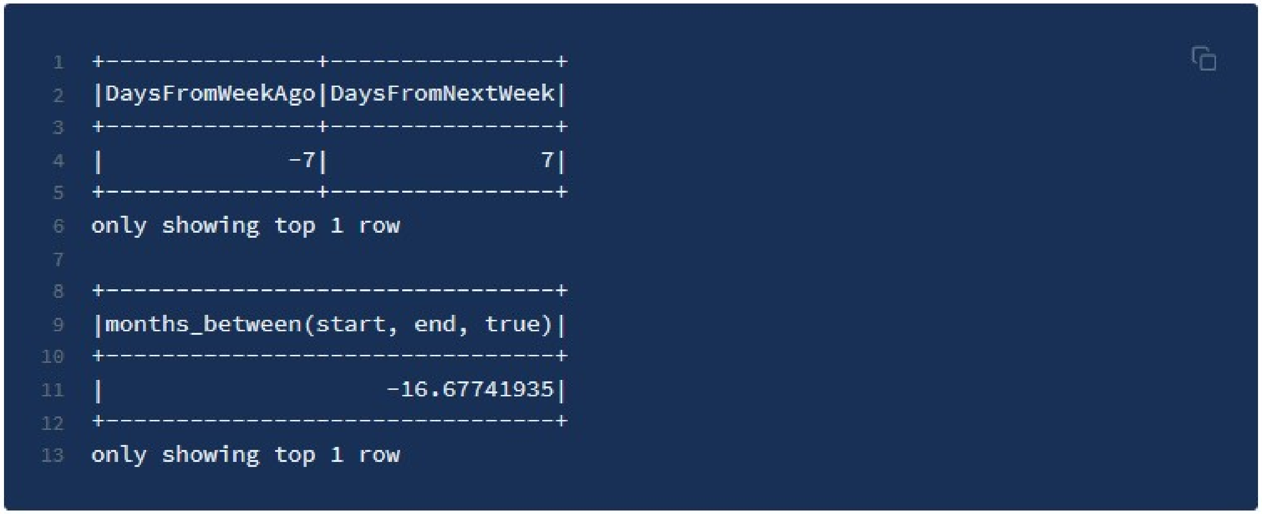
```
from pyspark.sql.functions import col, date_add, date_sub
dateDF.select('id', 'today', date_sub(col("today"), 5), date_add(col("today"), 5)).
```



```
+---+----------+------------------+------------------+
| id|     today|date_sub(today, 5)|date_add(today, 5)|
+---+----------+------------------+------------------+
|  0|2019-10-31|        2019-10-26|        2019-11-05|
+---+----------+------------------+------------------+
```



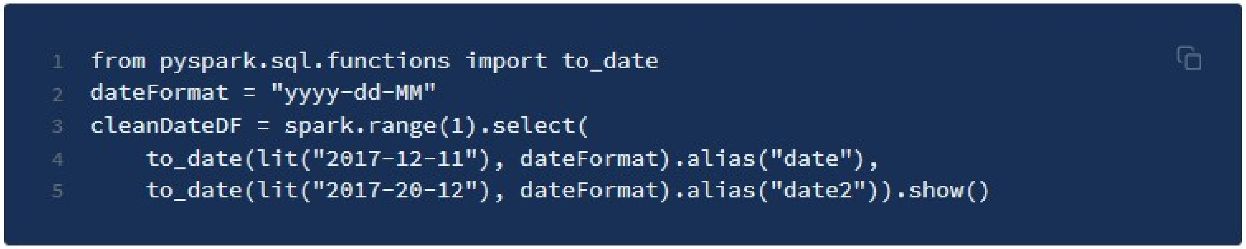
```
from pyspark.sql.functions import datediff, months_between, to_date, lit
dateDF.withColumn("week_ago", date_sub(col("today"), 7)) \
  .withColumn('next_week', date_add(col('today'),7)) \
  .select(datediff(col("week_ago"), col("today")).alias('DaysFromWeekAgo'), \
          datediff(col('next_week'), col('today')).alias('DaysFromNextWeek')).show(

dateDF.select(
    to_date(lit("2016-01-01")).alias("start"),
    to_date(lit("2017-05-22")).alias("end"))\
  .select(months_between(col("start"), col("end"))).show(1)
```



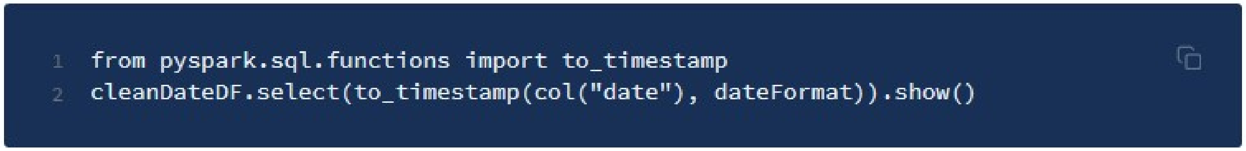
```
+---------------+----------------+
|DaysFromWeekAgo|DaysFromNextWeek|
+---------------+----------------+
|             -7|               7|
+---------------+----------------+
only showing top 1 row

+--------------------------------+
|months_between(start, end, true)|
+--------------------------------+
|                    -16.67741935|
+--------------------------------+
only showing top 1 row
```

CREATING DATES

We can convert a string or timestamp column to a *date* using *to_date* function. Besides the column, it can receive a format, specified according to *SimpleDateFormats* (https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html). If a value in the string column can not be recognized as a *date*, Spark will just return *null* as the corresponding value.



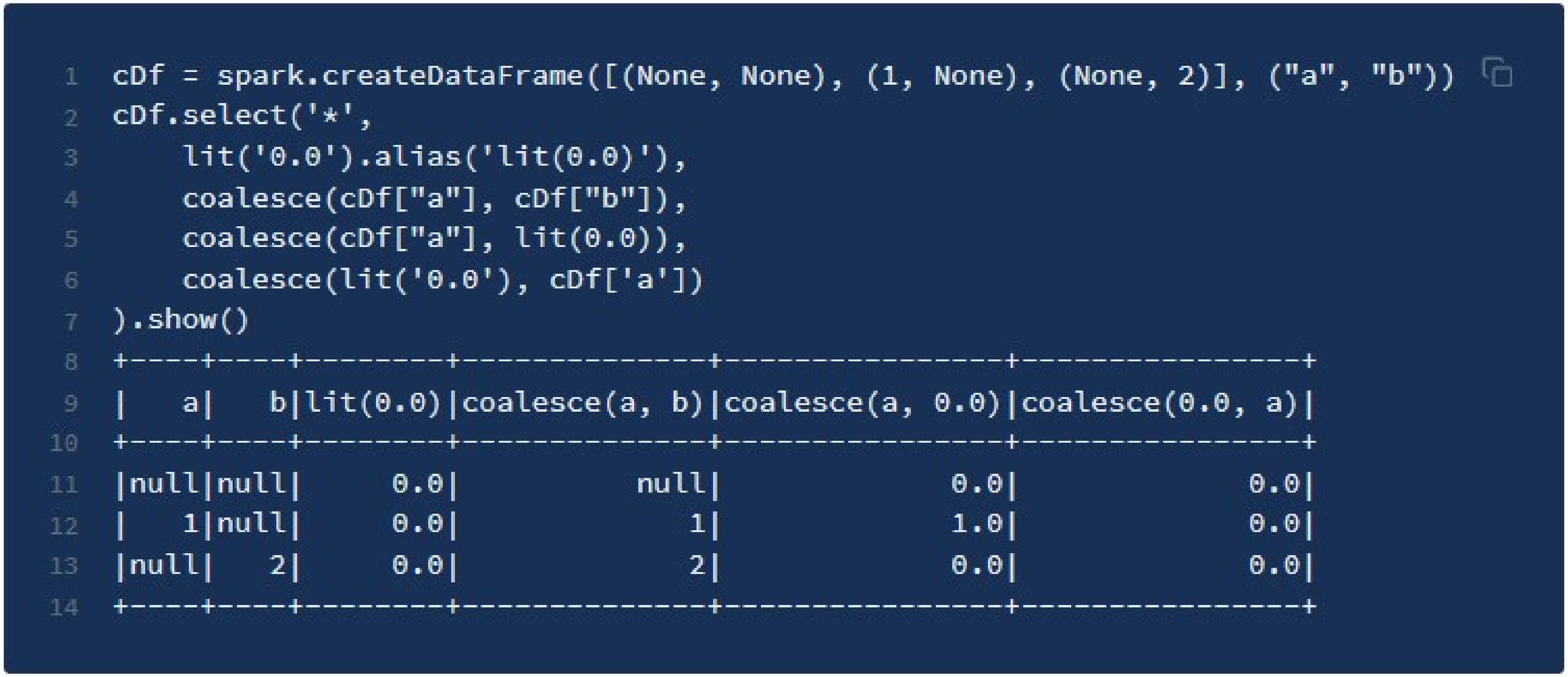
```
from pyspark.sql.functions import to_date
dateFormat = "yyyy-dd-MM"
cleanDateDF = spark.range(1).select(
    to_date(lit("2017-12-11"), dateFormat).alias("date"),
    to_date(lit("2017-20-12"), dateFormat).alias("date2")).show()
```



```
from pyspark.sql.functions import to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()
```

