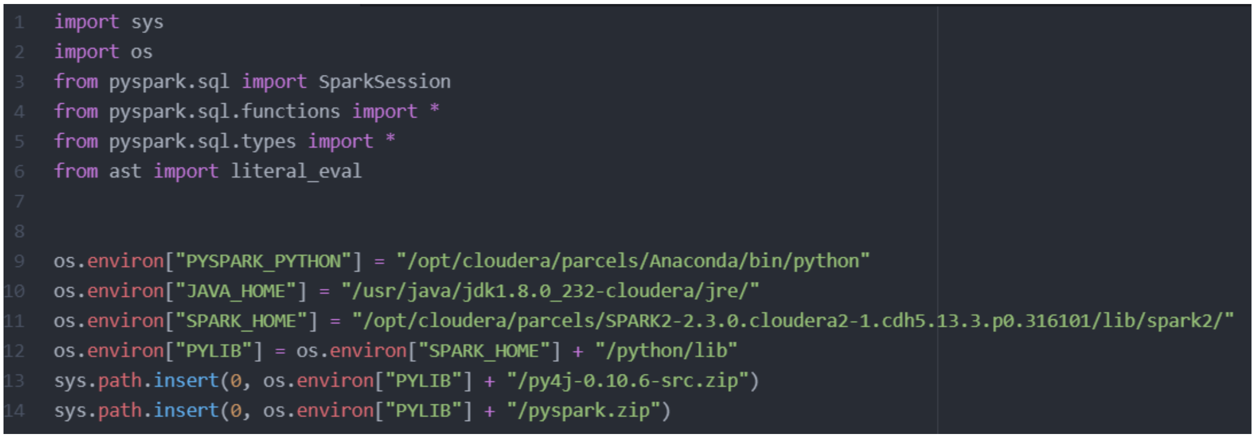
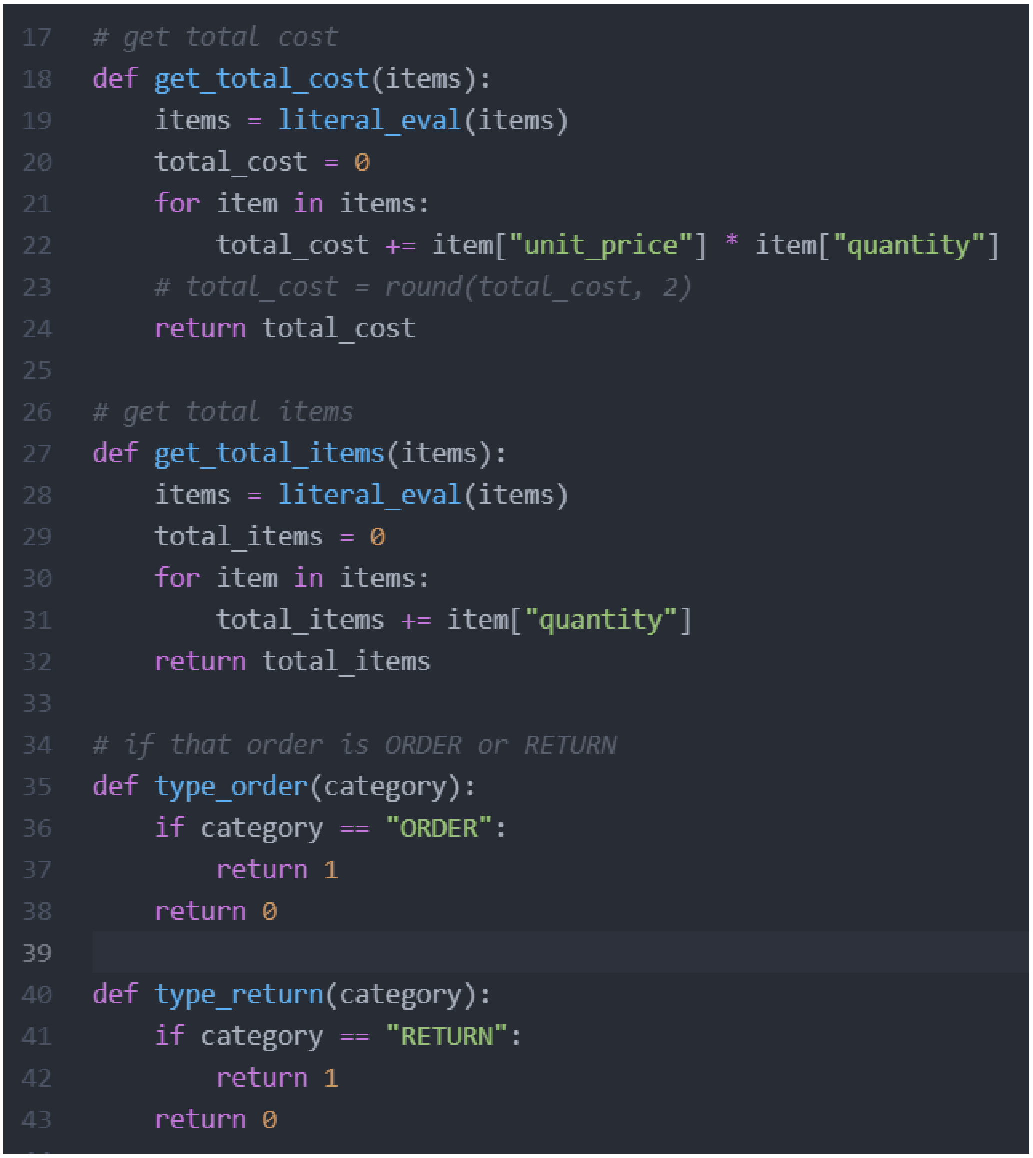
Code Logic - Retail Data Analysis



```
1   import sys
2   import os
3   from pyspark.sql import SparkSession
4   from pyspark.sql.functions import *
5   from pyspark.sql.types import *
6   from ast import literal_eval
7
8
9   os.environ["PYSPARK_PYTHON"] = "/opt/cloudera/parcels/Anaconda/bin/python"
10  os.environ["JAVA_HOME"] = "/usr/java/jdk1.8.0_232-cloudera/jre/"
11  os.environ["SPARK_HOME"] = "/opt/cloudera/parcels/SPARK2-2.3.0.cloudera2-1.cdh5.13.3.p0.316101/lib/spark2/"
12  os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
13  sys.path.insert(0, os.environ["PYLIB"] + "/py4j-0.10.6-src.zip")
14  sys.path.insert(0, os.environ["PYLIB"] + "/pyspark.zip")
```

