LONDON METROPOLITAN UNIVERSITY

islington college
(इस्लिङ्टन कलेज)

Module Code & Module Title

CC5067NI Smart Data Discovery

60% Individual Coursework

Academic Semester: Spring Semester 2025

Credit: 15 credit semester long module

Student Name: Nikhil Bhandari

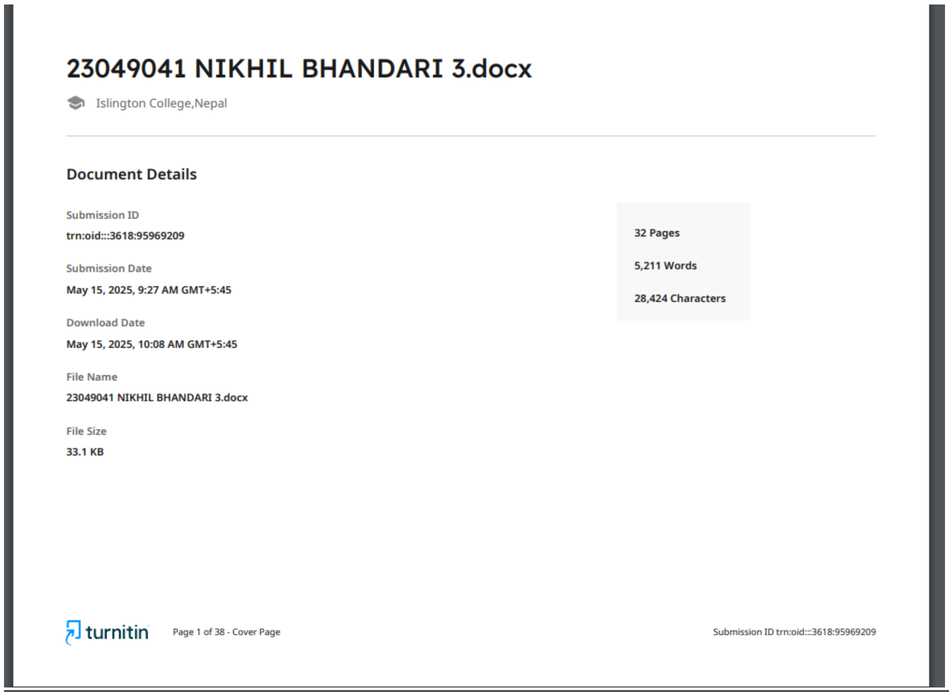
London Met ID: 23049041

College ID: NP01CP4A230338

Assignment Due Date: Thursday, May 15, 2025

Assignment Submission Date: Thursday, May 15, 2025

Submitted To: Mr. Dipeshor Silwal



23049041 NIKHIL BHANDARI 3.docx

Islington College,Nepal

Document Details

Submission ID
trn:oid:::3618:95969209

Submission Date
May 15, 2025, 9:27 AM GMT+5:45

Download Date
May 15, 2025, 10:08 AM GMT+5:45

File Name
23049041 NIKHIL BHANDARI 3.docx

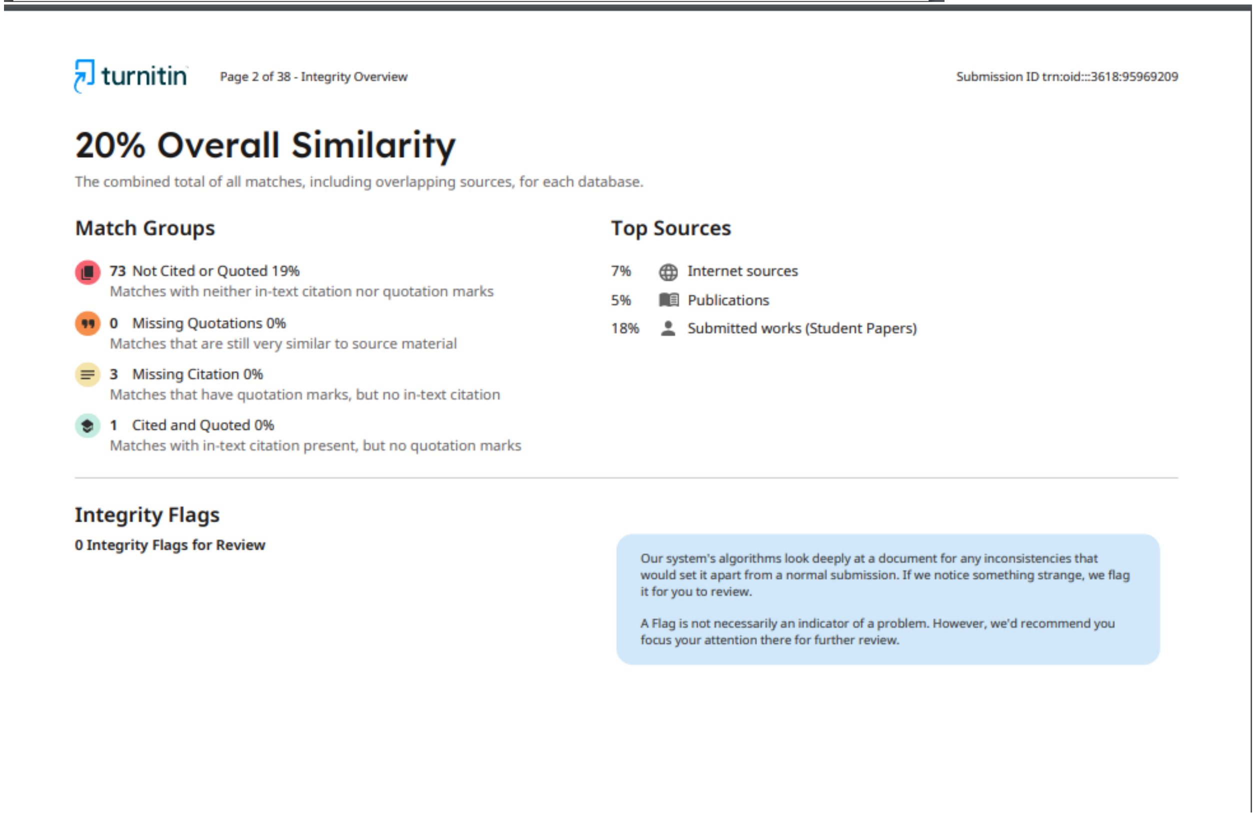
File Size
33.1 KB

32 Pages

5,211 Words

28,424 Characters

turnitin Page 1 of 38 - Cover Page Submission ID trn:oid:::3618:95969209



turnitin Page 2 of 38 - Integrity Overview Submission ID trn:oid:::3618:95969209

20% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Match Groups

73 Not Cited or Quoted 19%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

3 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

1 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

7%	Internet sources
5%	Publications
18%	Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Table of Contents