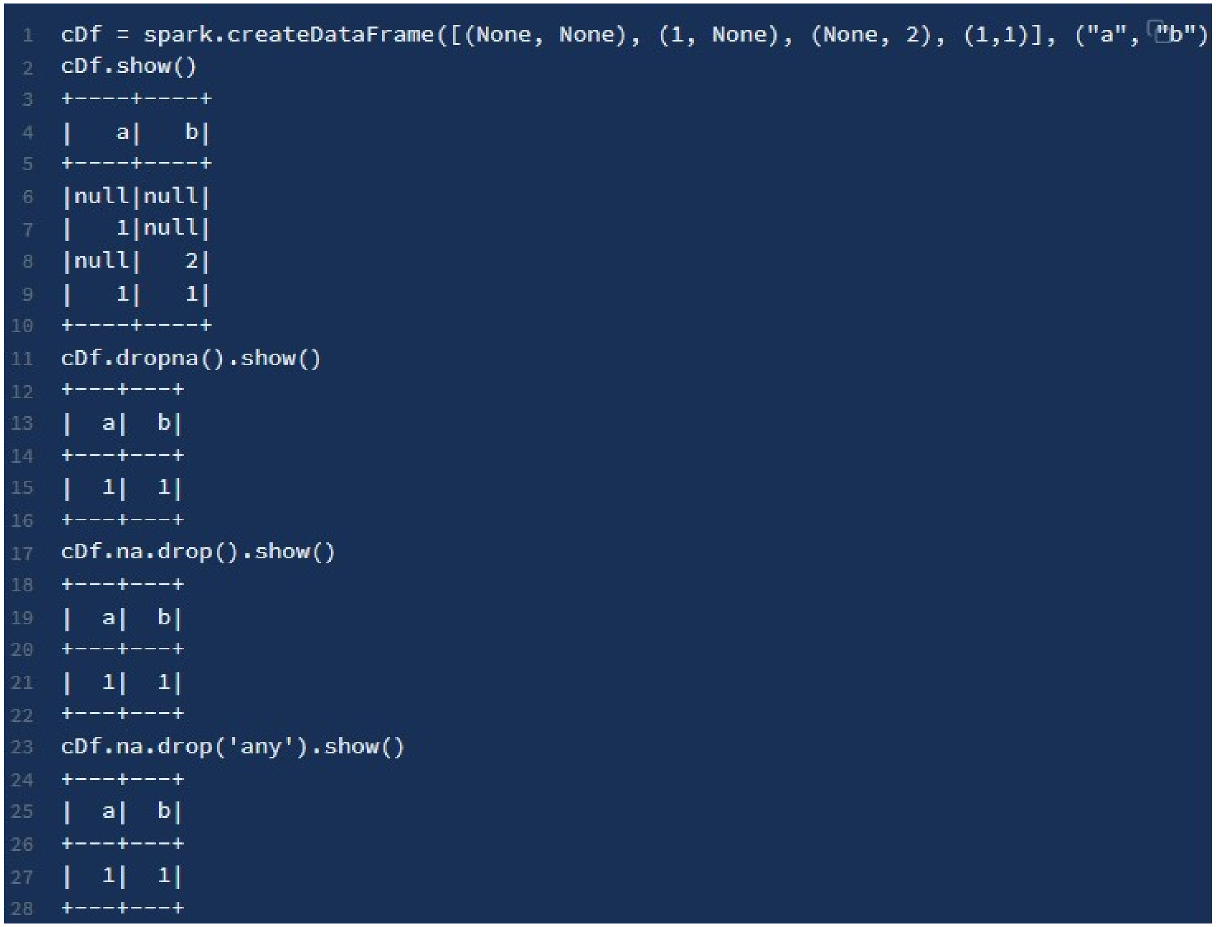
WORKING WITH NULL

COALESCE



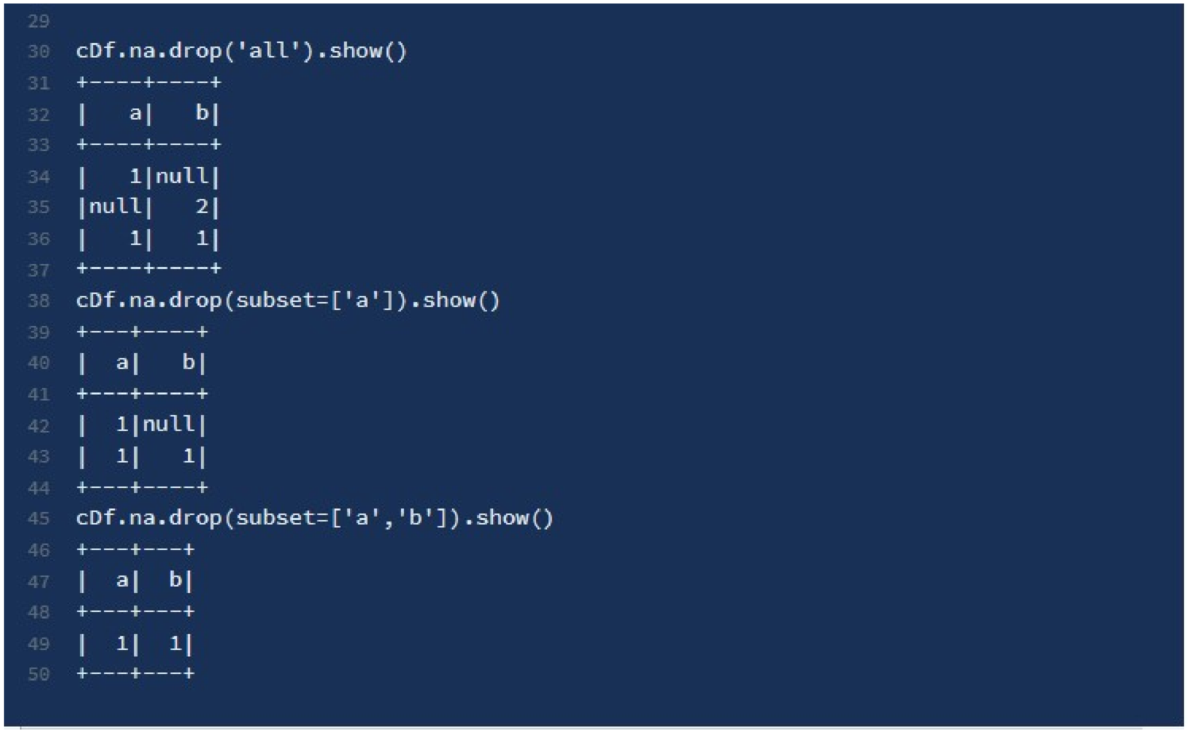
```
cDf = spark.createDataFrame([(None, None), (1, None), (None, 2)], ("a", "b"))
cDf.select('*',
    lit('0.0').alias('lit(0.0)'),
    coalesce(cDf["a"], cDf["b"]),
    coalesce(cDf["a"], lit(0.0)),
    coalesce(lit('0.0'), cDf['a'])
).show()
+----+----+--------+--------------+----------------+----------------+
|   a|   b|lit(0.0)|coalesce(a, b)|coalesce(a, 0.0)|coalesce(0.0, a)|
+----+----+--------+--------------+----------------+----------------+
|null|null|     0.0|          null|             0.0|             0.0|
|   1|null|     0.0|             1|             1.0|             0.0|
|null|   2|     0.0|             2|             0.0|             0.0|
+----+----+--------+--------------+----------------+----------------+
```