Here at lines 1-6, required python modules and functions are imported. literal_eval() function will be used to convert string from items column into a proper python list of dictionary.

At lines 9-14, necessary environment setups for running the code in cloudera instance are given here.

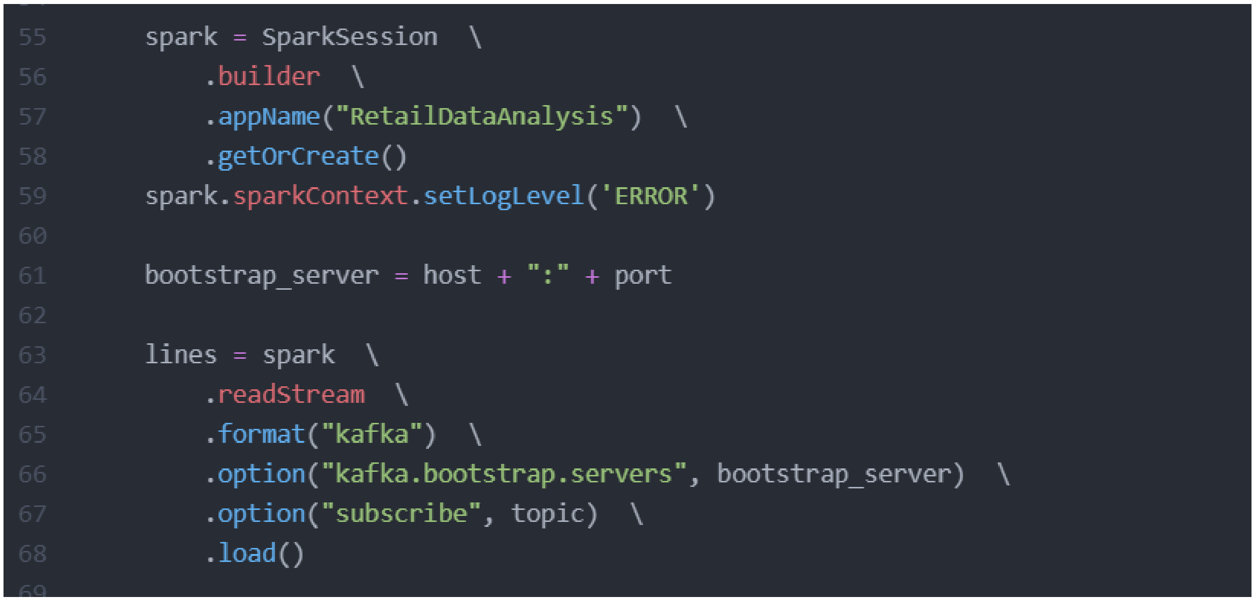


```python
# get total cost
def get_total_cost(items):
    items = literal_eval(items)
    total_cost = 0
    for item in items:
        total_cost += item["unit_price"] * item["quantity"]
    # total_cost = round(total_cost, 2)
    return total_cost

# get total items
def get_total_items(items):
    items = literal_eval(items)
    total_items = 0
    for item in items:
        total_items += item["quantity"]
    return total_items

# if that order is ORDER or RETURN
def type_order(category):
    if category == "ORDER":
        return 1
    return 0

def type_return(category):
    if category == "RETURN":
        return 1
    return 0
```

These are custom functions:

1. get_total_cost() function is used to calculate total cost by summing every multiplication of each item in each order. The formula is $\sum unitprice * quantity$.
2. get_total_items() function is used to retrieve total items by just summing the quantity of each ordered item. The formula is $\sum quantity$.

