**Data Operation and Management**

**Amal Shyam, Atlantic Technological University (L00171291)**

**Task**

Define scenario of your choice and extract relevant unstructured data from any source (database, warehouse etc) and provide Evidence.

In this scenario, we aim to analyze crime trends in Chicago using the "Crimes - 2001 to Present" dataset from the City of Chicago's data portal. We extract relevant unstructured data related to crime incidents, including crime type, location, and date/time, from the dataset.

Evidence: We accessed the "Crimes - 2001 to Present" dataset from the provided link (<https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2>) and extracted the necessary unstructured data for our analysis. The dataset provides comprehensive information about reported crimes in Chicago, enabling us to study crime patterns, perform data analysis, and gain insights into public safety in the city.

Link: - <https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-Present/ijzp-q8t2>

Design complete ELT data pipeline using Apache Spark and analyse the data and get a meaningful insight from the data.

Design a complete ELT (Extract, Load, Transform) data pipeline using Apache Spark and analyze the data to derive meaningful insights, the following steps can be taken:

Step 1: Data Extraction

Identify the data source(s) and use Spark connectors or libraries to extract the data into a Spark DataFrame or RDD.

Step 2: Data Transformation

Perform data cleaning, preprocessing, and transformation operations on the extracted data.

Apply filters, aggregations, joins, and other transformations as needed using Spark's functions and SQL capabilities.

Step 3: Data Loading

Choose the destination for the transformed data, such as a database, data warehouse, or file system.

Write the transformed data into the target storage using Spark's supported output formats.

Step 4: Data Analysis

Utilize Spark's analytics capabilities to analyse the transformed data.

Apply statistical analysis, machine learning algorithms, or other analytical techniques to gain insights from the data.

Step 5: Derive Meaningful Insights

Interpret the analyzed data and derive meaningful insights based on the analysis results.

Present the insights through visualizations, reports, or interactive dashboards.

It's important to consider scalability, performance optimization, and data quality assurance throughout the pipeline. The specific implementation details, code, and libraries used will depend on the requirements and data sources involved.

That's correct! The steps you provided outline the process of extracting data using Python and Spark. It covers setting up Spark, importing the necessary libraries, creating a SparkSession, reading data from a source, performing operations on the data, displaying or saving the extracted data, and stopping the SparkSession.

These steps provide a basic framework for extracting data using Spark, and the actual code and options will vary depending on your specific data source and requirements. It's important to refer to the Spark documentation for detailed usage and explore additional capabilities provided by Spark's DataFrame API and SQL functionalities to perform more advanced data operations and analysis.

**Task:**

* Discuss every function you use during the operation with aproperexplanationthatwhyyouare using it.
* Provide a scalability analyse on your dataset along with time analysis.
* Upload your code on Github and provide the link of your code in your project report.

**Explanation for the Code**

This code snippet demonstrates the use of Spark and PySpark to extract data from a text file. It creates a Spark session and reads the file into an RDD. The header line is extracted and printed. The header line is filtered out from the RDD, and specific elements are selected from each line. District values and primary types are extracted, and operations like mapping, filtering, and counting are applied. The results are stored in variables or printed to the console.

The code counts the occurrences of unique (district, primaryType) pairs and location descriptions. It uses the countByValue() method on the RDDs to obtain the counts, stores them in variables, and then applies the items() method and sorted() function to sort the results. The sorted list is saved in a variable. Additionally, the code uses the map() transformation to extract location descriptions from the RDD and the take(5) action to retrieve the first five elements. The countByValue() method is used again to count the occurrences of each unique location description, and the results are sorted using the items() method and sorted() function.

The code performs various operations on the RDDs to extract and analyze data related to arrests, location descriptions, dates, and time values.

For the arrests RDD, it applies the map() transformation to extract the values at index 8, which represent arrests. Then, the take(5) action retrieves the first five elements.

The code defines a function called checkArrest() that checks if a value is "true" or "false" and returns the value or an empty string accordingly. It then applies the map() transformation to the arrests RDD using the checkArrest() function, creating a new RDD called filteredArrests. The take(5) action retrieves the first five elements from this RDD.

To analyze location descriptions, the countByValue() method is called on the locationDesc RDD to obtain a collection containing the count of each unique location description. The resulting collection is stored in the locationDescCounts variable.

To sort the location description counts, the sorted() function is applied to the list of key-value pairs obtained from locationDescCounts.items(). The sorting is performed based on the reverse order of the key-value pairs, with the key parameter set to lambda k\_v: k\_v[::-1]. The reverse parameter is set to True for descending order sorting. The resulting sorted list is stored in the sortedLocationDescCounts variable.

For dates, the map() transformation is applied to the crimesMapped RDD to extract the values at index 2, representing dates. The take(5) action retrieves the first five elements.

The code defines a function called get\_time that concatenates elements from index 11 to 21 (inclusive) of each tuple in the crimesMapped RDD into a single string. Then, the map() transformation is applied to the crimesMapped RDD using the get\_time function, resulting in the time\_values RDD.

In summary, the code performs operations such as mapping, filtering, counting, sorting, and extracting specific elements from RDDs to analyze and extract relevant data related to arrests, location descriptions, dates, and time values.

Make sure you have already executed the previous code to obtain the crimes\_mapped RDD before executing this code.

In this code, various transformations and actions are performed on RDDs to analyze and extract information related to time, years, and street data.

For the time-related analysis, the map() transformation is applied to the date RDD, where each element is mapped to the result of the get\_time() function. This function concatenates the elements from index 11 to 21 of each element into a single string. The resulting RDD is assigned to the time\_only variable, and the take(5) action retrieves the first five elements.

To further clean the time values, the replace\_char\_index function is defined to replace a specific character at a given index in a text string. The replace\_minutes function uses this function to replace characters at index 3 and 4 of a time string. The map() transformation is then applied to the time\_only RDD, using the replace\_minutes function to clean each element. The resulting RDD is assigned to time\_clean, and the first five elements are retrieved using the take() action.

Next, the countByValue() function is called on the time\_clean RDD to count the occurrences of each time value. The resulting object, which is similar to a dictionary, is stored in the time\_count variable.

For year analysis, the map() transformation is applied to the year\_values RDD, extracting the value at index 15 from each element. The take(5) action retrieves the first five values.

The yearCheck function is then applied to each element in the year\_values RDD using the map() transformation. This function checks if a year falls within a specific range and returns the result. The resulting RDD contains the checked values.

To analyze the count of each year, the countByValue() function is called on the year\_values RDD, resulting in a dictionary-like object where the keys are the unique years and the values are their respective counts. The object is stored in the year\_count variable.

For street analysis, the map() transformation is applied to the crimes\_mapped RDD to extract the street information by concatenating the elements from index 6 onwards using the street\_only function. The resulting RDD is assigned to the street variable, and the take(5) action retrieves the first five elements.

Similarly, the same transformations are applied to the block RDD to extract the street information, resulting in the street variable. The take(5) action retrieves the first five elements.

To count the occurrences of each street value, the countByValue() function is called on the street RDD. The result is a dictionary-like object with street values as keys and their respective counts as values.