DROP



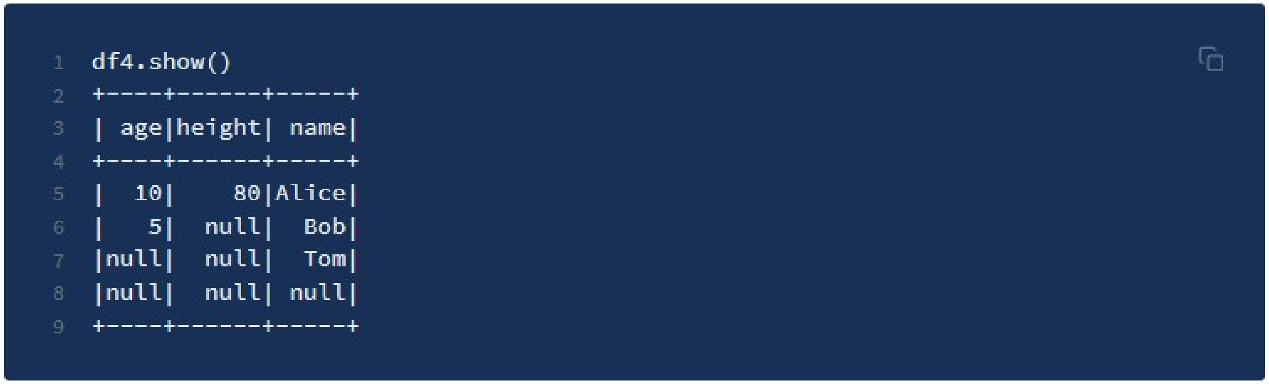
```
cDf = spark.createDataFrame([(None, None), (1, None), (None, 2), (1,1)], ("a", "b")
cDf.show()
+----+----+
|   a|   b|
+----+----+
|null|null|
|   1|null|
|null|   2|
|   1|   1|
+----+----+
cDf.dropna().show()
+---+---+
|  a|  b|
+---+---+
|  1|  1|
+---+---+
cDf.na.drop().show()
+---+---+
|  a|  b|
+---+---+
|  1|  1|
+---+---+
cDf.na.drop('any').show()
+---+---+
|  a|  b|
+---+---+
|  1|  1|
+---+---+
```

DROP (CONT)

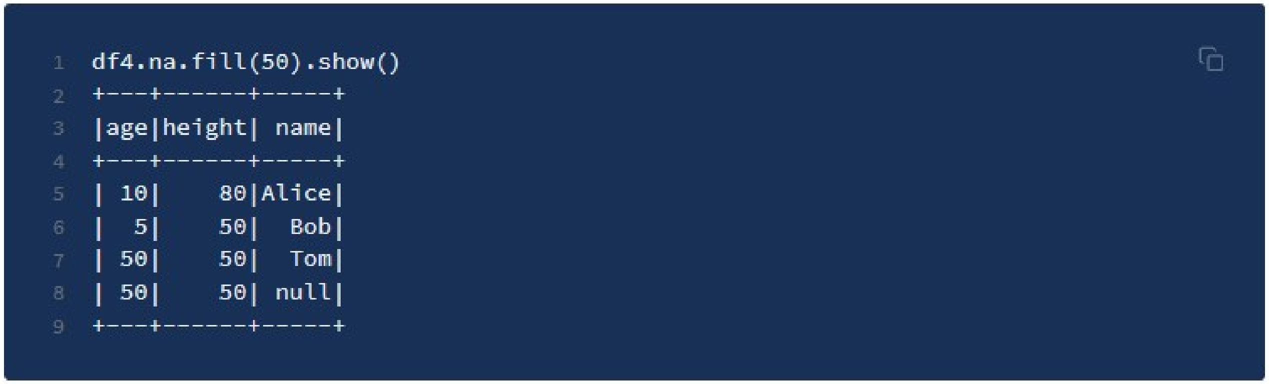


```
cDf.na.drop('all').show()
+----+----+
|   a|   b|
+----+----+
|   1|null|
|null|   2|
|   1|   1|
+----+----+
cDf.na.drop(subset=['a']).show()
+---+----+
|  a|   b|
+---+----+
|  1|null|
|  1|   1|
+---+----+
cDf.na.drop(subset=['a','b']).show()
+---+---+
|  a|  b|
+---+---+
|  1|  1|
+---+---+
```

FILL

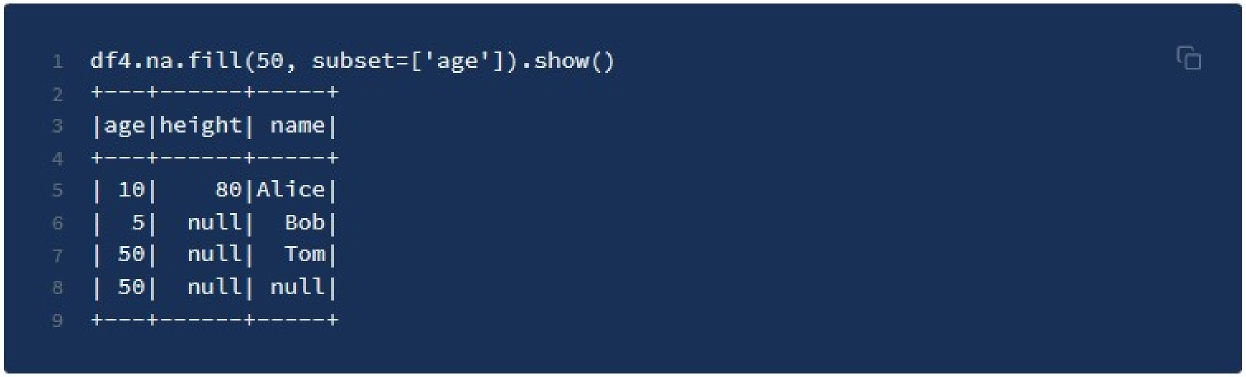


```
df4.show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  10|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

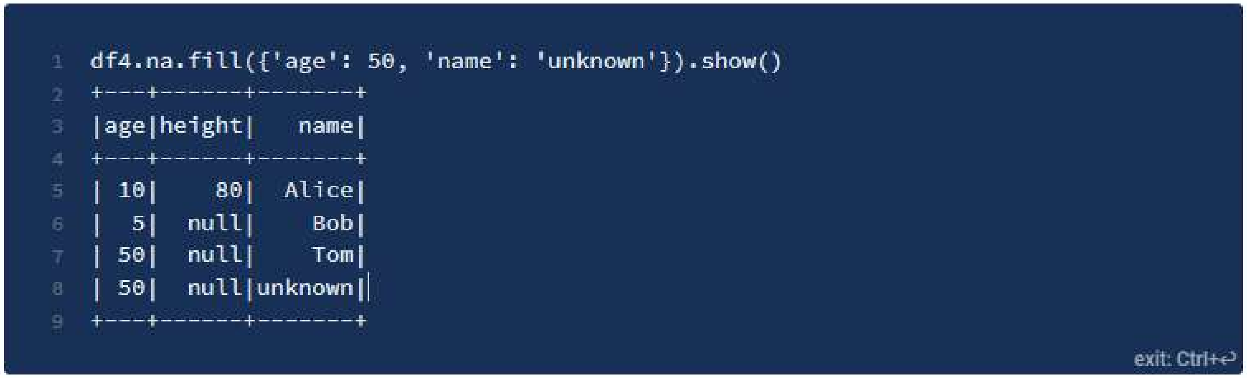