Tables Of Figures

1. Data Understanding

This dataset captures service complaints reported to the **New York City Police Department (NYPD)**, focusing on quality-of-life and public safety issues. There is 300699 data in this file. Each row represents a incident, such as noise disturbances (e.g., loud parties), blocked driveways, or illegal parking. The data includes timestamps for when the complaint was logged (*Created Date*) and resolved (*Closed Date*), enabling analysis of response times. It also includes location info like ZIP codes (*Incident Zip*), addresses, and exact coordinates (*Latitude/Longitude*), which helps map where problems happen. Some complaints have extra notes (*Descriptor*), like "Commercial Overnight Parking," while other columns (mostly empty here) seem related to bigger issues like bridges or highways. Most of the complaints in this dataset are about street or sidewalk problems, which makes sense for a busy city like NYC. With a mix of categories, dates, and locations, this data could be useful for studying city complaints.

There is a column information tabled below:

S.No	Column Name	Description	Data Type
0	Unique Key	A unique identifier for each complaint /service request.	Int64
1	Created Date	The date and time the complaint was created.	datetime64[ns]
2	Closed Date	The date and time the complaint was closed.	datetime64[ns]
3	Agency	The city agency responsible for handling the request.	object
4	Agency Name	The full name of the agency handling the request.	object
5	Complaint Type	The general category of the issue reported (e.g., Noise, Illegal Parking).	object
6	Descriptor	A more detailed description of the complaint type.	object
7	Location Type	The type of location where the issue occurred (e.g., Residential Building, Street/Sidewalk).	object

8	Incident Zip	The ZIP code where the complaint occurred.	float64
9	Incident Address	The street address where the issue happened.	object
10	Street Name	The name of the street where the complaint occurred.	object
11	Cross Street 1	The first intersecting street near the incident.	object
12	Cross Street 2	The second intersecting street near the incident.	object
13	Intersection Street 1	One street in the intersection where the issue occurred (used when no specific address is given).	object
14	Intersection Street 2	The second street in the intersection.	object
15	Address Type	Indicates if the address is a specific house, intersection, or blockface.	object
16	City	The city where the incident occurred.	object
17	Landmark	Noted landmark near the complaint location (if any).	object
18	Facility Type	The type of facility involved in the complaint (e.g., Public School).	object
19	Status	The current status of the complaint (e.g., Open, Closed).	object
20	Due Date	The date by which the agency is expected to resolve the issue.	object
21	Resolution Description	Description of the resolution action taken by the agency.	object
22	Resolution Action Updated Date	The last time the resolution action was updated.	object
23	Community Board	The community board responsible for the area where the issue occurred.	object
24	Borough	The borough where the complaint was filed (e.g., Brooklyn, Manhattan).	object
25	X Coordinate (State Plane)	X coordinate of the incident location using NYC’s State Plane coordinate system.	float64

26	Y Coordinate (State Plane)	Y coordinate of the incident location using NYC's State Plane coordinate system.	float64
27	Park Facility Name	Name of the park facility (if applicable).	object
28	Park Borough	The borough where the park is located.	object
29	School Name	The name of the school associated with the complaint or incident.	object
30	School Number	A unique number identifying the school in the city's school system.	object
31	School Region	The administrative region the school belongs to.	object
32	School Code	A specific code used to identify the school within the education system.	object
33	School Phone Number	The main contact phone number for the school.	object
34	School Address	The street address where the school is located.	object
35	School City	The city in which the school is located.	object
36	School State	The U.S. state (usually NY) where the school is located.	object
37	School Zip	The ZIP code for the school's address.	object
38	School Not Found	Indicates whether the system was unable to match the school information (e.g., Yes/No).	object
39	School or Citywide Complaint	States whether the complaint pertains to a specific school or is a broader, citywide education issue.	float64
40	Vehicle Type	The type of vehicle involved in the complaint (e.g., Car, Truck, Bus), if applicable.	float64
41	Taxi Company Borough	The borough where the taxi company involved in the complaint is based.	float64
42	Taxi Pick Up Location	The location where the taxi picked up a passenger, relevant for taxi-related complaints.	float64

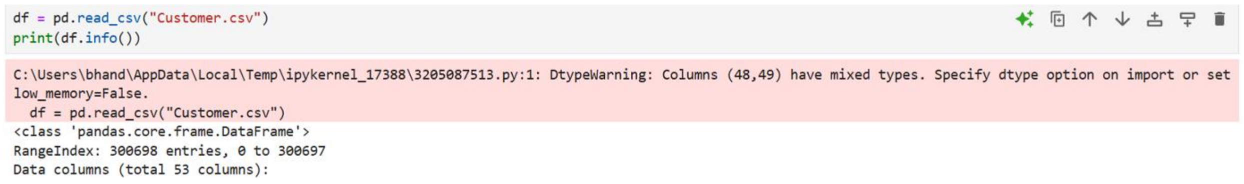
43	Bridge Highway Name	The name of the bridge or highway referenced in the complaint.	object
44	Bridge Highway Direction	The direction of traffic flow on the bridge or highway (e.g., Northbound, Eastbound).	object
45	Road Ramp	The name or designation of a specific road ramp involved in the incident.	object
46	Bridge Highway Segment	The specific segment of the bridge or highway where the issue occurred.	object
47	Garage Lot Name	The name of the parking garage or lot associated with the complaint.	float64
48	Ferry Direction	Indicates the direction of ferry travel involved in the complaint (e.g., To Manhattan, To Staten Island).	object
49	Ferry Terminal Name	The name of the ferry terminal related to the incident or complaint.	object
50	Latitude	The geographic latitude coordinate of the incident location.	float64
51	Longitude	The geographic longitude coordinate of the incident location.	float64
52	Location	A combined geographic point field showing the latitude and longitude of the incident.	object

2 Data Preparation

2.1 Import the dataset

I imported the dataset **Customer.csv** (I rename the file name from Customer Service_Requests_from_2010_to_Present.csv) using the pandas library, which is one of the most widely used Python libraries for data manipulation and analysis. pandas provides powerful data structures like Series and DataFrame that make it easy to load, explore, clean, and analyze structured data. I used the read_csv() function to load the CSV file into a DataFrame named df, which allows for accessible handling of tabular data.