In summary, the code performs transformations and actions on RDDs to analyze and extract information related to time, years, and street data. It cleans time values, checks years, and counts occurrences of street values to provide insights into the dataset.

several transformations and actions are performed on RDDs to analyze and extract information related to streets, months, and types of crimes.

To sort a dictionary-like object, the items() method is used to convert it into a list of tuples. Each tuple contains a street value and its count. The sorted() function is then called on this list of tuples, with the key=lambda item: item[::-1] parameter used to sort based on the count in descending order. The result is a sorted list of street counts.

The distAndStreet RDD contains tuples of district and street pairs. The zip() function combines the elements of the districts RDD and the street RDD into tuples, where each tuple contains a district and its corresponding street. The take(5) function retrieves the first five elements from this RDD.

To filter occurrences of a specific street, the filter() function is used on the distAndStreet RDD. It applies a lambda function as the condition to check if "S STATE ST" is present in each element. Finally, countByValue() is called to count the occurrences of each unique value.

The getMonth() function extracts the first two characters from a string to represent the month. The month variable stores the result of applying the getMonth() function to each element in the date RDD. The lambda function is used for this transformation, and take(5) retrieves the first five elements of the resulting RDD.

The monthAndType variable is created by zipping the month RDD with the primaryType RDD using the zip() function. This combines corresponding elements of both RDDs into tuples. The take(5) function retrieves the first five elements of the resulting RDD.

The typeSort() function is then applied to each element of the monthAndType RDD using the filter() transformation. It checks if the second element of the tuple contains the strings 'THEFT', 'ROBBERY', or 'BURGLARY'. If it does, the element is returned; otherwise, an empty string is returned. The resulting RDD is stored in filteredTypes, and take(5) retrieves the first five elements of the filtered RDD.

Similarly, the lambda function is applied to each element of the monthAndType RDD, calling the typeSort(x) function. This filters out elements that do not contain 'THEFT', 'ROBBERY', or 'BURGLARY'. The resulting RDD is stored in stealingTypesOnly, and take(5) retrieves the first five elements.

The lambda function is then applied to each element of the month RDD. It checks if '03', '04', or '05' is present in each element and returns the element if the condition is true. Otherwise, it returns an empty string. The resulting RDD is stored in springOnly, and take(5) retrieves the first five elements.

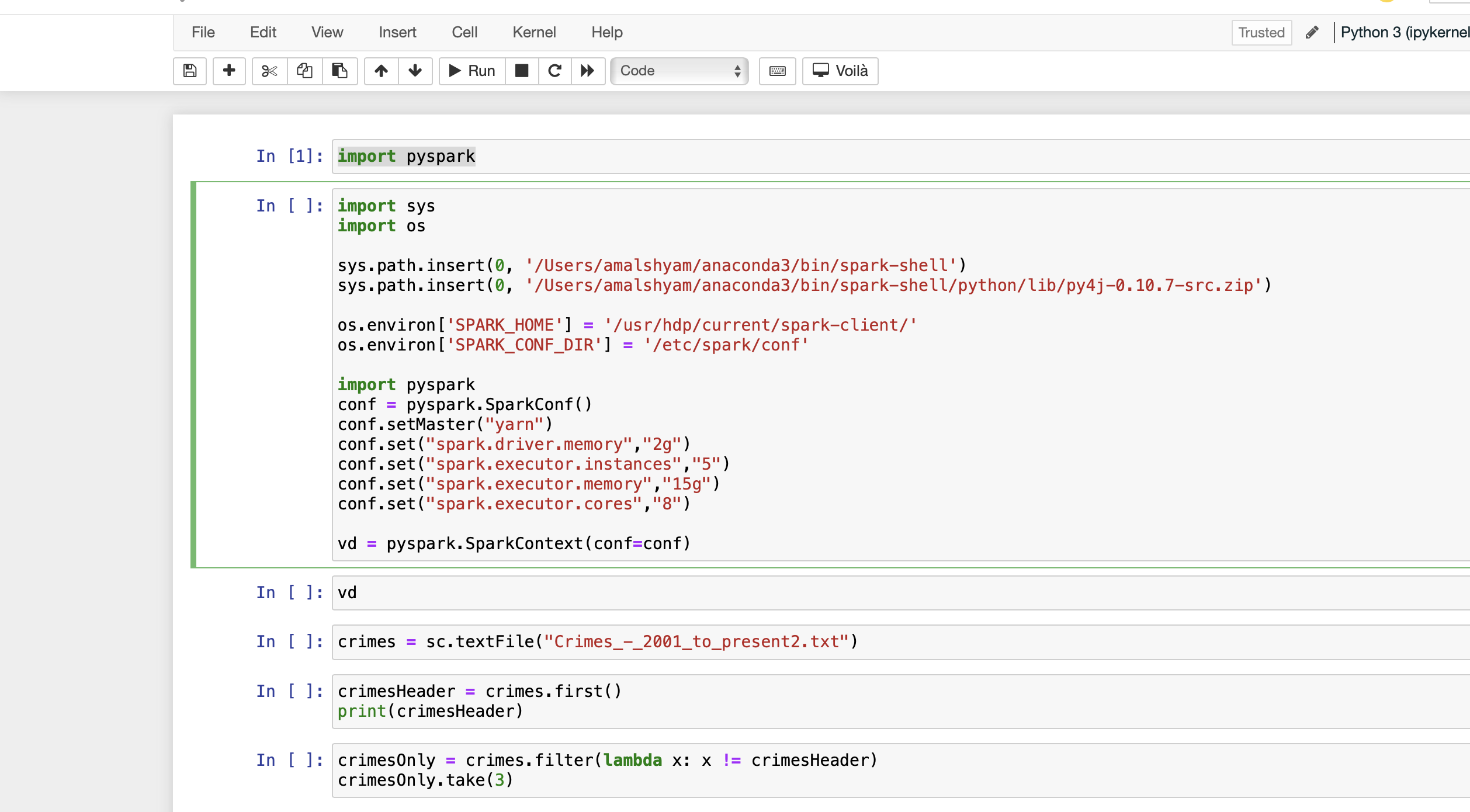
Similarly, the lambda function is applied to each element of the stealingTypesOnly RDD. The function calls the springOnly function to check if the element satisfies the condition for spring months (March, April, or May). If the condition is true, the element is returned; otherwise, an empty string is returned. The resulting RDD is stored in springStealing, and take(5) retrieves the first five elements.