```
df4.na.fill(50).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
|  5|    50|  Bob|
| 50|    50|  Tom|
| 50|    50| null|
+---+------+-----+
```

FILL (CONT)

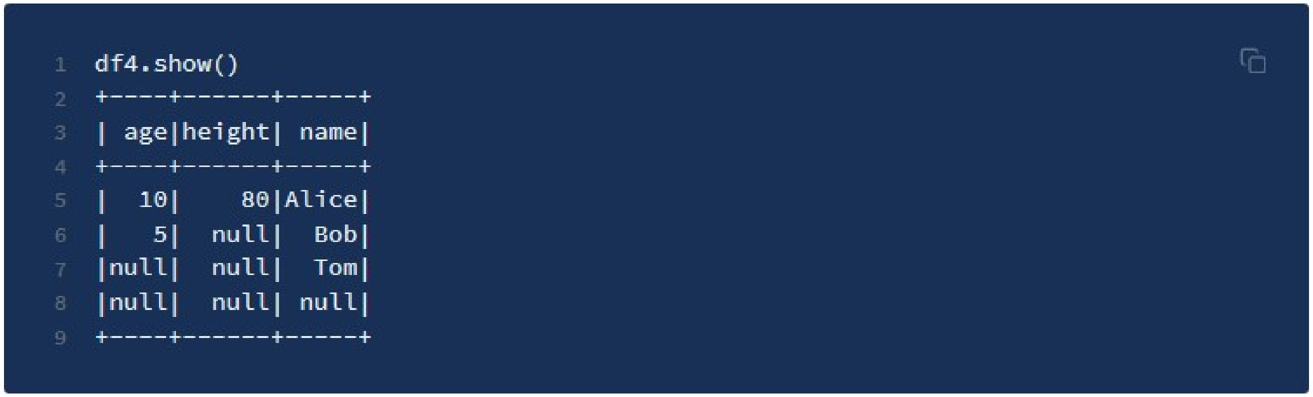


```
df4.na.fill(50, subset=['age']).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
|  5|  null|  Bob|
| 50|  null|  Tom|
| 50|  null| null|
+---+------+-----+
```

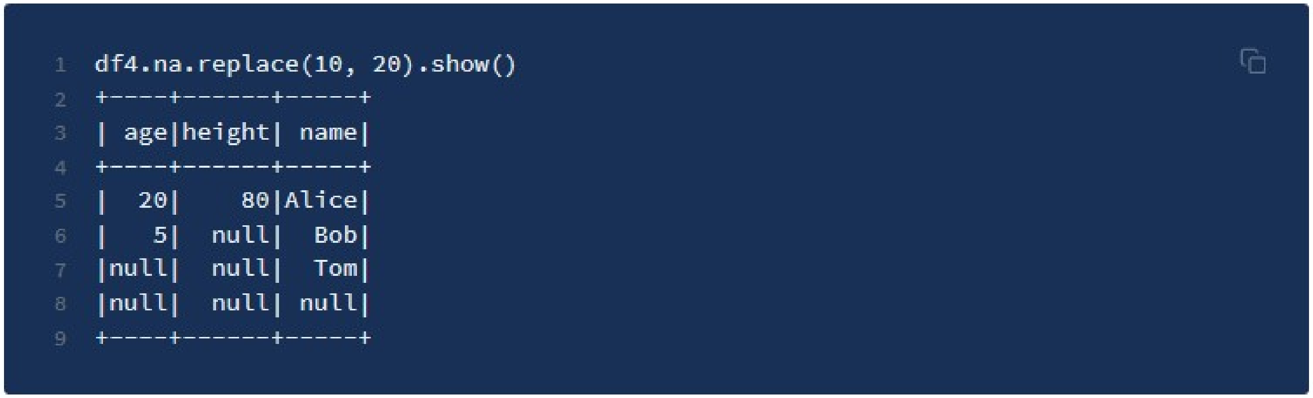


```
df4.na.fill({'age': 50, 'name': 'unknown'}).show()
+---+------+-------+
|age|height|   name|
+---+------+-------+
| 10|    80|  Alice|
|  5|  null|    Bob|
| 50|  null|    Tom|
| 50|  null|unknown|
+---+------+-------+
```

REPLACE

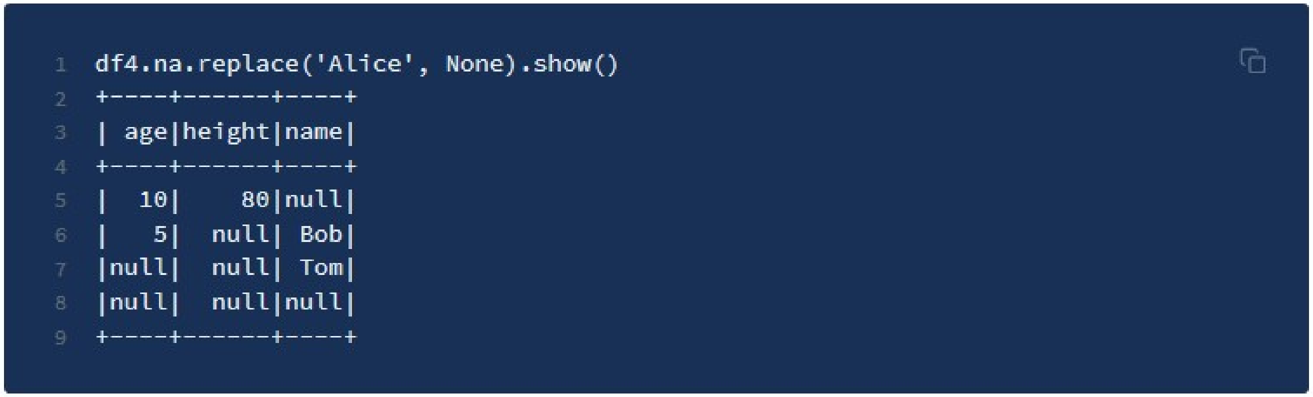


```
df4.show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  10|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