When I initially ran the code first, I encountered the following **warning message**:



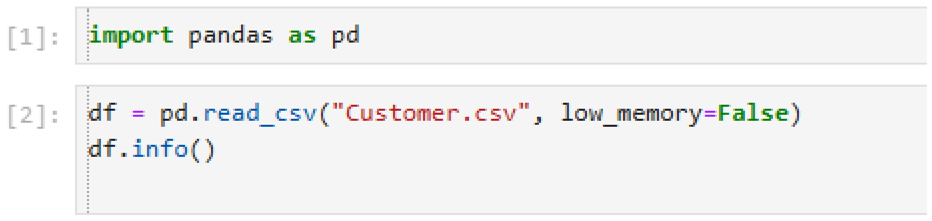
```
df = pd.read_csv("Customer.csv")
print(df.info())
```

```
C:\Users\bhand\AppData\Local\Temp\ipykernel_17388\3205087513.py:1: DtypeWarning: Columns (48,49) have mixed types. Specify dtype option on import or set
low_memory=False.
  df = pd.read_csv("Customer.csv")
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300698 entries, 0 to 300697
Data columns (total 53 columns):
```

Figure 1: Warning messaging while trying to import csv file

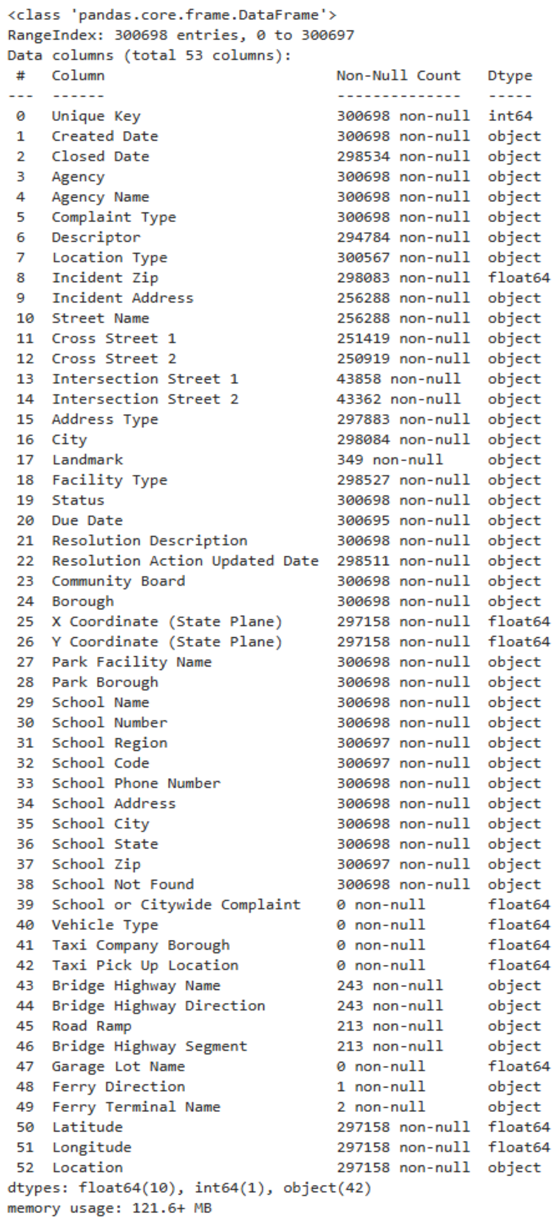
This warning indicated that columns 48 and 49 contained mixed data types (such as both numbers and strings), which caused pandas to be unsure about how to interpret the data types when reading the file in chunks.

To fix this, I updated the read_csv() function by adding the parameter low_memory =False. This tells pandas to read the entire file into memory before determining data types, which resolved the issue and allowed the dataset to be imported correctly. After successfully loading the data, I used the df.info() method to inspect the structure of the DataFrame. This provided a brief summary including the number of rows, column names, data types, and non-null value counts. This step was essential for understanding the initial condition of the dataset and for planning the necessary data cleaning and preprocessing steps.



```
[1]: import pandas as pd

[2]: df = pd.read_csv("Customer.csv", low_memory=False)
     df.info()
```