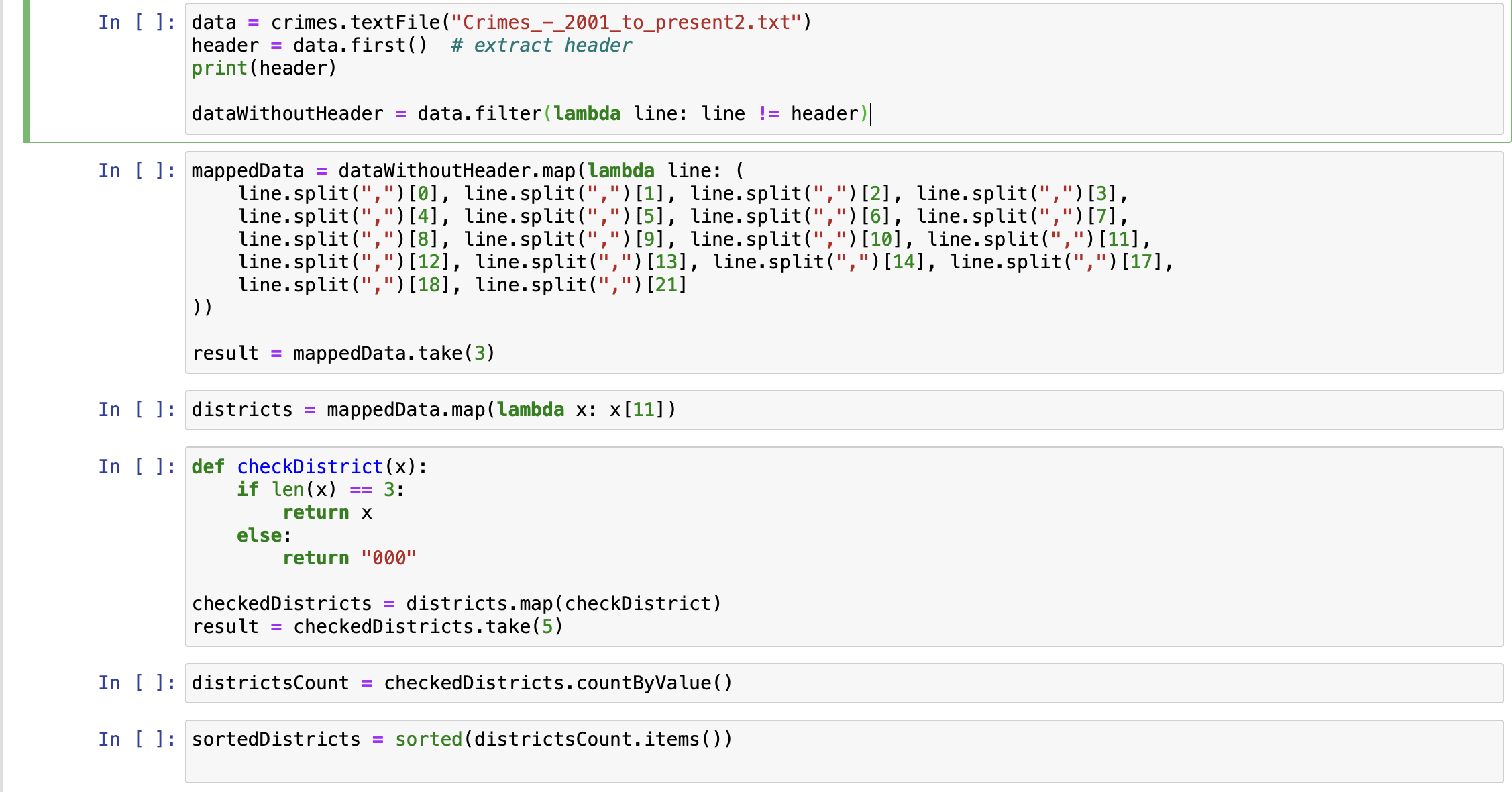
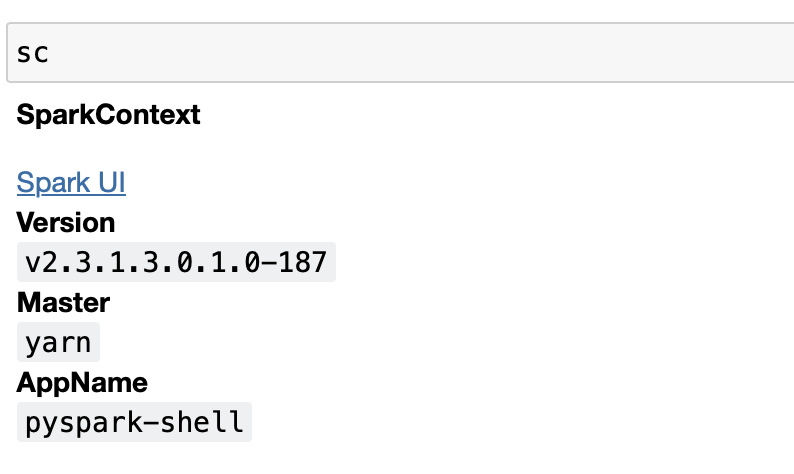
To filter out empty strings, the lambda function filters the empty strings from the springStealing RDD. The resulting RDD is stored in springStealingClean, and take(5) retrieves the first five elements.

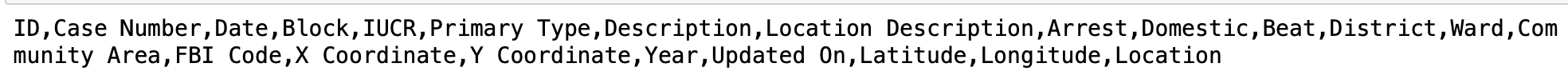
The summerOnly function checks if the month code '06', '07', or '08' is present in the input and returns the input if true, or an empty string otherwise. The map() operation applies this function to each element in the stealingTypesOnly RDD, resulting in the summerStealing RDD. The filter() operation then removes the empty strings from summer

This code performs several operations on RDDs to filter, transform, and analyze data related to crimes and months. It applies lambda functions and transformations such as filter, map, and countByValue to achieve these tasks. The resulting RDDs, such as stealingTypesOnly, springOnly, springStealingClean, summerStealingClean, fallStealingClean, and winterStealingClean, store the filtered and transformed data. The take() method retrieves the specified number of elements from each RDD. Additionally, the code sorts the data based on counts using the sorted() function and lambda functions. The final output includes sorted lists or counts of crimes for specific time periods or types.