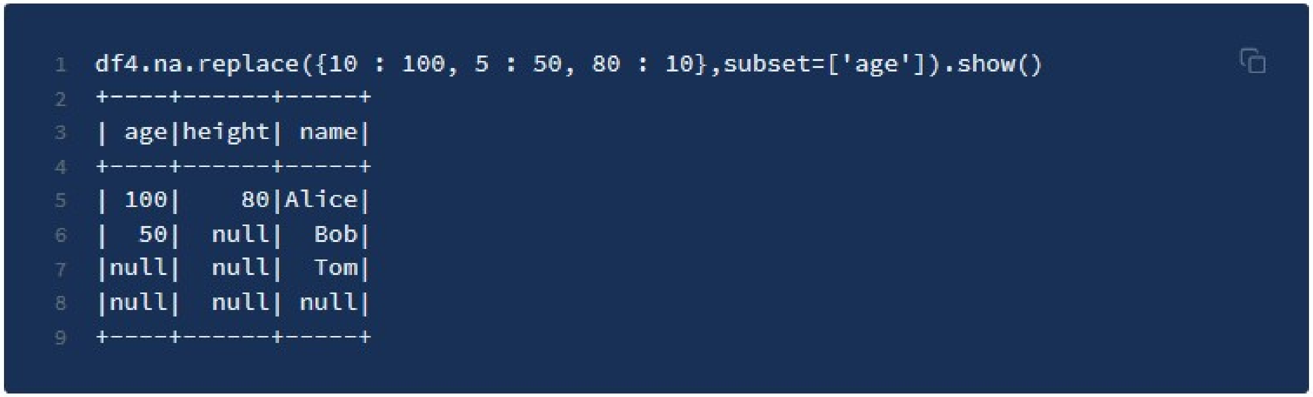


```
df4.na.replace(10, 20).show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  20|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

REPLACE (CONT)



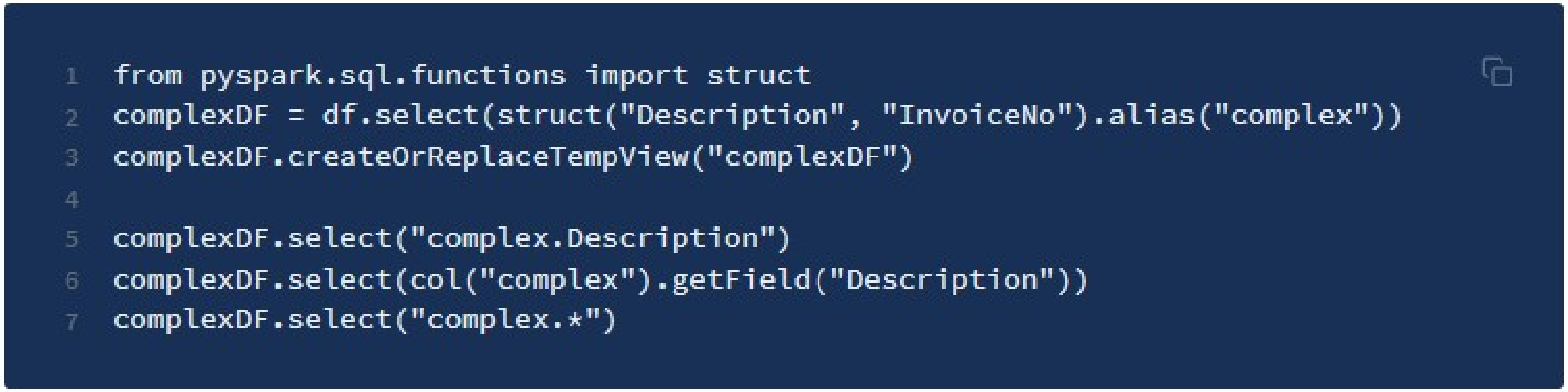
```
df4.na.replace('Alice', None).show()
+----+------+----+
| age|height|name|
+----+------+----+
|  10|    80|null|
|   5|  null| Bob|
|null|  null| Tom|
|null|  null|null|
+----+------+----+
```



```
df4.na.replace({10 : 100, 5 : 50, 80 : 10},subset=['age']).show()
+----+------+-----+
| age|height| name|
+----+------+-----+
| 100|    80|Alice|
|  50|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
```

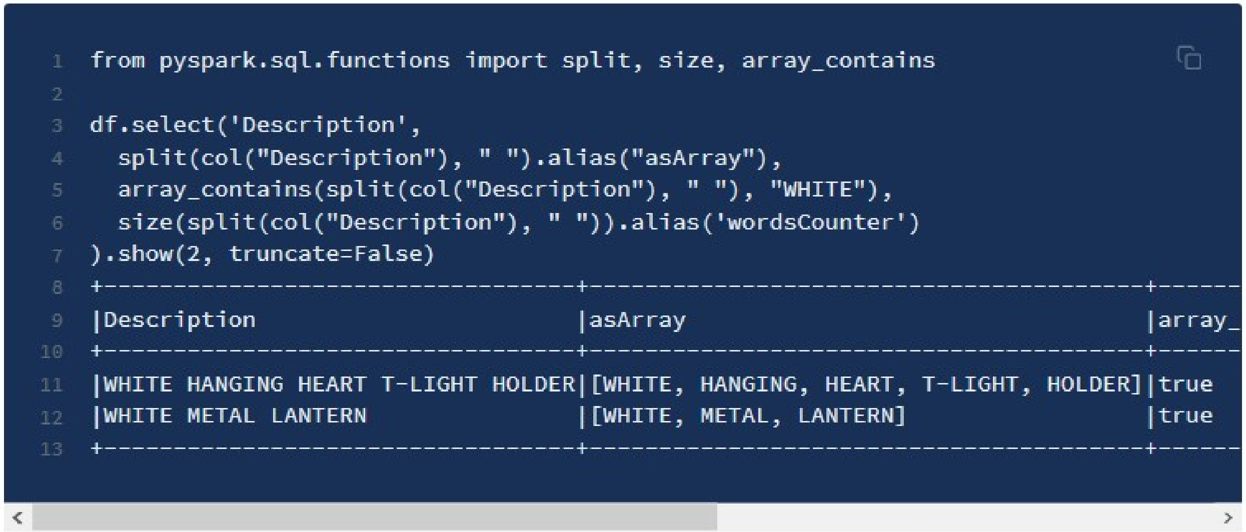
COMPLEX TYPES

STRUCTS



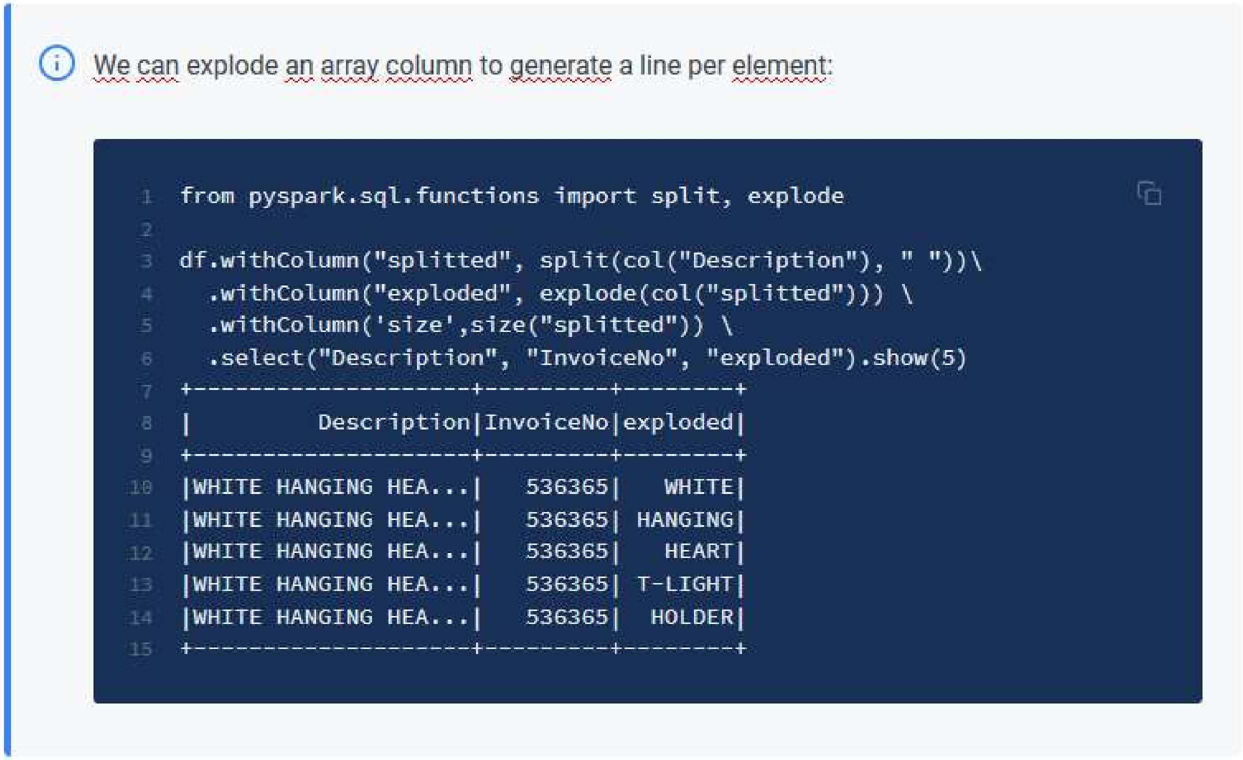
```
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")