Figure 2:Import csv file code



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300698 entries, 0 to 300697
Data columns (total 53 columns):
 #   Column                          Non-Null Count   Dtype  
---  ------                          --------------   -----  
 0   Unique Key                      300698 non-null  int64  
 1   Created Date                    300698 non-null  object 
 2   Closed Date                     298534 non-null  object 
 3   Agency                          300698 non-null  object 
 4   Agency Name                     300698 non-null  object 
 5   Complaint Type                  300698 non-null  object 
 6   Descriptor                      294784 non-null  object 
 7   Location Type                   300567 non-null  object 
 8   Incident Zip                    298083 non-null  float64
 9   Incident Address                256288 non-null  object 
 10  Street Name                     256288 non-null  object 
 11  Cross Street 1                  251419 non-null  object 
 12  Cross Street 2                  250919 non-null  object 
 13  Intersection Street 1           43858 non-null   object 
 14  Intersection Street 2           43362 non-null   object 
 15  Address Type                    297883 non-null  object 
 16  City                            298084 non-null  object 
 17  Landmark                        349 non-null     object 
 18  Facility Type                   298527 non-null  object 
 19  Status                          300698 non-null  object 
 20  Due Date                        300695 non-null  object 
 21  Resolution Description          300698 non-null  object 
 22  Resolution Action Updated Date  298511 non-null  object 
 23  Community Board                 300698 non-null  object 
 24  Borough                         300698 non-null  object 
 25  X Coordinate (State Plane)      297158 non-null  float64
 26  Y Coordinate (State Plane)      297158 non-null  float64
 27  Park Facility Name              300698 non-null  object 
 28  Park Borough                    300698 non-null  object 
 29  School Name                     300698 non-null  object 
 30  School Number                   300698 non-null  object 
 31  School Region                   300697 non-null  object 
 32  School Code                     300697 non-null  object 
 33  School Phone Number             300698 non-null  object 
 34  School Address                  300698 non-null  object 
 35  School City                     300698 non-null  object 
 36  School State                    300698 non-null  object 
 37  School Zip                      300697 non-null  object 
 38  School Not Found                300698 non-null  object 
 39  School or Citywide Complaint    0 non-null       float64
 40  Vehicle Type                    0 non-null       float64
 41  Taxi Company Borough            0 non-null       float64
 42  Taxi Pick Up Location           0 non-null       float64
 43  Bridge Highway Name             243 non-null     object 
 44  Bridge Highway Direction        243 non-null     object 
 45  Road Ramp                       213 non-null     object 
 46  Bridge Highway Segment          213 non-null     object 
 47  Garage Lot Name                 0 non-null       float64
 48  Ferry Direction                 1 non-null       object 
 49  Ferry Terminal Name             2 non-null       object 
 50  Latitude                        297158 non-null  float64
 51  Longitude                       297158 non-null  float64
 52  Location                        297158 non-null  object 
dtypes: float64(10), int64(1), object(42)
memory usage: 121.6+ MB
```

Figure 3: Import csv Output

2.2 Provide your insight on the information and details that the provided dataset carries.

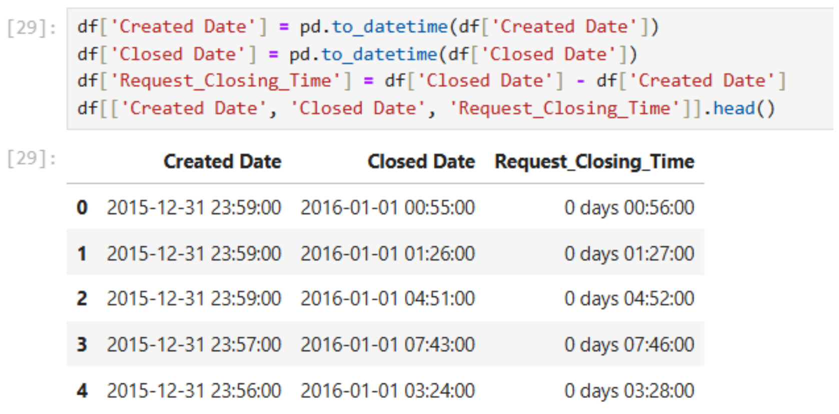
The dataset contains customer service requests submitted in New York City from 2010 to the present. It includes **300,698 entries across 53 columns**, capturing a wide range of information related to public complaints and service needs. Each record represents a complaint made by residents and includes information such as complaint type, responsible agency, timestamps (created and closed dates), and location details (address, ZIP code, latitude, and longitude).

The most frequent complaint types include noise issues, blocked driveways, and illegal parking. Temporal data allows for trend and response time analysis, while geographic data supports hotspot and borough-level analysis. Although key fields are well-populated, some infrastructure-related columns (e.g., bridge and ferry data) have missing values. This dataset is really useful for studying how well the city handles complaints, what issues people report most, and where problems keep happening.

2.3 Convert the columns "Created Date" and "Closed Date" to datetime datatype and create a new column "Request_Closing_Time" as the time elapsed between request creation and request closing.

To analyze the time efficiency of service request resolutions, I processed the "Created Date" and "Closed Date" columns by converting them to datetime format using the pd.to_datetime() function from the pandas library. This conversion is essential because these columns were initially in string format, which would not allow for accurate time calculations. After successfully converting both columns to datetime objects, I created a new column called "Request_Closing_Time" by subtracting "Created Date" from "Closed Date". This computed the time duration taken to close each request, resulting in a timedelta object representing the time difference.

The resulting "Request_Closing_Time" column provides a precise measure of how long it took for each complaint to be resolved. For instance, in the first few records, the closing time ranges from **55 minutes to over 7 hours**, depending on the nature of the request. This newly created column is essential for performing further statistical analysis and visualizations, such as calculating the average response time, identifying delays, and comparing resolution speeds across complaint types and locations.



```
[29]: df['Created Date'] = pd.to_datetime(df['Created Date'])
      df['Closed Date'] = pd.to_datetime(df['Closed Date'])
      df['Request_Closing_Time'] = df['Closed Date'] - df['Created Date']
      df[['Created Date', 'Closed Date', 'Request_Closing_Time']].head()
```

[29]:

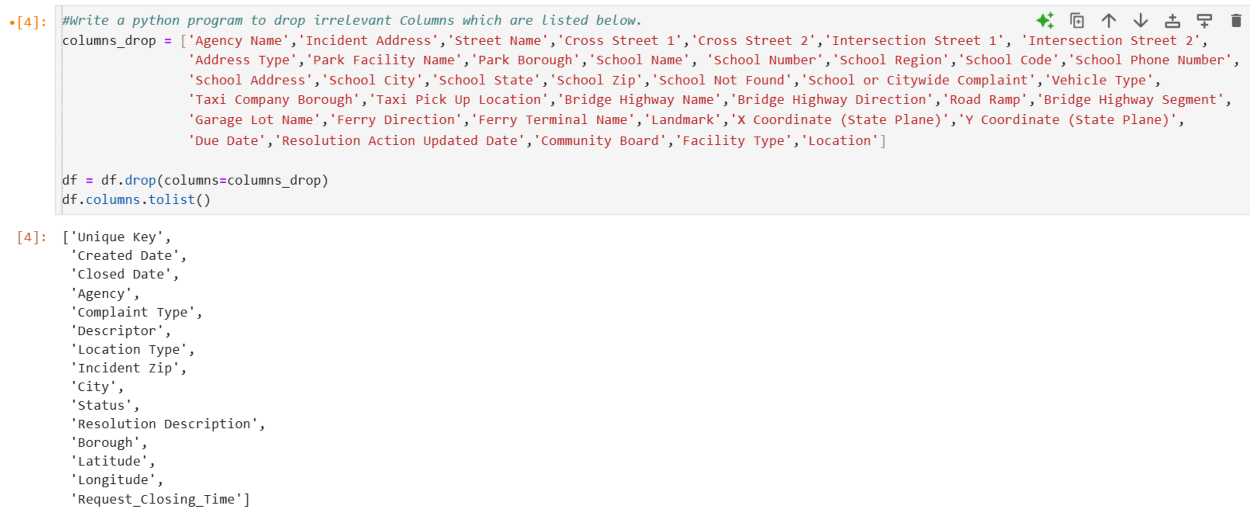
	Created Date	Closed Date	Request_Closing_Time
0	2015-12-31 23:59:00	2016-01-01 00:55:00	0 days 00:56:00
1	2015-12-31 23:59:00	2016-01-01 01:26:00	0 days 01:27:00
2	2015-12-31 23:59:00	2016-01-01 04:51:00	0 days 04:52:00
3	2015-12-31 23:57:00	2016-01-01 07:43:00	0 days 07:46:00
4	2015-12-31 23:56:00	2016-01-01 03:24:00	0 days 03:28:00

Figure 4: Created Request_Closing_Time

2.4 Write a python program to drop irrelevant Columns which are listed below.

I removed several unnecessary or less meaningful columns using the drop() function from the pandas library to make the dataset easier to work with and focus my analysis on the most relevant information. These included detailed location fields like "Incident Address" and "Cross Street 1", school-related data such as "School Name" and "School Phone Number", transportation-specific fields like "Taxi Pick Up Location" and "Bridge Highway Name", and system-generated metadata such as "Community Board" and "Resolution Action Updated Date". Most of these columns either had a lot of missing or inconsistent values, weren’t useful for answering the main research questions, or added extra complexity without much benefit. So, I created a list called columns_drop and used df.drop(columns=columns_drop) to remove them all at once.

This reduced the dataset from 53 columns down to 15, keeping only the most important variables. The remaining columns included identifiers (like "Unique Key"), date and time fields ("Created Date", "Closed Date"), key categorical variables ("Agency", "Complaint Type", "Location Type"), location details ("City", "Incident Zip", "Borough", "Latitude", "Longitude"), and the calculated "Request_Closing_Time" column. This step helped the dataset it easier to interpret, and allowed me to focus on variables that directly support the analysis goals. It also helped reduce noise in the data, which improves the overall efficiency and clarity of the next stages of the project.