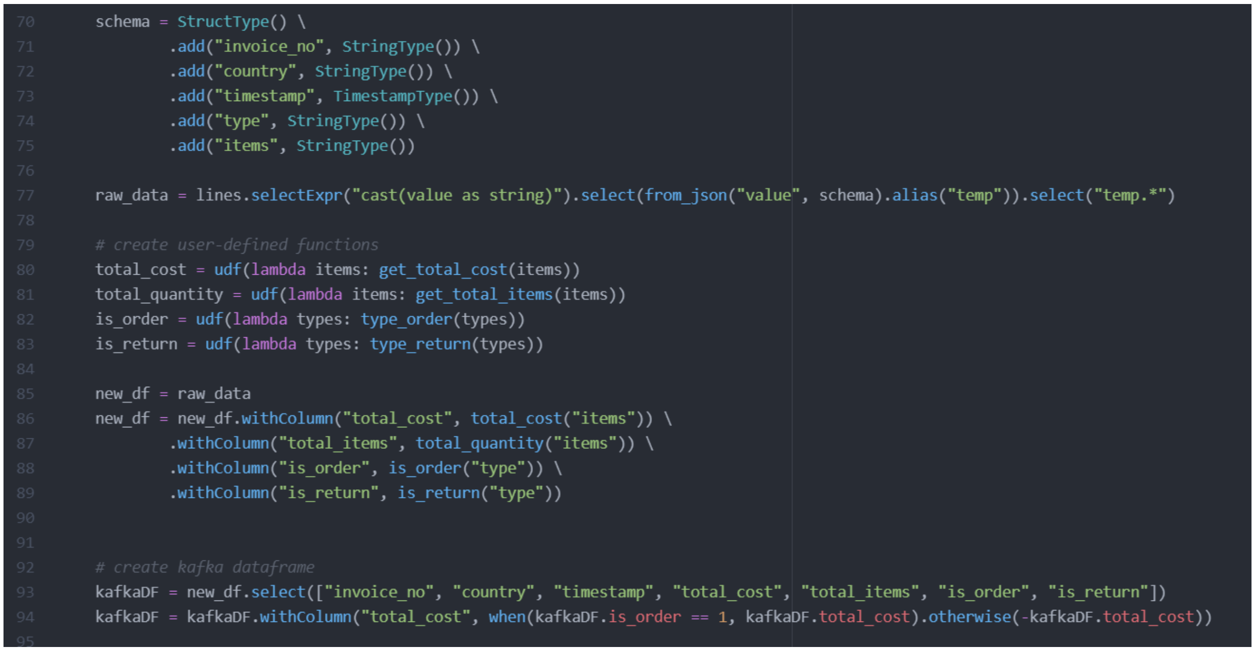
3. type_order() function is used to map type of order if the type is “ORDER”, return 1. Otherwise, return 0.
4. type_return() function is used to map type of order if the type is “RETURN”, return 1. Otherwise, return 0.



```
55      spark = SparkSession  \
56          .builder  \
57          .appName("RetailDataAnalysis")  \
58          .getOrCreate()
59      spark.sparkContext.setLogLevel('ERROR')
60
61      bootstrap_server = host + ":" + port
62
63      lines = spark  \
64          .readStream  \
65          .format("kafka")  \
66          .option("kafka.bootstrap.servers", bootstrap_server)  \
67          .option("subscribe", topic)  \
68          .load()
```

At lines 55-59, spark session is created as well as setting up Log Level.

At lines 63-68 is the beginning of the spark streaming job for reading streaming data from Kafka bootstrap server received from the command line arguments.



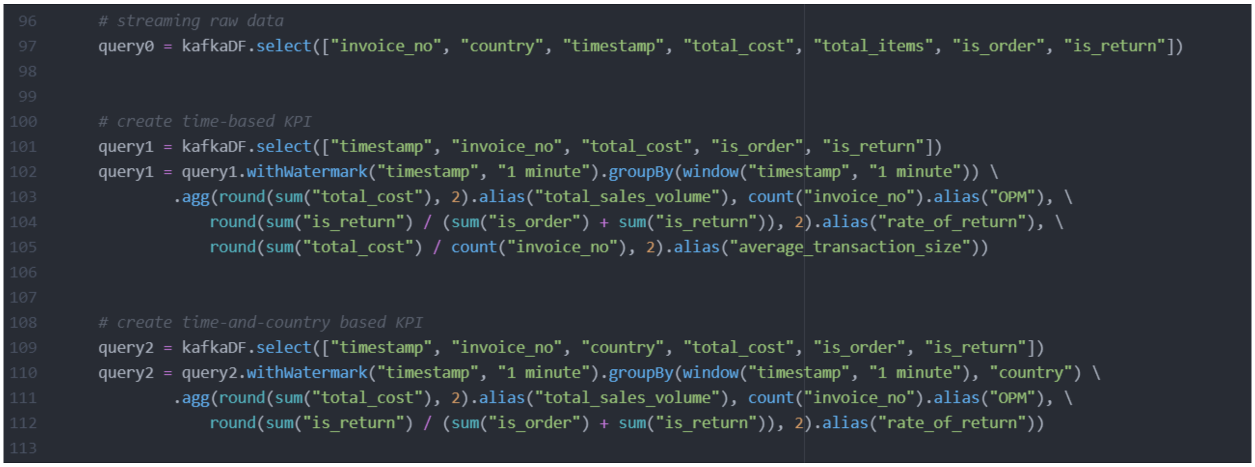
```
70    schema = StructType() \
71            .add("invoice_no", StringType()) \
72            .add("country", StringType()) \
73            .add("timestamp", TimestampType()) \
74            .add("type", StringType()) \
75            .add("items", StringType())
76
77    raw_data = lines.selectExpr("cast(value as string)").select(from_json("value", schema).alias("temp")).select("temp.*")
78
79    # create user-defined functions
80    total_cost = udf(lambda items: get_total_cost(items))
81    total_quantity = udf(lambda items: get_total_items(items))
82    is_order = udf(lambda types: type_order(types))
83    is_return = udf(lambda types: type_return(types))
84
85    new_df = raw_data
86    new_df = new_df.withColumn("total_cost", total_cost("items")) \
87            .withColumn("total_items", total_quantity("items")) \
88            .withColumn("is_order", is_order("type")) \
89            .withColumn("is_return", is_return("type"))
90
91
92    # create kafka dataframe
93    kafkaDF = new_df.select(["invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return"])
94    kafkaDF = kafkaDF.withColumn("total_cost", when(kafkaDF.is_order == 1, kafkaDF.total_cost).otherwise(-kafkaDF.total_cost))
95
```

