Lambda Expressions and Functions

- Author: Mahmoud Parsian
- Last updated: December 1, 2022

1. Introduction

For Spark transformations we can either use Lambda Expressions or functions (as I will demonstrate it shortly). Lambda Expressions help you to define a method (or function) without declaring it (kind of a name-less function). So, you do not need a name (as an anonymous method) for the method, return-type, and so on. Lambda Expressions, like anonymous inner classes, provide the way to pass behaviors to functions. Lambda, however, is a much more concise way of writing the code.

In PySpark, you may use Lambda expressions (as a replacement for anonymous functions — a function without a name), when you want to run a function once and don't need to save it for future reuse.

2. Why do you need a UDF?

- UDF's are used to extend the functions of the framework Re-use these UDF on multiple DataFrame's.
- For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark built-in features don't have this function hence you can create it a UDF and reuse this as needed on many Data Frames.
- UDF's are once created they can be re-used on several DataFrame's and SQL expressions.

3. Python and Lambda Expressions

The Python programming language supports the creation of anonymous functions (i.e. functions defined without an explicit name), using a construct called `lambda`.

The general structure of a lambda function is:

```
lambda <args>: <expr>

where
      <args> is a list of arguments
      <expr> is an expression
```

Consider the following Python function which triples the value of a scalar:

```
def triple(x):
   return 3*x
```

For instance to use this function:

```
print(triple(10))
 30
```

The same function can be written as a lambda expression:

```
triple_as_lambda = lambda x: 3*x
```

And you call it as:

```
print(triple_as_lambda(10))
30
```

4. Syntax of Lambda Expressions

What is the syntax of Lambda expressions? A Lambda expression consists of two parts: 0 or more arguments and the function body:

- Lambda Expression with one argument:

```
lambda argument : function
```

- Lambda Expression with two or more arguments:

```
lambda argument-1, argument-2, ... : function
```

- Lambda Expression with `if-else`

```
# Let a1, a2, ... be Lambda Expression arguments
# if condition is true, then {
#   return Expression1
# }
# else {
#   return Expression2
# }
lambda a1, a2, ...: Expression1 if condition else Expression2
```

5. Lambda Expressions in PySpark

If you are not used to Lambda expressions, defining Python functions and then passing in function names to Spark transformations will work as well. The Spark documentation seems to use Lambda expressions in all of the PySpark examples. So it is better to get used to Lambda expressions.

Lambda expressions can have only one statement which returns the value (value can be a simple data type — such as Integer or String — or it can be a composite data type — such as an array, tuple, list, or dictionary). In case you need to have multiple statements in your functions, you need to use the pattern of defining explicit functions and passing in their names.

6. Example 1: Word Count

Let `records` be an `RDD[String]`. Then, we may replace the following transformation, which uses Lambda expression:

```
# data: list of strings
# records: RDD[String]
# words: RDD[String]
# convert all words to lowercase and flatten it to words
# spark is an instance of a SparkSession
data = ["fox jumped high", "fox jumped high again"]
records = spark.sparkContext.parallelize(data)
words = records.flatMap(lambda line: line.lower().split(" "))
```

with the following transformation, which uses Python functions:

First define your desired function:

```
# line as a string of words
def flatten_words(line):
    return line.lower().split(" ")
#end-def
```

Then use the defined function instead of a Lambda Expression:

```
# convert all words to lowercase and flatten it to words
words = records.flatMap(flatten_words)
```

7. Example 2: Sum Up Frequencies Per Key

Let us say that we want to add up values per key, but ignore the negative numbers. This means that if a key has all negative numbers, then that key will be dropped from the final result. For example

```
('A', 3), ('A', 4), ('A', -2) => ('A', 7)
('B', -3), ('B', 4), ('B', 5) => ('B', 9)
('C', 4) => ('C', 4)
('D', -7), ('D', -9) => dropped from result
('E', -5) => dropped from result
('F', -2), ('F', 0) => ('F', 0)
```