```
•[4]: #Write a python program to drop irrelevant Columns which are listed below.
      columns_drop = ['Agency Name','Incident Address','Street Name','Cross Street 1','Cross Street 2','Intersection Street 1', 'Intersection Street 2',
                      'Address Type','Park Facility Name','Park Borough','School Name', 'School Number','School Region','School Code','School Phone Number',
                      'School Address','School City','School State','School Zip','School Not Found','School or Citywide Complaint','Vehicle Type',
                      'Taxi Company Borough','Taxi Pick Up Location','Bridge Highway Name','Bridge Highway Direction','Road Ramp','Bridge Highway Segment',
                      'Garage Lot Name','Ferry Direction','Ferry Terminal Name','Landmark','X Coordinate (State Plane)','Y Coordinate (State Plane)',
                      'Due Date','Resolution Action Updated Date','Community Board','Facility Type','Location']

      df = df.drop(columns=columns_drop)
      df.columns.tolist()

 [4]: ['Unique Key',
       'Created Date',
       'Closed Date',
       'Agency',
       'Complaint Type',
       'Descriptor',
       'Location Type',
       'Incident Zip',
       'City',
       'Status',
       'Resolution Description',
       'Borough',
       'Latitude',
       'Longitude',
       'Request_Closing_Time']
```

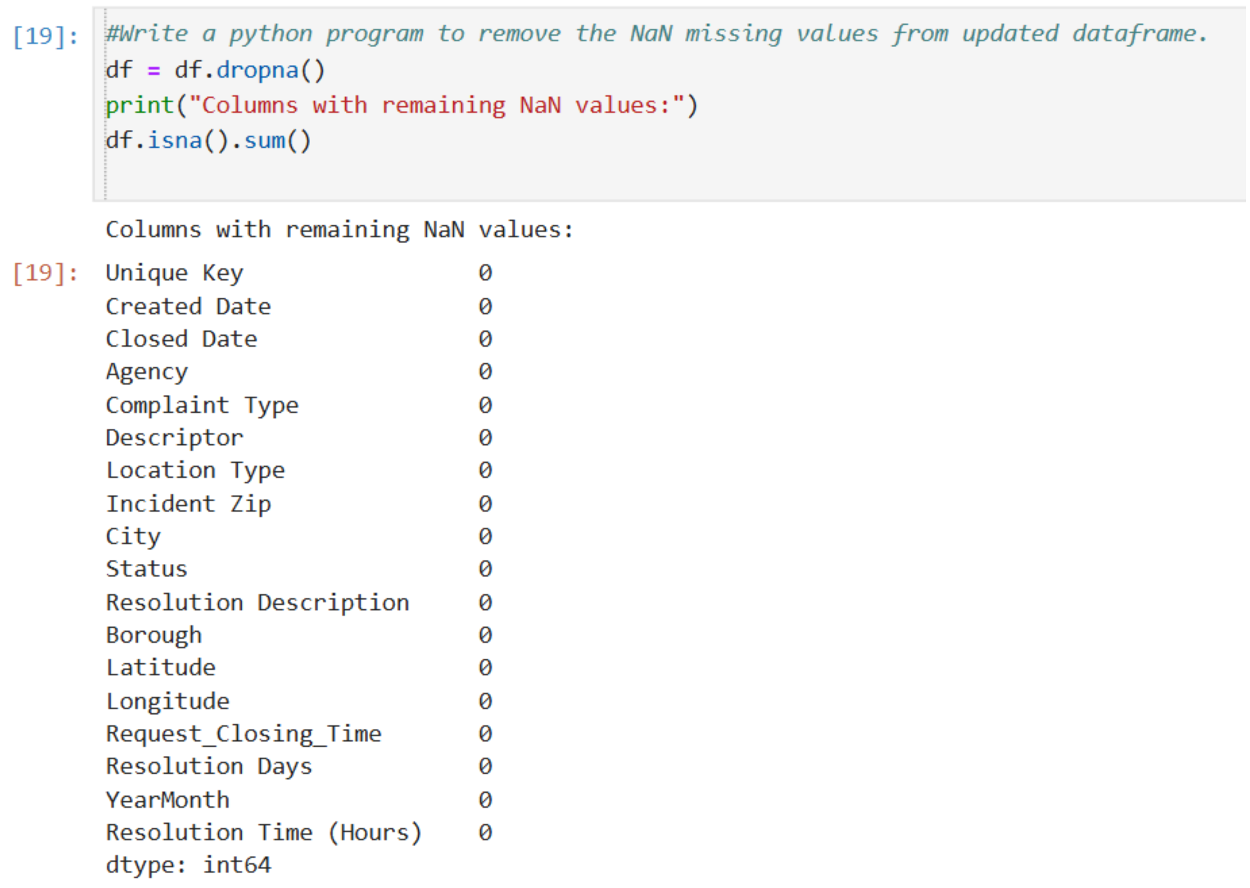
Figure 5:Drop irrelevant columns

2.5 Write a python program to remove the NaN missing values from updated dataframe.

As part of the data cleaning and preparation process, it is essential to handle missing or null values to maintain the truth and consistency of subsequent analysis. In this step, I used the dropna() function provided by the pandas library to remove any rows in the dataset that contained NaN (Not a Number) values. Missing data can result from incomplete records, data entry errors, or system limitations, and if left unaddressed, it can lead to inaccurate calculations or biased results.

The code df = df.dropna() effectively filters out all rows that contain at least one missing value across any column. This approach was chosen to ensure that the dataset remains consistent and complete for statistical analysis, especially since the columns retained after the earlier cleaning step are critical to our objectives. For example, key variables such as 'Complaint Type', 'City', 'Incident Zip', 'Resolution Description', and 'Request_Closing_Time' must be fully populated to draw reliable insights about service performance and complaint trends.

To verify the removal of missing values, I used df.isna().sum(), which confirmed that all columns now have zero missing values. The cleaned DataFrame now contains important attributes like **'Unique Key'**, **'Created Date'**, **'Complaint Type'**, and derived columns like **'Resolution Days'** and **'Resolution Time (Hours)'**. This step is very important because it makes sure the dataset is strong and reliable before doing more advanced things like analyzing data or making predictions. It helps reduce mistakes and makes the results more accurate.