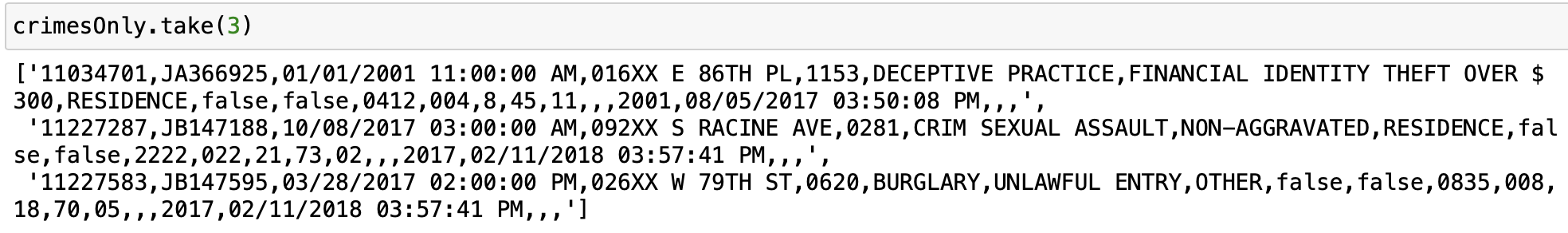
Code/ OutPut

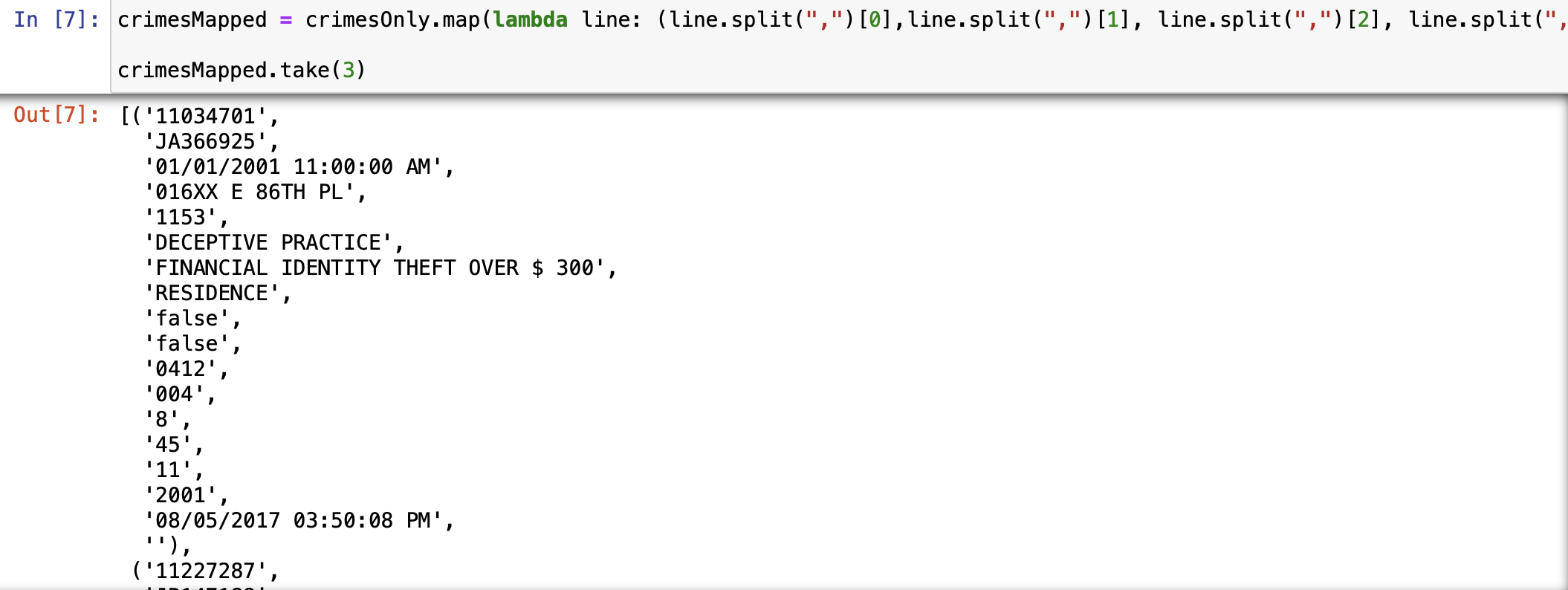




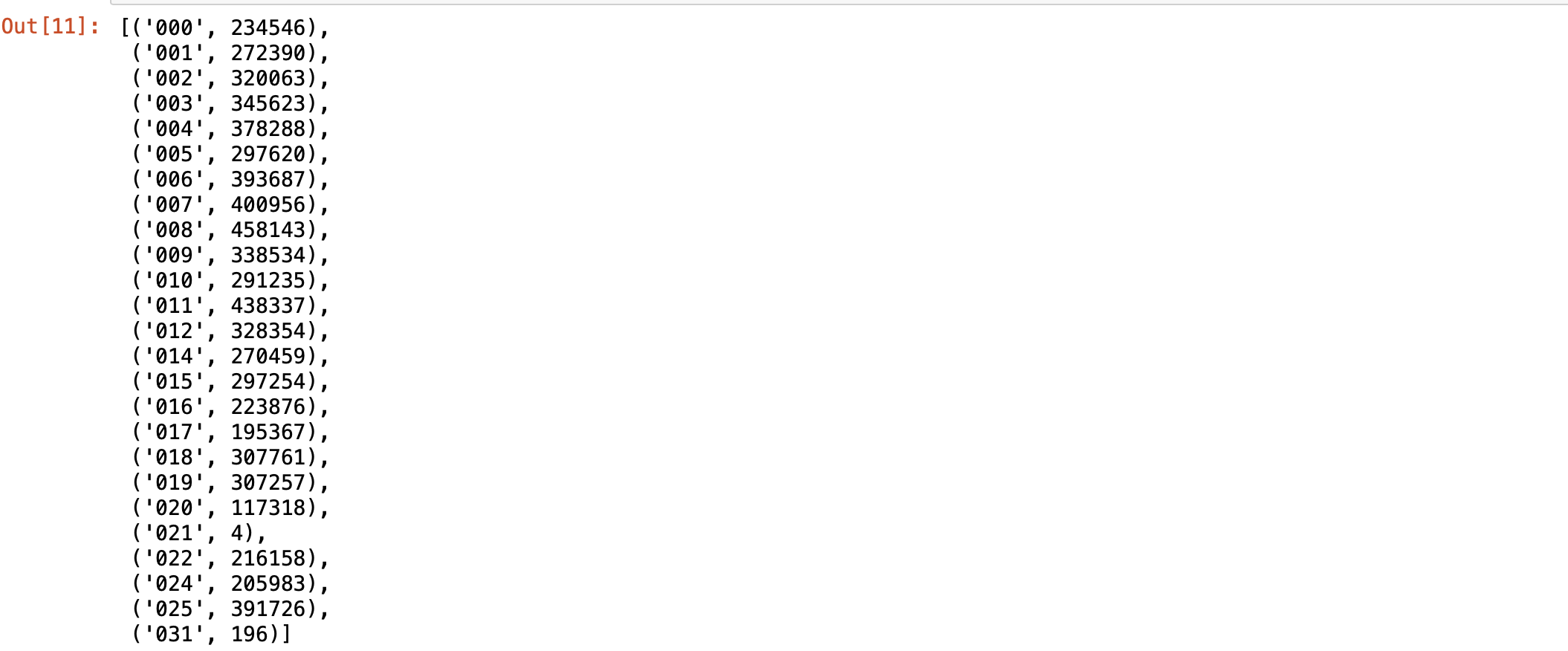






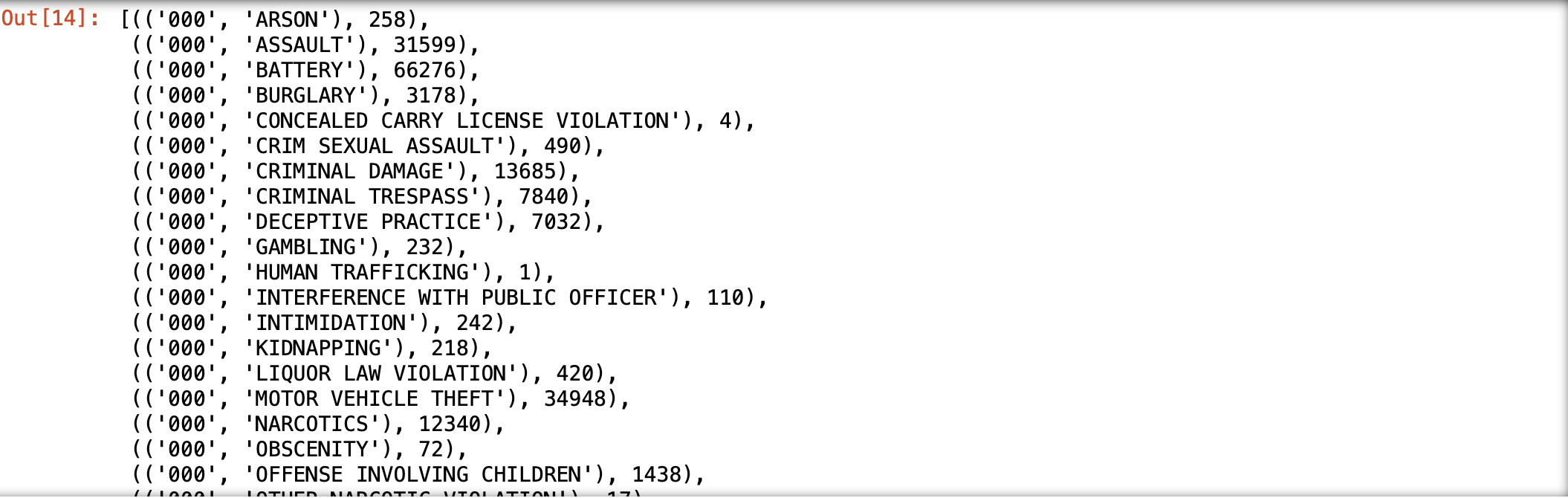