At lines 70-75, the schema is created. There is invoice_no, country, timestamp, type and items. "timestamp” column is set as TimestampType to get proper timestamp.

Code in line 77 reads the data in sql dataframe format.

At line 80-83, user-defined functions are created and will return custom function outputs.

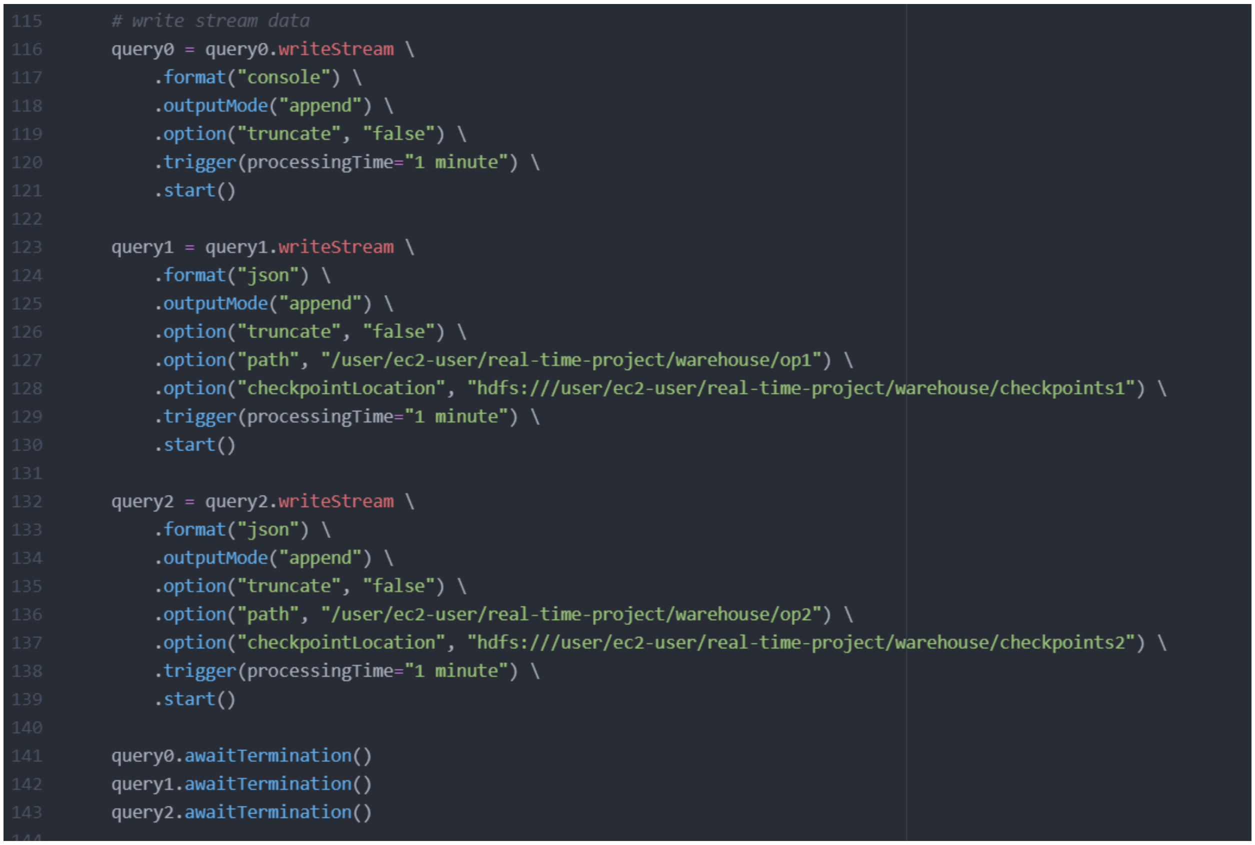
At line 85-94, 4 new columns(total_cost, total_items, is_order and is_return) are created and values in total_cost column will become negative if is_order is equal to 0, otherwise they will stay the same.



```
96      # streaming raw data
97      query0 = kafkaDF.select(["invoice_no", "country", "timestamp", "total_cost", "total_items", "is_order", "is_return"])
98
99
100     # create time-based KPI
101     query1 = kafkaDF.select(["timestamp", "invoice_no", "total_cost", "is_order", "is_return"])
102     query1 = query1.withWatermark("timestamp", "1 minute").groupBy(window("timestamp", "1 minute")) \
103             .agg(round(sum("total_cost"), 2).alias("total_sales_volume"), count("invoice_no").alias("OPM"), \
104                 round(sum("is_return") / (sum("is_order") + sum("is_return")), 2).alias("rate_of_return"), \
105                 round(sum("total_cost") / count("invoice_no"), 2).alias("average_transaction_size"))
106
107
108     # create time-and-country based KPI
109     query2 = kafkaDF.select(["timestamp", "invoice_no", "country", "total_cost", "is_order", "is_return"])
110     query2 = query2.withWatermark("timestamp", "1 minute").groupBy(window("timestamp", "1 minute"), "country") \
111             .agg(round(sum("total_cost"), 2).alias("total_sales_volume"), count("invoice_no").alias("OPM"), \
112                 round(sum("is_return") / (sum("is_order") + sum("is_return")), 2).alias("rate_of_return"))
113
```