```
[19]: #Write a python program to remove the NaN missing values from updated dataframe.
      df = df.dropna()
      print("Columns with remaining NaN values:")
      df.isna().sum()

      Columns with remaining NaN values:

[19]: Unique Key                 0
      Created Date               0
      Closed Date                0
      Agency                     0
      Complaint Type             0
      Descriptor                 0
      Location Type              0
      Incident Zip               0
      City                       0
      Status                     0
      Resolution Description     0
      Borough                    0
      Latitude                   0
      Longitude                  0
      Request_Closing_Time       0
      Resolution Days            0
      YearMonth                  0
      Resolution Time (Hours)    0
      dtype: int64
```

Figure 6:Remove the nan missing value

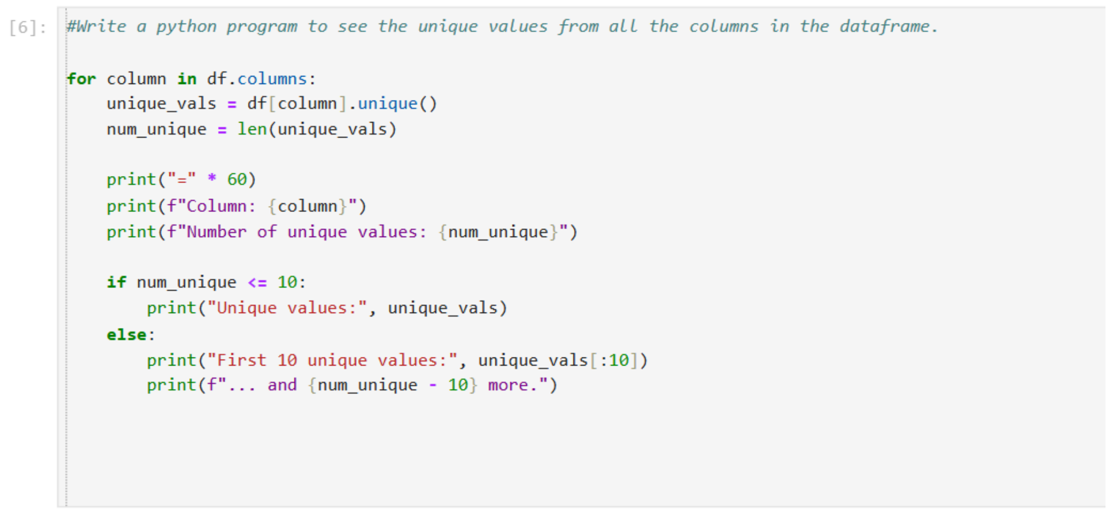
2.6 Write a python program to see the unique values from all the columns in the dataframe.

To better understand the structure and distribution of values within the dataset, the loop iterates through each column of the cleaned DataFrame and displays the number of unique values along with a sample of these values. This process is crucial for exploratory data analysis (EDA), as it helps identify categorical variables, spot differences, recognize repeated patterns.