7.1. Solution by Lambda Expressions

Let `pairs` be an `RDD[(String, Integer)]`. First, we drop (key, value) pairs if the value is less than zero. Then, we sum up the values per key.

```
# data: list of pairs
# pairs: RDD[(String, Integer)]
# results: RDD[(String, Integer)]
# sum up values per key
# spark is an instance of a SparkSession
>>> data = [('A', 3), ('A', 4), ('A', -2),
        ('B', -3), ('B', 4), ('B', 5),
        ('C', 4),
        ('D', -7), ('D', -9),
        ('E', -5),
        ('F', -2), ('F', 0)]
>>> pairs = spark.sparkContext.parallelize(data)
```

```
>>> positives = pairs.filter(lambda x: x[1] >= 0)
>>> results = positives.reduceByKey(lambda x, y: x+y)
>>> results.collect()
[('B', 9), ('C', 4), ('A', 7), ('F', 0)]
```

7.2. Solution by Functions

Let `pairs` be an `RDD[(String, Integer)]`. First, we drop (key, value) pairs if the value is less than zero. Then, we sum up the values per key.

First we define some basic functions for filtering and sum up.

- Filter function

```
# filter negative numbers
# pair: (key, value)
def drop_negatives(pair):
  value = pair[1]
  if value >= 0:
    return True
  else:
    return False
#end-def
```

- Sum up function

```
# add two numbers
def add_numbers(x, y):
  return x+y
#end-def
```

Now, let's rewrite the transformations by our defined functions:

```
# pairs: RDD[(String, Integer)]
# results: RDD[(String, Integer)]
# sum up values per key
# spark is an instance of a SparkSession
>>> data = [('A', 3), ('A', 4), ('A', -2),
        ('B', -3), ('B', 4), ('B', 5),
        ('C', 4),
        ('D', -7), ('D', -9),
```

```
        ('E', -5),
        ('F', -2), ('F', 0)]
>>> pairs = spark.sparkContext.parallelize(data)
>>> positives = pairs.filter(drop_negatives)
>>> results = positives.reduceByKey(add_numbers)
>>> results.collect()
[('B', 9), ('C', 4), ('A', 7), ('F', 0)]
```

8. Example 3: Lambda Expressions with `if-else`

Given an RDD[Integer], let's implement the following logic (expressed as a pseudo code) on the given RDD:

```
if (x < 2) {
   return x*10
}
else {
   if (x < 4) {
      return x**2
   }
   else {
      return x+10
   }
}
```

Let's implement this logic for an RDD[Integer]:

```
# data: list of integers
# spark is an instance of a SparkSession
>>> data = [1, 2, 3, 4, 5, 6, 7]
>>> numbers = spark.sparkContext.parallelize(data)
>>> results = numbers.map(lambda x: x*10 if x<2 else (x**2 if x<4 else x+10))
>>> results.collect()
[10, 4, 9, 14, 15, 16, 17]
```

The same transformation can be implemented by a Python function:

```
def demo_if_else(x):
  if x < 2:
    return x*10
  else:
    if x < 4:
```

```
      return x**2
    else:
      return x+10
#end-def
```

Now, we use the define Python function:

```
# data: list of integers
# spark is an instance of a SparkSession
>>> data = [1, 2, 3, 4, 5, 6, 7]
>>> numbers = spark.sparkContext.parallelize(data)
>>> results = numbers.map(demo_if_else)
>>> results.collect()
[10, 4, 9, 14, 15, 16, 17]
```

9. Example 4: UDF Example with Annotation

In this section, I provide detailed examples on how to create and use a UDF in PySpark.