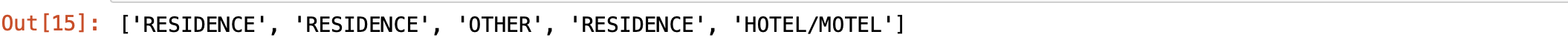


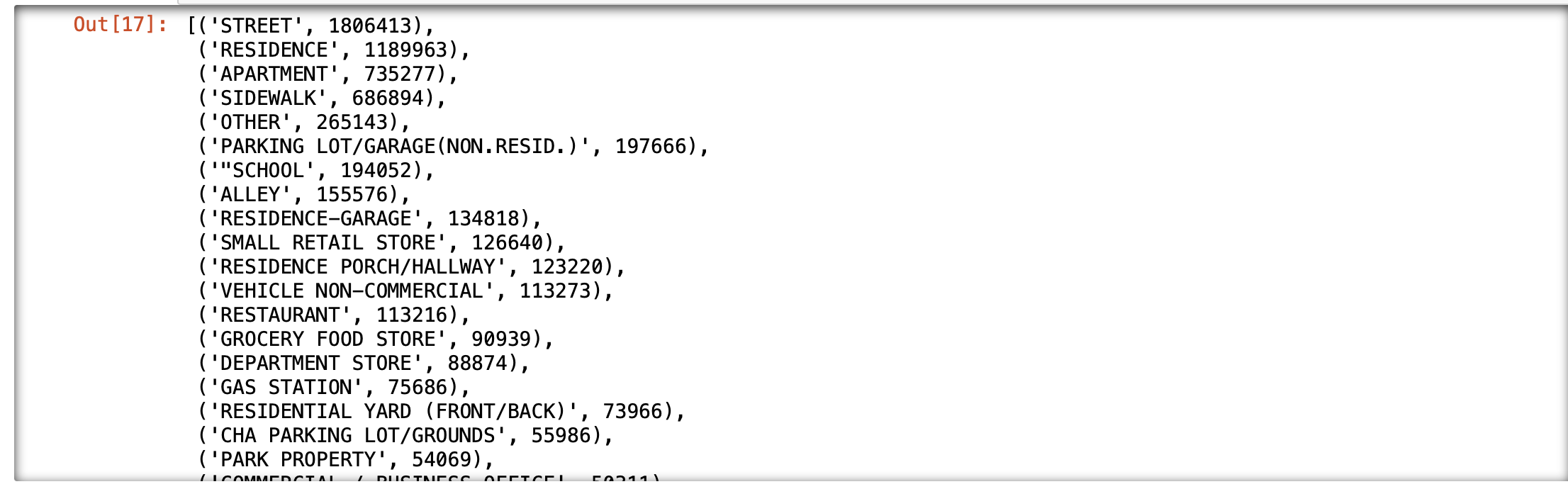


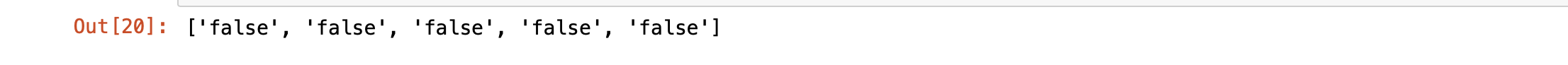


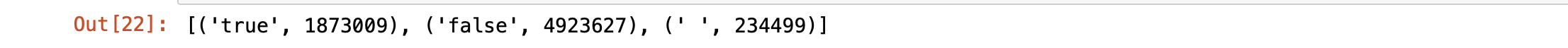




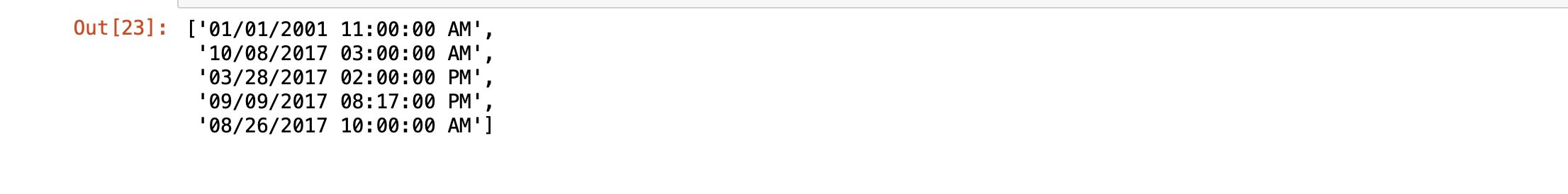