complexDF.select("complex.Description")
complexDF.select(col("complex").getField("Description"))
complexDF.select("complex.*")
```

ARRAYS



```
from pyspark.sql.functions import split, size, array_contains

df.select('Description',
  split(col("Description"), " ").alias("asArray"),
  array_contains(split(col("Description"), " "), "WHITE"),
  size(split(col("Description"), " ")).alias('wordsCounter')
).show(2, truncate=False)
+----------------------------------+----------------------------------------+------
|Description                       |asArray                                 |array_
+----------------------------------+----------------------------------------+------
|WHITE HANGING HEART T-LIGHT HOLDER|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|true
|WHITE METAL LANTERN               |[WHITE, METAL, LANTERN]                 |true
+----------------------------------+----------------------------------------+------
```

ARRAYS

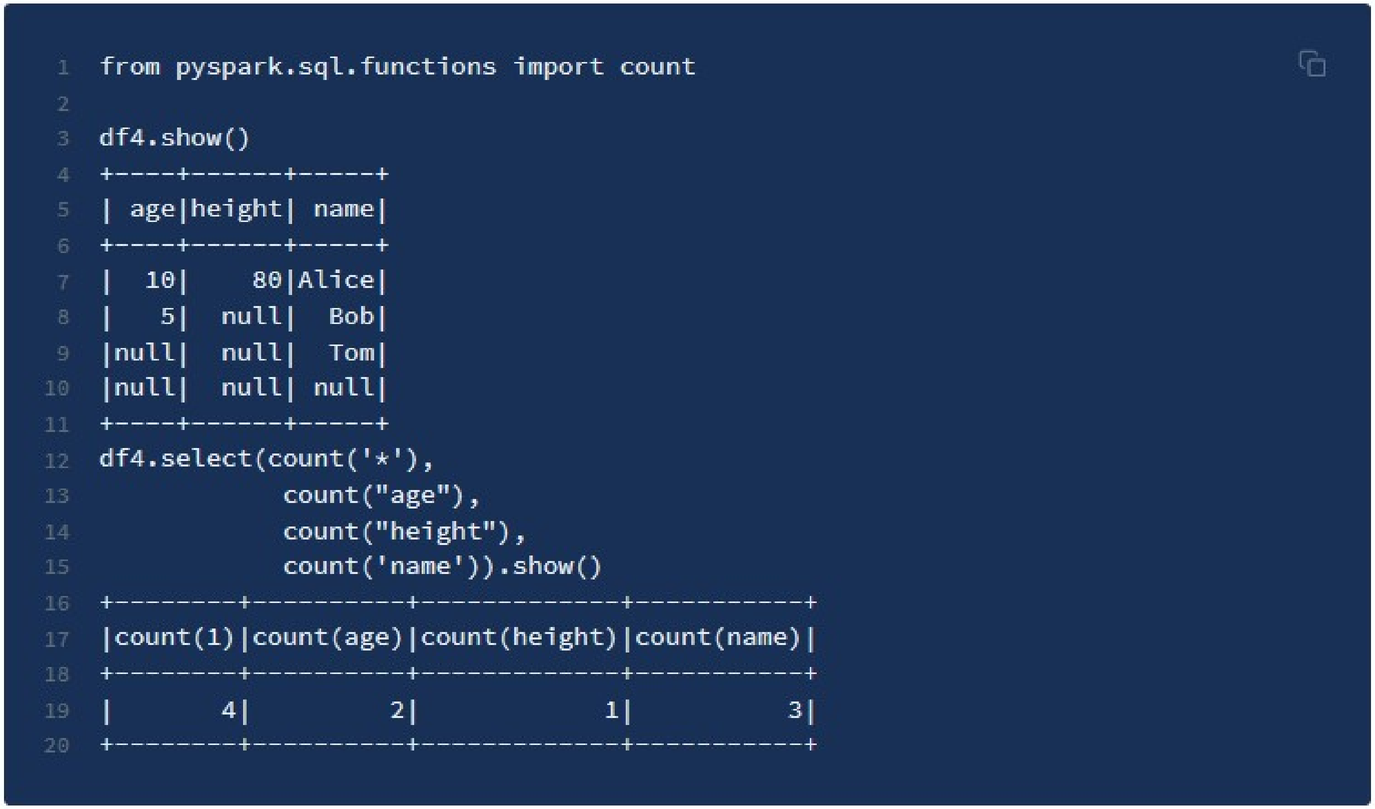


We can explode an array column to generate a line per element:

```
from pyspark.sql.functions import split, explode

df.withColumn("splitted", split(col("Description"), " "))\
  .withColumn("exploded", explode(col("splitted"))) \
  .withColumn('size',size("splitted")) \
  .select("Description", "InvoiceNo", "exploded").show(5)
+--------------------+---------+--------+
|         Description|InvoiceNo|exploded|
+--------------------+---------+--------+
|WHITE HANGING HEA...|   536365|   WHITE|
|WHITE HANGING HEA...|   536365| HANGING|
|WHITE HANGING HEA...|   536365|   HEART|
|WHITE HANGING HEA...|   536365| T-LIGHT|
|WHITE HANGING HEA...|   536365|  HOLDER|
+--------------------+---------+--------+
```

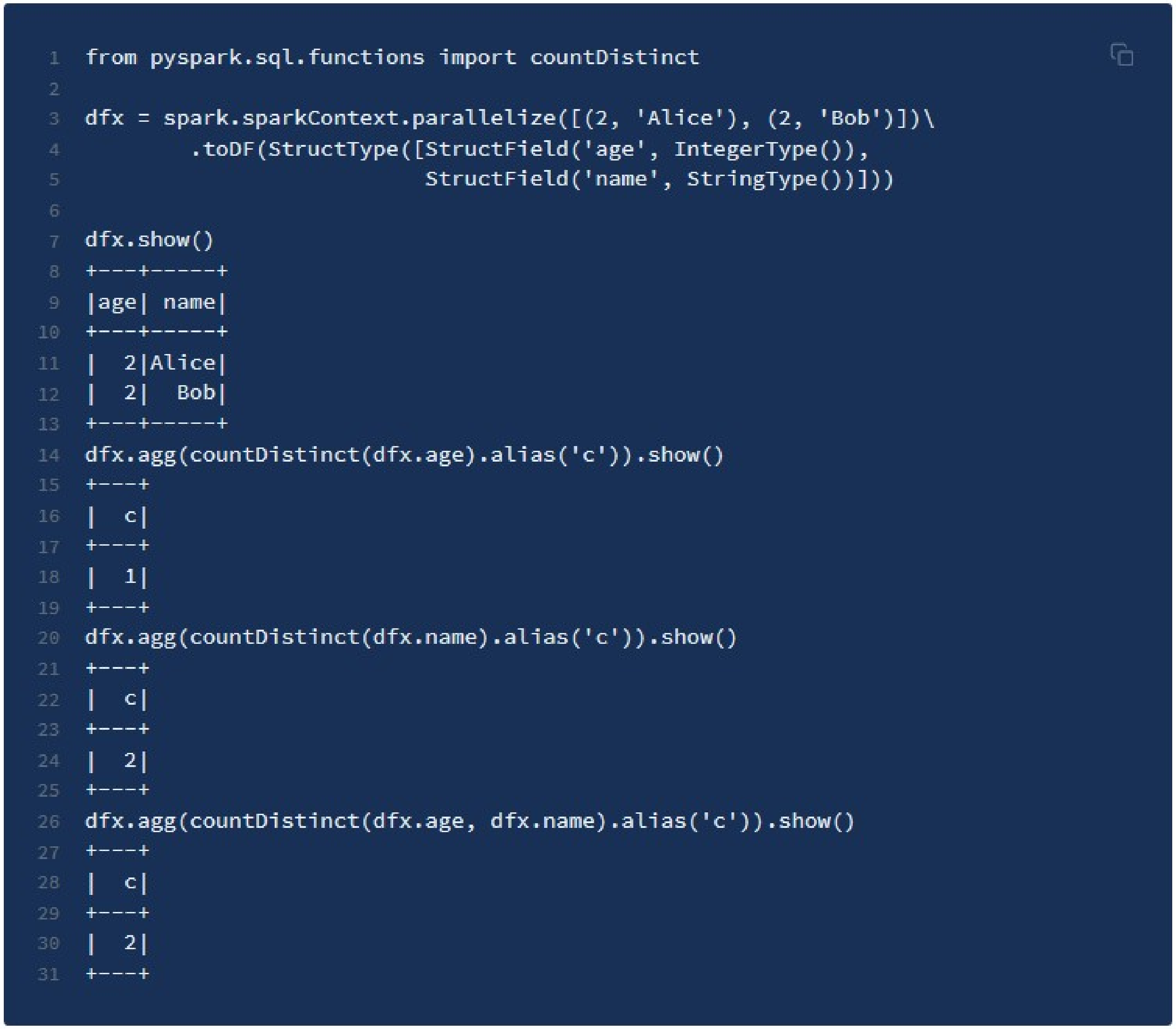
AGGREGATION

COUNT



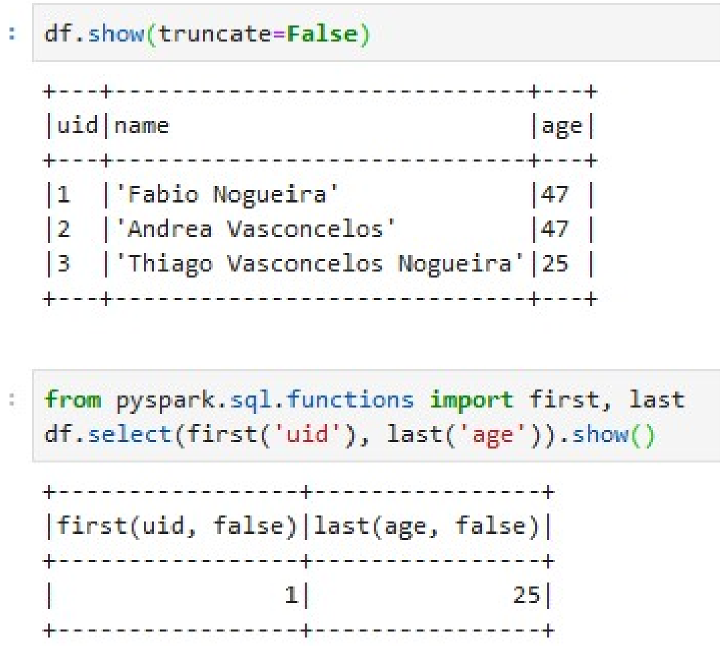
```
from pyspark.sql.functions import count

df4.show()
+----+------+-----+
| age|height| name|
+----+------+-----+
|  10|    80|Alice|
|   5|  null|  Bob|
|null|  null|  Tom|
|null|  null| null|
+----+------+-----+
df4.select(count('*'),
            count("age"),
            count("height"),
            count('name')).show()
+--------+----------+-------------+-----------+
|count(1)|count(age)|count(height)|count(name)|
+--------+----------+-------------+-----------+
|       4|         2|            1|          3|
+--------+----------+-------------+-----------+
```

COUNT



```
from pyspark.sql.functions import countDistinct

dfx = spark.sparkContext.parallelize([(2, 'Alice'), (2, 'Bob')])\
        .toDF(StructType([StructField('age', IntegerType()),
                          StructField('name', StringType())]))

dfx.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  2|  Bob|
+---+-----+
dfx.agg(countDistinct(dfx.age).alias('c')).show()
+---+
|  c|
+---+
|  1|
+---+
dfx.agg(countDistinct(dfx.name).alias('c')).show()
+---+
|  c|
+---+
|  2|
+---+
dfx.agg(countDistinct(dfx.age, dfx.name).alias('c')).show()
+---+
|  c|
+---+
|  2|
+---+
```

FIRST AND LAST



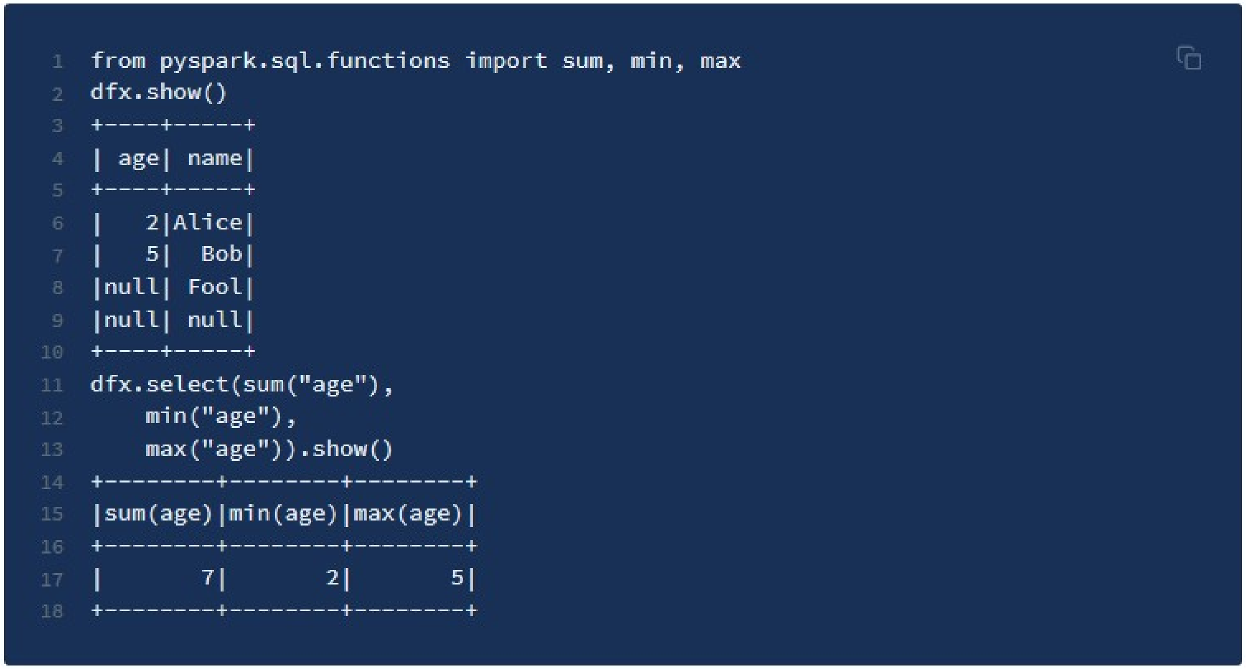
```
df.show(truncate=False)
```