The code uses a for loop to iterate over df.columns, applying df[column].unique() to retrieve the array of unique values in each column, and len() to count them. If a column has 10 or fewer unique values, all of them are printed for complete visibility. If it has more than 10, only the first 10 values are displayed to keep the output concise, followed by a count of how many more unique values exist beyond those shown.

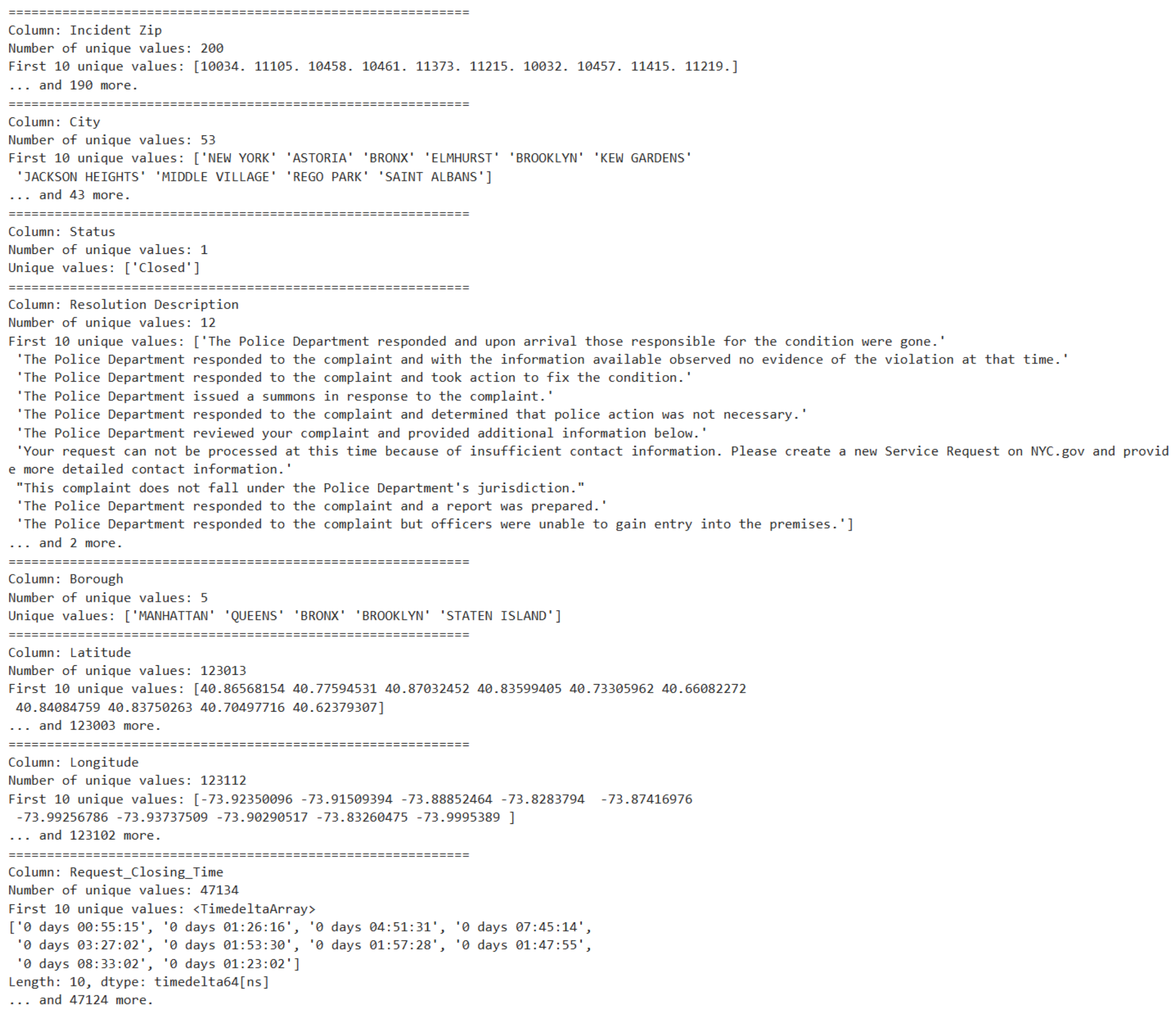
From the results, it was observed that the 'Unique Key' column has over 291,000 unique values, confirming it serves as an identifier for each record. Temporal fields like 'Created Date' and 'Closed Date' also had a high number of unique values, which is expected given the time-stamped nature of individual service requests. Interestingly, fields like 'Agency' and 'Status' contained only a single unique value ('NYPD' and 'Closed' respectively), indicating that all entries are handled by the same agency and that all complaints in this dataset have been resolved.

Categorical fields like 'Complaint Type', 'Descriptor', 'Location Type', 'City', and 'Borough' had a manageable range of unique values, making them suitable for grouping, filtering, and possibly one-hot encoding in future steps. For example, there are 15 unique complaint types and 41 descriptors, which reveal the diversity of complaints lodged across NYC. Geographic coordinates such as 'Latitude' and 'Longitude' showed over 123,000 unique entries, corresponding to the precise locations of complaints. Finally, calculated fields like 'Request_Closing_Time' also showed a wide range of values, indicating varying durations for resolving complaints.



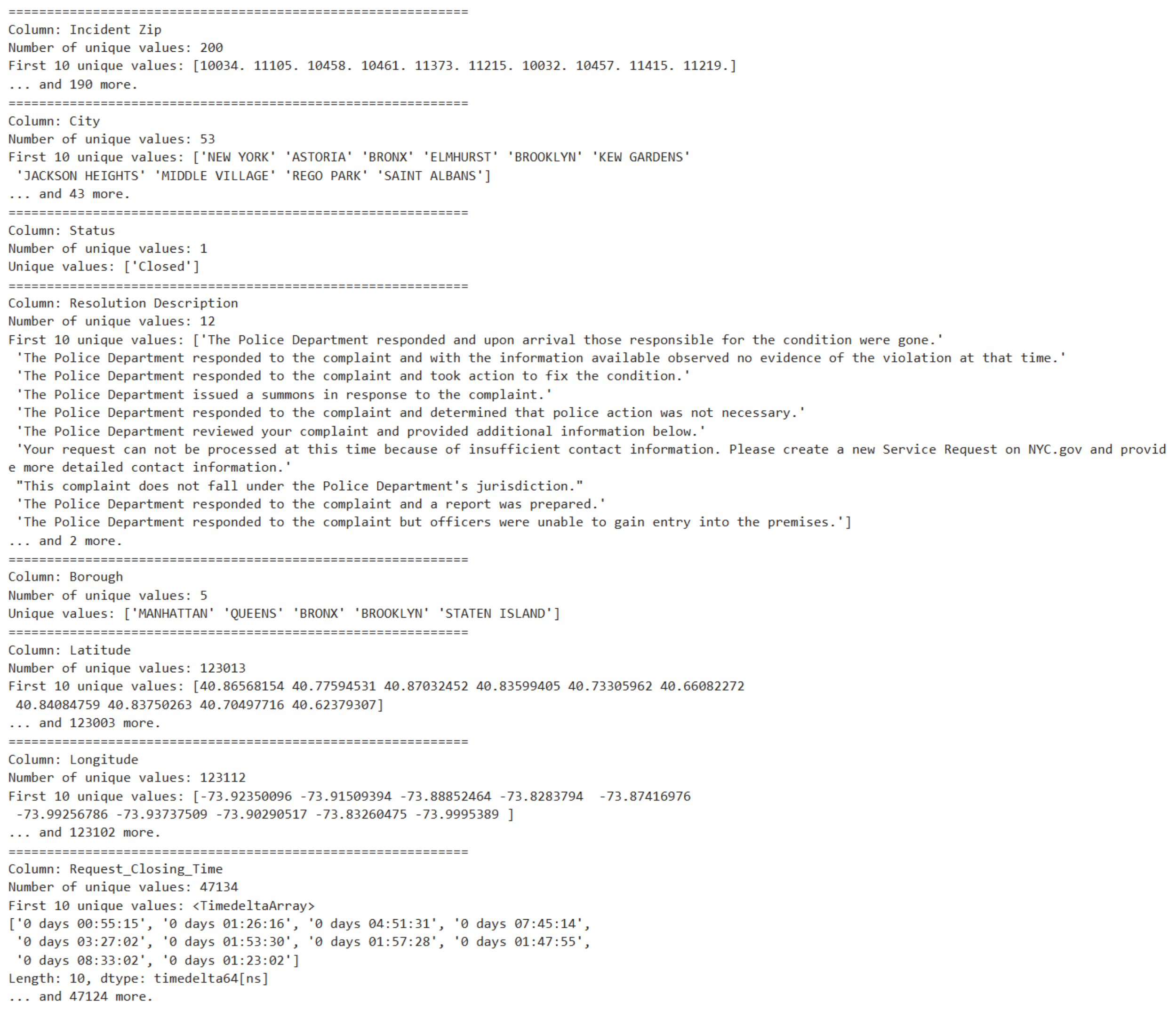
```
[6]: #Write a python program to see the unique values from all the columns in the dataframe.

