HOW TO WRITE SPARK UDFS (USER DEFINED FUNCTIONS) IN PYTHON

In this article, I'll explain how to write user defined functions (UDF) in Python for Apache Spark. The code for this example is here.

(This tutorial is part of our Apache Spark Guide. Use the right-hand menu to navigate.)

Why do you need UDFs?

Spark stores data in dataframes or RDDs—resilient distributed datasets. Think of these like databases. As with a traditional SQL database, e.g. mySQL, you cannot create your own custom function and run that against the database directly. You have to **register the function** first. That is, save it to the database as if it were one of the built-in database functions, like sum(), average, count(),etc.

That's the case with Spark dataframes. With Spark RDDs you can run functions directly against the rows of an RDD.

Three approaches to UDFs

There are three ways to create UDFs:

- df = df.withColumn
- df = sqlContext.sql("sql statement from <df>")
- rdd.map(customFunction())

We show the three approaches below, starting with the first.

Approach 1: withColumn()

Below, we create a simple dataframe and RDD. We write a function to convert the only text field in the data structure to an integer. That is something you might do if, for example, you are working with machine learning where all the data must be converted to numbers before you plug that into an algorithm.

Notice the imports below. Refer to those in each example, so you know what object to import for each of the three approaches.

Below is the complete code for Approach 1. First, we look at key sections. Create a dataframe using the usual approach:

```
df = spark.createDataFrame(data,schema=schema)
```

Now we do two things. First, we create a function **colsInt** and register it. That registered function calls another function **toInt()**, which we don't need to register. The first argument in **udf.register("colsInt", colsInt)** is the name we'll use to refer to the function. The second is the function we want to register.

```
colsInt = udf(lambda z: toInt(z), IntegerType())
spark.udf.register("colsInt", colsInt)

def toInt(s):
    if isinstance(s, str) == True:
        st =
        return(int(''.join(st)))
    else:
         return Null
```

Then we call the function **colinsInt**, like this. The first argument is the name of the new column we want to create. The second is the column in the dataframe to plug into the function.

```
df2 = df.withColumn( 'semployee',colsInt('employee'))
```

Remember that **df** is a column object, not a single employee. That means we have to loop over all rows that column—so we use this lambda (in-line) loop.

```
colsInt = udf(lambda z: toInt(z), IntegerType())
```

Here is Approach 1 all together:

```
import pyspark
from pyspark import SQLContext
from pyspark.sql.types import StructType, StructField, IntegerType,
FloatType, StringType
from pyspark.sql.functions import udf
from pyspark.sql import Row
```

```
conf = pyspark.SparkConf()


sc = pyspark.SparkContext.getOrCreate(conf=conf)
spark = SQLContext(sc)

schema = StructType()

data = ]

df = spark.createDataFrame(data,schema=schema)

colsInt = udf(lambda z: toInt(z), IntegerType())
spark.udf.register("colsInt", colsInt)

def toInt(s):
    if isinstance(s, str) == True:
        st =
        return(int(''.join(st)))
    else:
         return Null


df2 = df.withColumn( 'semployee',colsInt('employee'))
```

Now we show the results. Notice that the new column **semployee** has been added. **withColumn()** creates a new dataframe so we created **df2**.

```
df2.show()

+-----+--------+---+----------+
|sales|employee| ID| semployee|
+-----+--------+---+----------+
| 10.2|    Fred|123|1394624364|
+-----+--------+---+----------+
```

Approach 2: Using SQL

The first step here is to register the dataframe as a table, so we can run SQL statements against it. **df** is the dataframe and **dftab** is the temporary table we create.

```
spark.registerDataFrameAsTable(df, "dftab")
```

Now we create a new dataframe **df3** from the existing on **df** and apply the **colsInt** function to the **employee** column.

```
df3 = spark.sql("select sales, employee, ID, colsInt(employee) as iemployee
from dftab")
```

Here are the results:

```
df3.show()

+-----+--------+---+----------+
|sales|employee| ID| iemployee|
+-----+--------+---+----------+
| 10.2|    Fred|123|1394624364|
+-----+--------+---+----------+
```

Approach 3: RDD Map

A dataframe does not have a map() function. If we want to use that function, we must convert the dataframe to an RDD using **dff.rdd**.

Apply the function like this:

```
rdd = df.rdd.map(toIntEmployee)
```

This passes a row object to the function toIntEmployee. So, we have to return a row object. The RDD is immutable, so we must create a new row.

Below, we refer to the employee element in the row by name and then convert each letter in that field to an integer and concatenate those.

```
def toIntEmployee(rdd):
    s = rdd
    if isinstance(s, str) == True:
        st =
        e = int(''.join(st))
    else:
        e = s
    return Row(rdd,rdd,rdd,e)
```

Now we print the results:

```
for x in rdd.collect():
    print(x)

<row (10.199999809265137, 'Fred', 123, 70114101100)>
```