1. Create a DataFrame
2. Create a Python Function
3. Test your Python Function
4. Convert a Python function to PySpark UDF
5. Using UDF with DataFrame “select”
6. Using UDF with DataFrame “addColumn”
7. Registering PySpark UDF & use it on SQL
8. Creating UDF using annotation

9.1. 1 Create a DataFrame

DataFrames can be created from collections, RDDs, files, and many other data sources.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
column_names = ["ID", "Name"]
my_data = [("100", "john jones"),
```

```
           ("200", "tracey smith"),
           ("300", "amy sanders")]
df = spark.createDataFrame(data=my_data,schema=column_names)
df.show(truncate=False)
+-----+------------+
|ID   |Name        |
+-----+------------+
|100  |john jones  |
|200  |tracey smith|
|300  |amy sanders |
+-----+------------+
```

9.2. 2 Create a Python Function

The first step in creating a UDF is creating a Python function. The convertCase() function takes a string parameter and converts the first letter of every word to capital letter. UDF's take parameters of your choice and returns a value.

```
def convert_case(name):
  result_string = ""
  arr = name.split(" ")
  for x in arr:
    result_string +=  x[0:1].upper() + x[1:len(x)] + " "
  #end-for
  return result_string.strip()
#end-def
```

9.3. 3 Test created Python function

```
>>> def convert_case(name):
...   result_string = ""
...   arr = name.split(" ")
...   for x in arr:
...     result_string +=  x[0:1].upper() + x[1:len(x)] + " "
...   #end-for
...   return result_string.strip()
... #end-def
...
>>> convert_case("alex smith")
'Alex Smith'
>>> convert_case("alex jr smith")
```

```
'Alex Jr Smith'
>>>
```

9.4. 4 Convert a Python function to UDF

```
# Converting a Python function to UDF
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
# return type is StringType()
convert_case_udf = udf(
   lambda p: convert_case(p),
   StringType()
)
```

9.5. 5 Using UDF with DataFrame `select()` function

Now you can use `convert_case_udf()` on a DataFrame column:

```
from pyspark.sql.functions import col
df.select(col("ID"), \
    convert_case_udf(col("Name")).alias("FullName") ) \
   .show(truncate=False)
+-----+------------+
|ID   |FullName    |
+-----+------------+
|100  |John Jones  |
|200  |Tracey Smith|
|300  |Amy Sanders |
+-----+------------+
```

9.6. 6 Using UDF with DataFrame `addColumn()` function

```
df.withColumn("FullName", converted_case_udf(col("Name"))) \
  .show(truncate=False)
+-----+--------------+-------------+
|ID   | Name         |FullName     |
+-----+--------------+-------------+
|100  | john jones   | John Jones  |
|200  | tracey smith | Tracey Smith|
|300  | Amy Sanders  | Amy Sanders |
+-----+--------------+-------------+
```

9.7. 7 Registering PySpark UDF & use it on SQL

In order to use convert_case() function on PySpark SQL, you need to register the function with PySpark by using `spark.udf.register()`.

```
spark.udf.register("convert_UDF", convert_case, StringType())
df.createOrReplaceTempView("MY_TABLE")
spark.sql("select ID, convert_UDF(Name) as Name from MY_TABLE")
     .show(truncate=False)
+-----+------------+
|ID   |Name        |
+-----+------------+
|100  |John Jones  |
|200  |Tracey Smith|
|300  |Amy Sanders |
+-----+------------+
```

9.8. 8 Creating UDF using annotation

There are many ways to create a UDF. You may use Spark's annotation (`@udf`) to create a UDF.

```
# import required libraries
from pyspark.sql.types import StringType
from pyspark.sql.functions import udf

# use annotation to create a UDF
@udf(returnType=StringType())
def upper_case(str):
  return str.upper()

df.withColumn("UpperName", upper_case(col("Name")))
.show(truncate=False)
+-----+--------------+--------------+
|ID   | Name         |UpperName     |
+-----+--------------+--------------+
|100  | john jones   | JOHN JONES   |
|200  | tracey smith | TRACEY SMITH |
|300  | Amy Sanders  | AMY SANDERS  |
+-----+--------------+--------------+
```