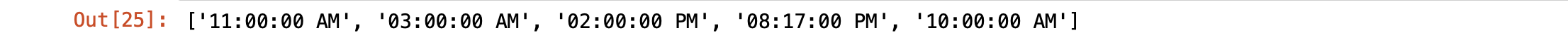


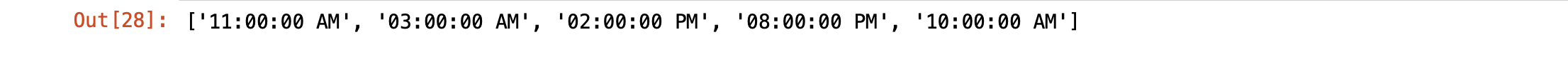




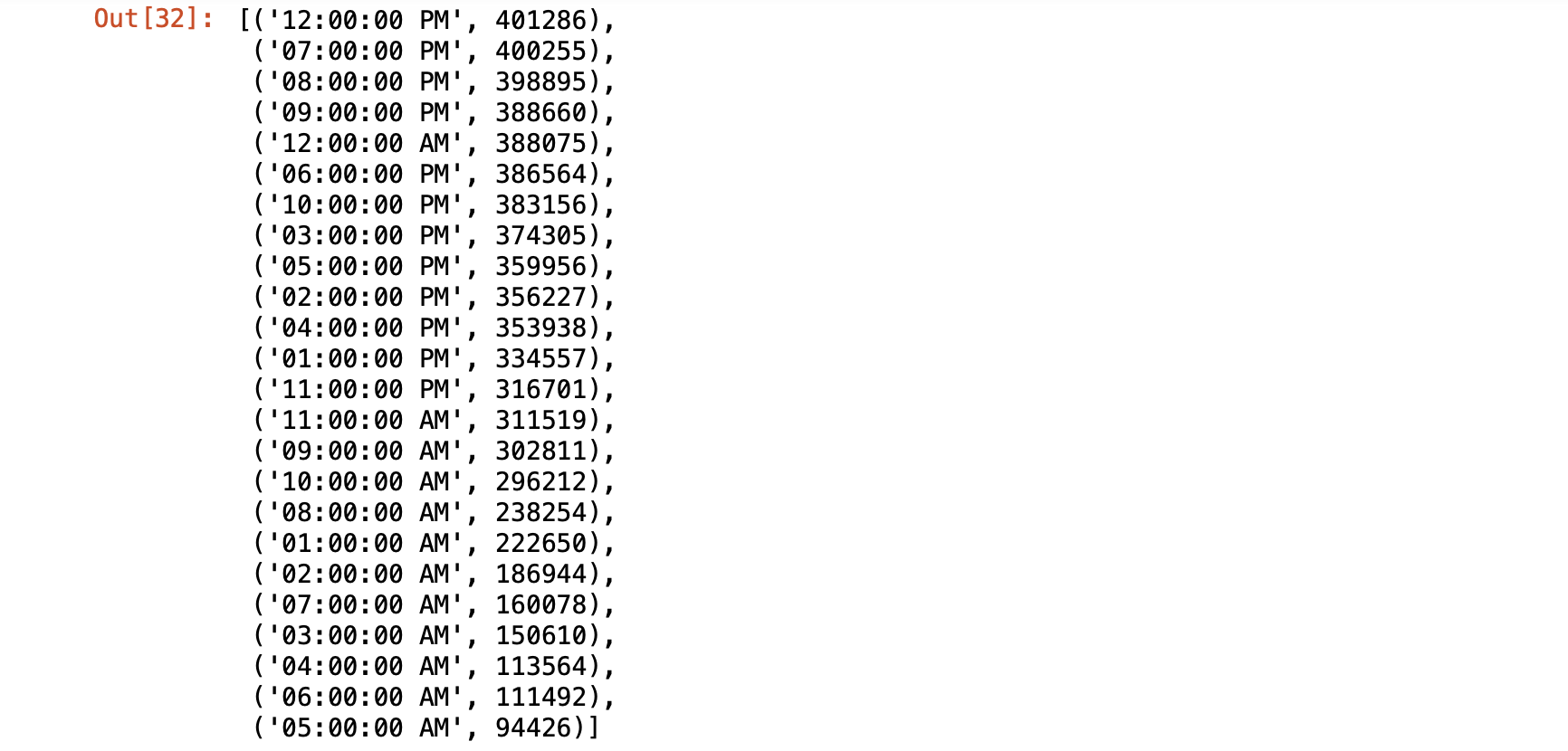




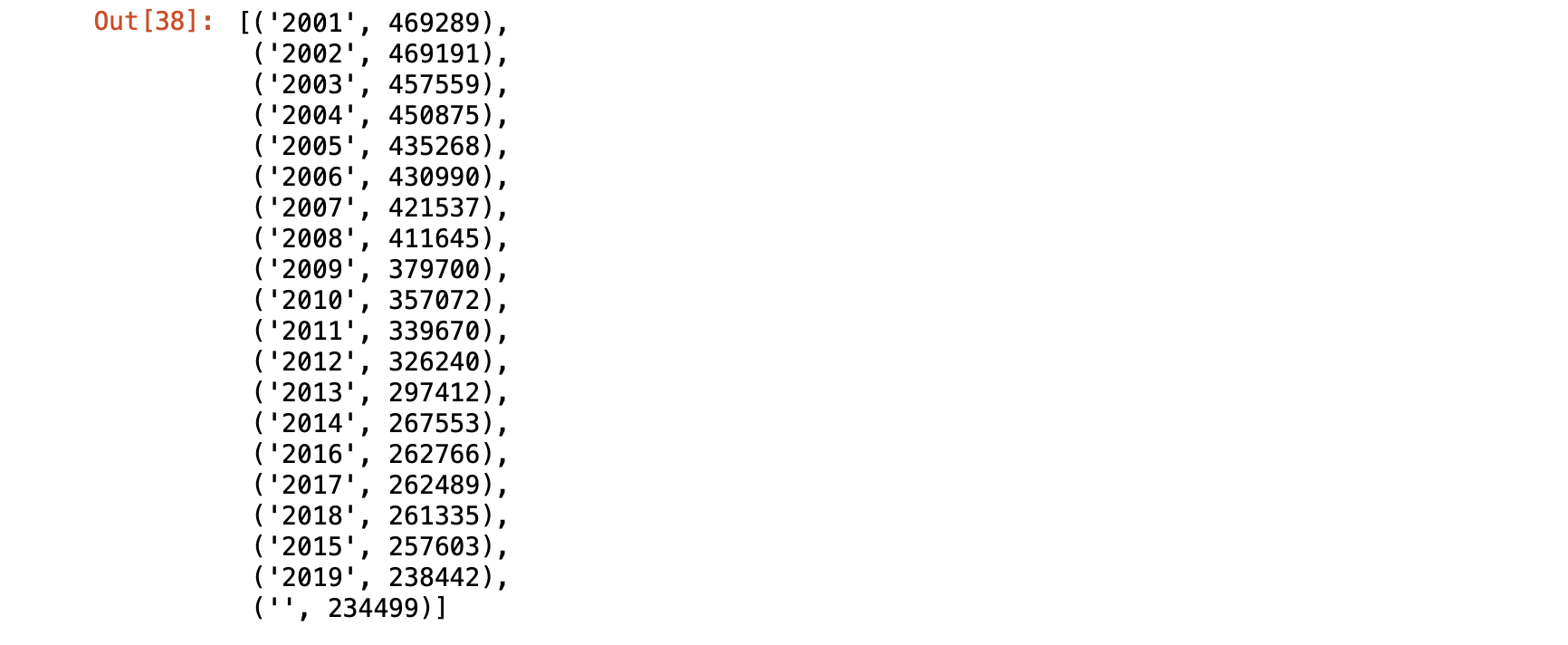




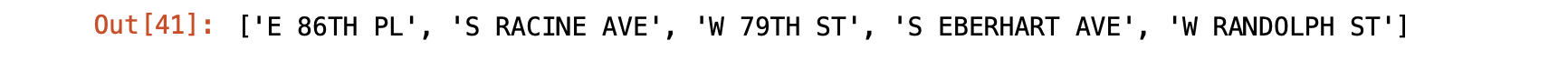