Here, from line 92 to line 112, batch SQL dataframes with proper schema are created. query0 for console output, query1 for time-based KPI and query2 for time-and-country based KPI.

To calculate the KPIs, it can be done by just summing total costs because the values of total_cost column are already in decent positive and negative values that were transformed earlier.



```
    # write stream data
    query0 = query0.writeStream \
        .format("console") \
        .outputMode("append") \
        .option("truncate", "false") \
        .trigger(processingTime="1 minute") \
        .start()

    query1 = query1.writeStream \
        .format("json") \
        .outputMode("append") \
        .option("truncate", "false") \
        .option("path", "/user/ec2-user/real-time-project/warehouse/op1") \
        .option("checkpointLocation", "hdfs:///user/ec2-user/real-time-project/warehouse/checkpoints1") \
        .trigger(processingTime="1 minute") \
        .start()

    query2 = query2.writeStream \
        .format("json") \
        .outputMode("append") \
        .option("truncate", "false") \
        .option("path", "/user/ec2-user/real-time-project/warehouse/op2") \
        .option("checkpointLocation", "hdfs:///user/ec2-user/real-time-project/warehouse/checkpoints2") \
        .trigger(processingTime="1 minute") \
        .start()

    query0.awaitTermination()
    query1.awaitTermination()
    query2.awaitTermination()
```

Here, all dataframes will be written and wait for their termination.

Command used to run :

```
spark2-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.2 spark-streaming.py 18.211.252.152 9092 real-time-project
```