```
+---+-----------------------------+---+
|uid|name                         |age|
+---+-----------------------------+---+
|1  |'Fabio Nogueira'             |47 |
|2  |'Andrea Vasconcelos'         |47 |
|3  |'Thiago Vasconcelos Nogueira'|25 |
+---+-----------------------------+---+
```

```
from pyspark.sql.functions import first, last
df.select(first('uid'), last('age')).show()
```

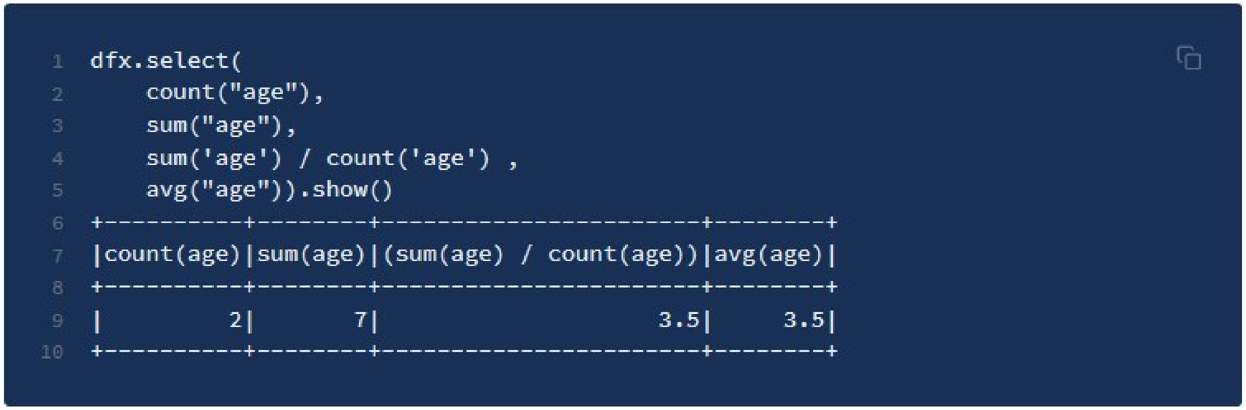
```
+-----------------+----------------+
|first(uid, false)|last(age, false)|
+-----------------+----------------+
|                1|              25|
+-----------------+----------------+
```

SUM, MIN, AND MAX

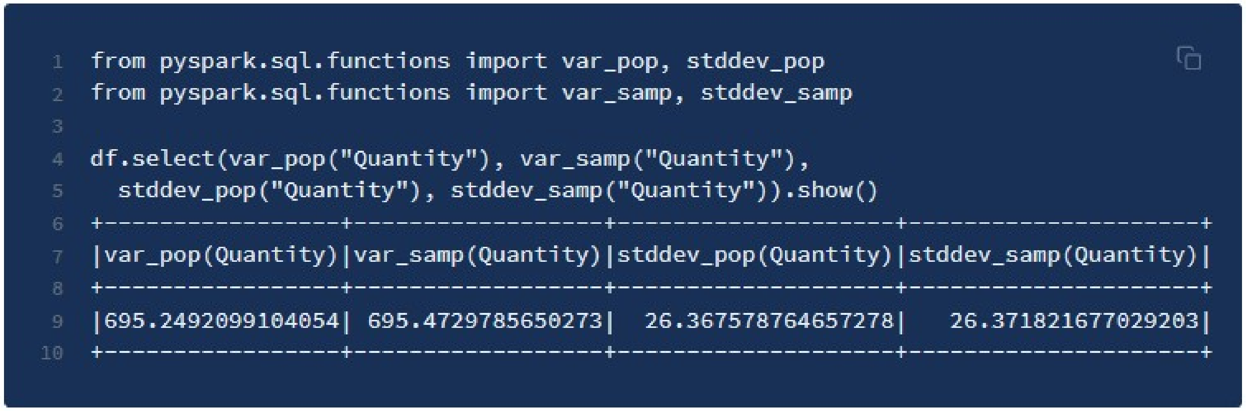


```
from pyspark.sql.functions import sum, min, max
dfx.show()
+----+-----+
| age| name|
+----+-----+
|   2|Alice|
|   5|  Bob|
|null| Fool|
|null| null|
+----+-----+
dfx.select(sum("age"),
    min("age"),
    max("age")).show()
+--------+--------+--------+
|sum(age)|min(age)|max(age)|
+--------+--------+--------+
|       7|       2|       5|
+--------+--------+--------+
```

AVG, VAR, AND STDEV

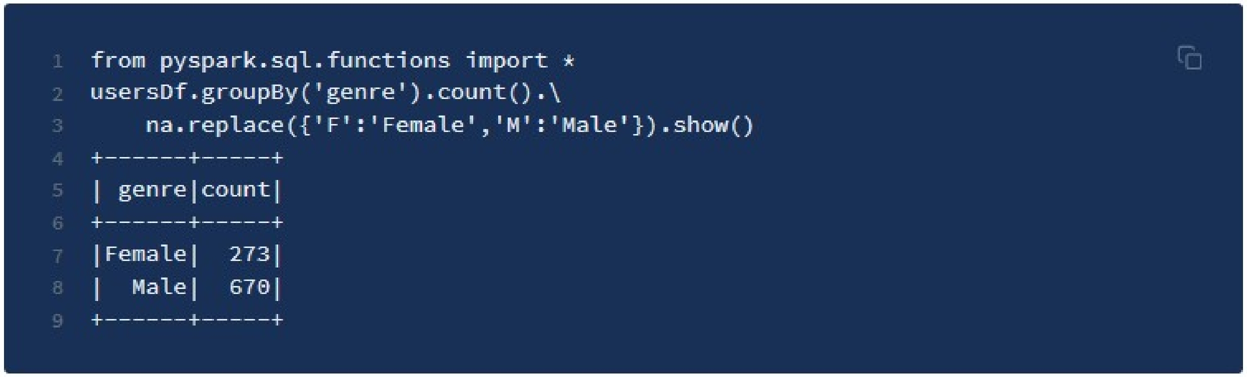


```
dfx.select(
    count("age"),
    sum("age"),
    sum('age') / count('age') ,
    avg("age")).show()
+----------+--------+-----------------------+--------+
|count(age)|sum(age)|(sum(age) / count(age))|avg(age)|
+----------+--------+-----------------------+--------+
|         2|       7|                    3.5|     3.5|
+----------+--------+-----------------------+--------+
```



```
from pyspark.sql.functions import var_pop, stddev_pop
from pyspark.sql.functions import var_samp, stddev_samp

df.select(var_pop("Quantity"), var_samp("Quantity"),
  stddev_pop("Quantity"), stddev_samp("Quantity")).show()
+-----------------+------------------+--------------------+---------------------+
|var_pop(Quantity)|var_samp(Quantity)|stddev_pop(Quantity)|stddev_samp(Quantity)|
+-----------------+------------------+--------------------+---------------------+
|695.2492099104054| 695.4729785650273|  26.367578764657278|   26.371821677029203|
+-----------------+------------------+--------------------+---------------------+
```

GROUP BY



```
from pyspark.sql.functions import *
usersDf.groupBy('genre').count().\
    na.replace({'F':'Female','M':'Male'}).show()
+------+-----+
| genre|count|
+------+-----+
|Female|  273|
|  Male|  670|
+------+-----+
```

QUESTIONS???