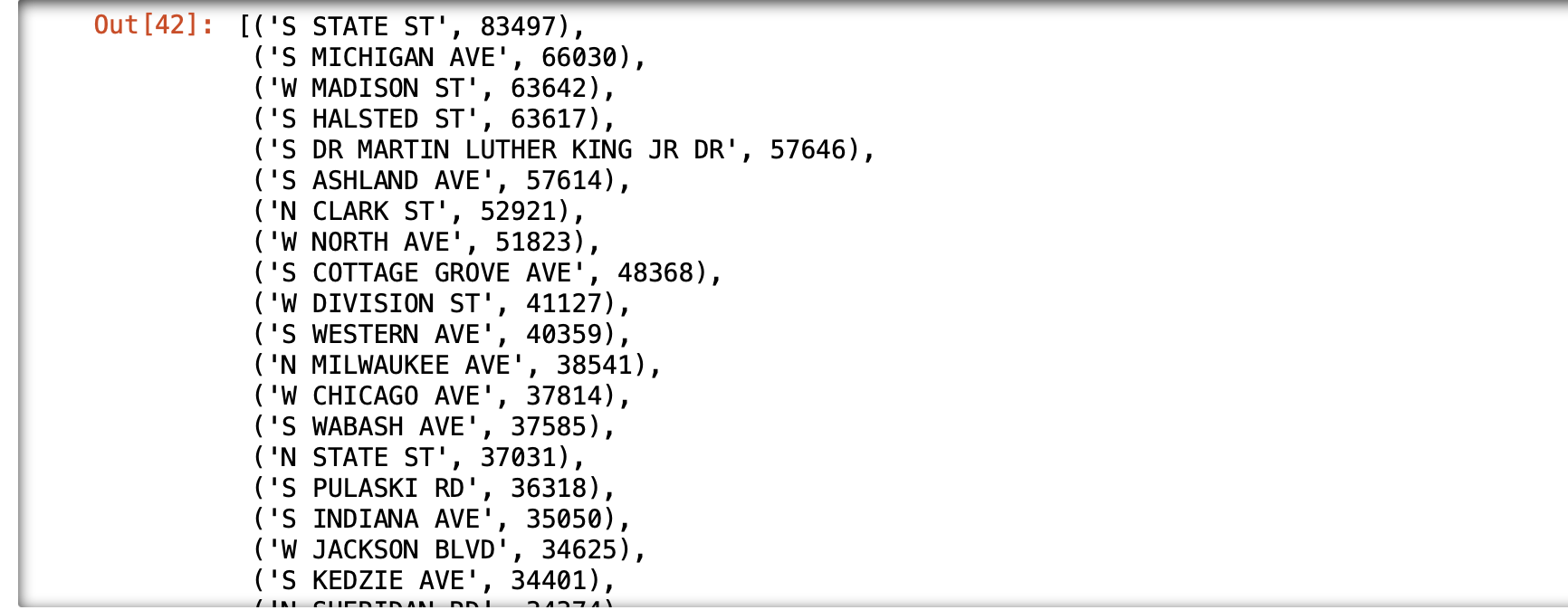


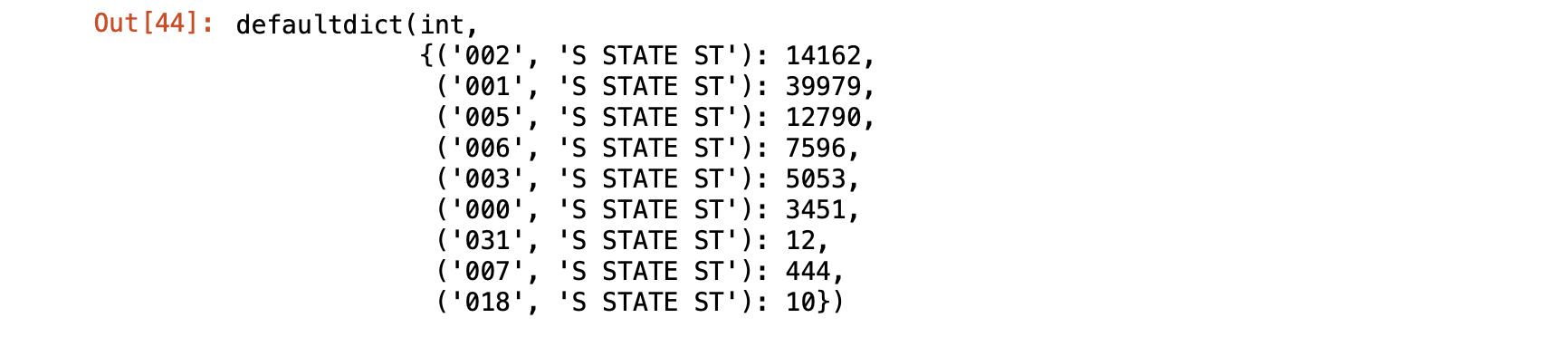












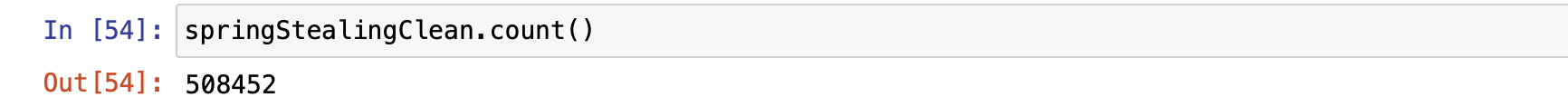


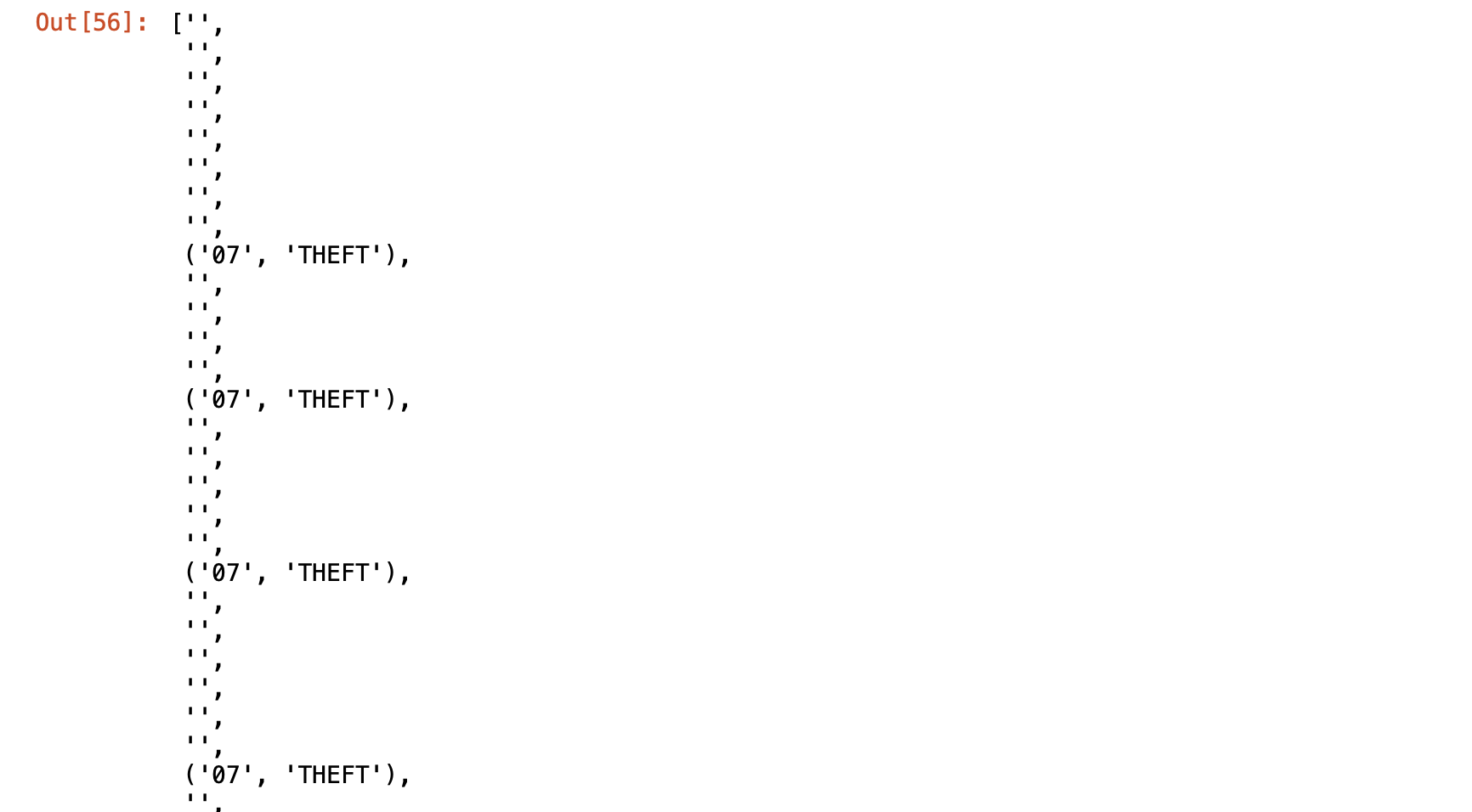






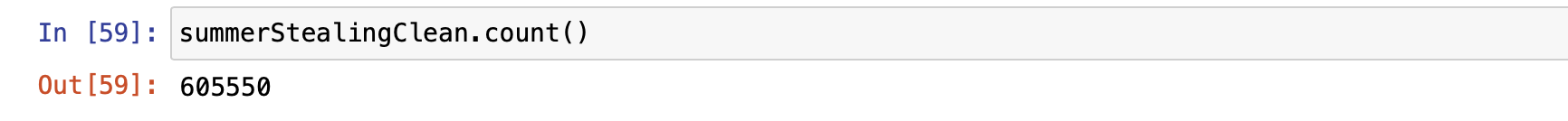


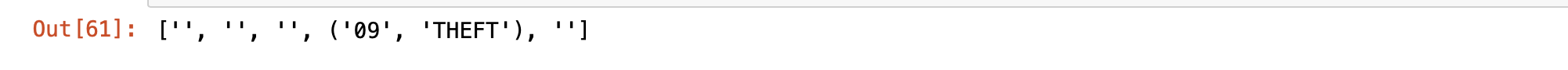


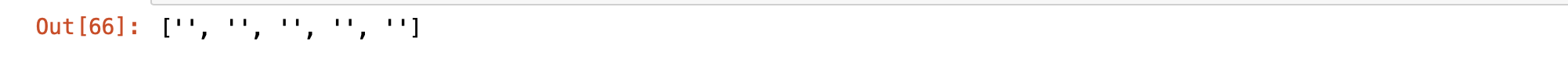






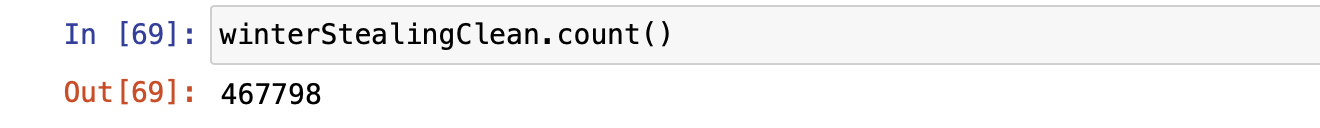


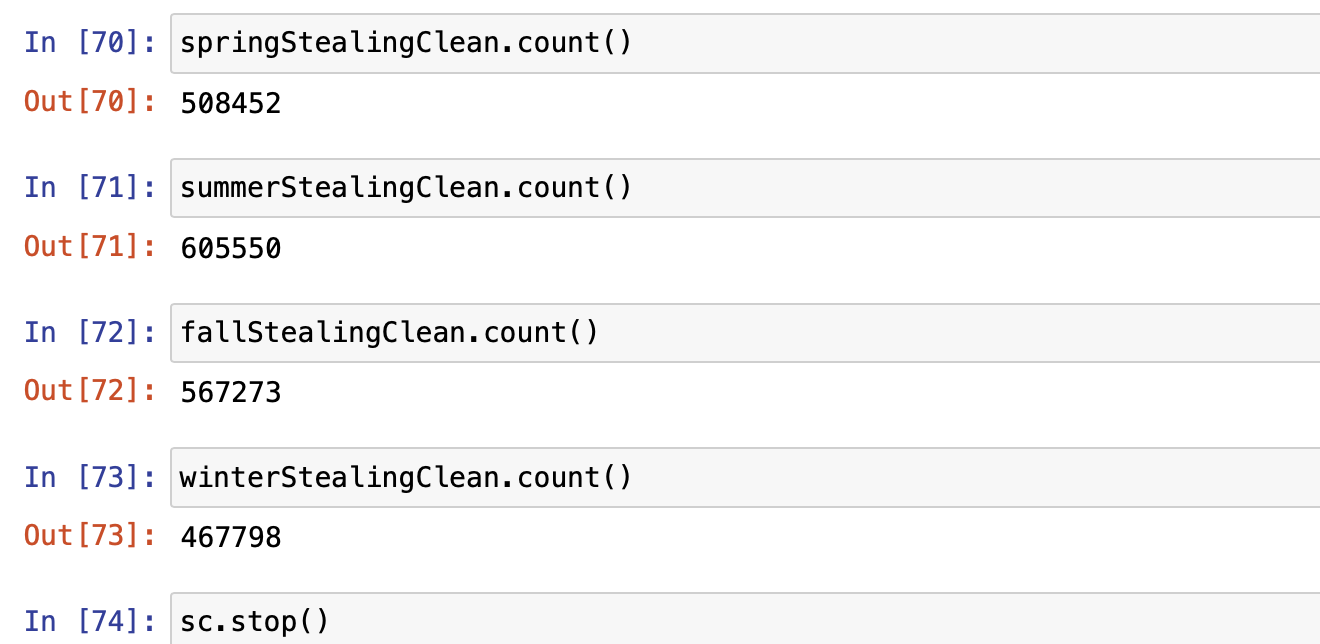












Scalability