for column in df.columns:
    unique_vals = df[column].unique()
    num_unique = len(unique_vals)

    print("=" * 60)
    print(f"Column: {column}")
    print(f"Number of unique values: {num_unique}")

    if num_unique <= 10:
        print("Unique values:", unique_vals)
    else:
        print("First 10 unique values:", unique_vals[:10])
        print(f"... and {num_unique - 10} more.")
```

Figure 7: Unique value



```
============================================================
Column: Incident Zip
Number of unique values: 200
First 10 unique values: [10034. 11105. 10458. 10461. 11373. 11215. 10032. 10457. 11415. 11219.]
... and 190 more.
============================================================
Column: City
Number of unique values: 53
First 10 unique values: ['NEW YORK' 'ASTORIA' 'BRONX' 'ELMHURST' 'BROOKLYN' 'KEW GARDENS'
 'JACKSON HEIGHTS' 'MIDDLE VILLAGE' 'REGO PARK' 'SAINT ALBANS']
... and 43 more.
============================================================
Column: Status
Number of unique values: 1
Unique values: ['Closed']
============================================================
Column: Resolution Description
Number of unique values: 12
First 10 unique values: ['The Police Department responded and upon arrival those responsible for the condition were gone.'
 'The Police Department responded to the complaint and with the information available observed no evidence of the violation at that time.'
 'The Police Department responded to the complaint and took action to fix the condition.'
 'The Police Department issued a summons in response to the complaint.'
 'The Police Department responded to the complaint and determined that police action was not necessary.'
 'The Police Department reviewed your complaint and provided additional information below.'
 'Your request can not be processed at this time because of insufficient contact information. Please create a new Service Request on NYC.gov and provid
e more detailed contact information.'
 "This complaint does not fall under the Police Department's jurisdiction."
 'The Police Department responded to the complaint and a report was prepared.'
 'The Police Department responded to the complaint but officers were unable to gain entry into the premises.']
... and 2 more.
============================================================
Column: Borough
Number of unique values: 5
Unique values: ['MANHATTAN' 'QUEENS' 'BRONX' 'BROOKLYN' 'STATEN ISLAND']
============================================================
Column: Latitude
Number of unique values: 123013
First 10 unique values: [40.86568154 40.77594531 40.87032452 40.83599405 40.73305962 40.66082272
 40.84084759 40.83750263 40.70497716 40.62379307]
... and 123003 more.
============================================================
Column: Longitude
Number of unique values: 123112
First 10 unique values: [-73.92350096 -73.91509394 -73.88852464 -73.8283794  -73.87416976
 -73.99256786 -73.93737509 -73.90290517 -73.83260475 -73.9995389 ]
... and 123102 more.
============================================================
Column: Request_Closing_Time
Number of unique values: 47134
First 10 unique values: <TimedeltaArray>
['0 days 00:55:15', '0 days 01:26:16', '0 days 04:51:31', '0 days 07:45:14',
 '0 days 03:27:02', '0 days 01:53:30', '0 days 01:57:28', '0 days 01:47:55',
 '0 days 08:33:02', '0 days 01:23:02']
Length: 10, dtype: timedelta64[ns]
... and 47124 more.
```

Figure 8: Result of unique value (1)



```
============================================================
Column: Incident Zip
Number of unique values: 200
First 10 unique values: [10034. 11105. 10458. 10461. 11373. 11215. 10032. 10457. 11415. 11219.]
... and 190 more.
============================================================
Column: City
Number of unique values: 53
First 10 unique values: ['NEW YORK' 'ASTORIA' 'BRONX' 'ELMHURST' 'BROOKLYN' 'KEW GARDENS'
 'JACKSON HEIGHTS' 'MIDDLE VILLAGE' 'REGO PARK' 'SAINT ALBANS']
... and 43 more.
============================================================
Column: Status
Number of unique values: 1
Unique values: ['Closed']
============================================================
Column: Resolution Description
Number of unique values: 12
First 10 unique values: ['The Police Department responded and upon arrival those responsible for the condition were gone.'
 'The Police Department responded to the complaint and with the information available observed no evidence of the violation at that time.'
 'The Police Department responded to the complaint and took action to fix the condition.'
 'The Police Department issued a summons in response to the complaint.'
 'The Police Department responded to the complaint and determined that police action was not necessary.'
 'The Police Department reviewed your complaint and provided additional information below.'
 'Your request can not be processed at this time because of insufficient contact information. Please create a new Service Request on NYC.gov and provid
e more detailed contact information.'
 "This complaint does not fall under the Police Department's jurisdiction."
 'The Police Department responded to the complaint and a report was prepared.'
 'The Police Department responded to the complaint but officers were unable to gain entry into the premises.']
... and 2 more.
============================================================
Column: Borough
Number of unique values: 5
Unique values: ['MANHATTAN' 'QUEENS' 'BRONX' 'BROOKLYN' 'STATEN ISLAND']
============================================================
Column: Latitude
Number of unique values: 123013
First 10 unique values: [40.86568154 40.77594531 40.87032452 40.83599405 40.73305962 40.66082272
 40.84084759 40.83750263 40.70497716 40.62379307]
... and 123003 more.
============================================================
Column: Longitude
Number of unique values: 123112
First 10 unique values: [-73.92350096 -73.91509394 -73.88852464 -73.8283794  -73.87416976
 -73.99256786 -73.93737509 -73.90290517 -73.83260475 -73.9995389 ]
... and 123102 more.
============================================================
Column: Request_Closing_Time
Number of unique values: 47134
First 10 unique values: <TimedeltaArray>
['0 days 00:55:15', '0 days 01:26:16', '0 days 04:51:31', '0 days 07:45:14',
 '0 days 03:27:02', '0 days 01:53:30', '0 days 01:57:28', '0 days 01:47:55',
 '0 days 08:33:02', '0 days 01:23:02']
Length: 10, dtype: timedelta64[ns]
... and 47124 more.
```

Figure 9:Result of unique value (2)

3. Data Exploration

3.1 Write a Python program to show summary statistics of sum, mean, standard deviation, skewness, and kurtosis of the data frame.

To gain a deeper understanding of the numerical data in the dataset, a Python program was written to generate key **summary statistics** for each numeric column. This includes calculating the **sum**, **mean (average)**, **standard deviation**, **skewness**, and **kurtosis** all of which help describe the shape and distribution of the data.

The program first selects only the numeric columns using select_dtypes, focusing on columns with integer and float values. It then calculates the following:

- **Sum**: The total value of all entries in a column. While this is more useful for meaningful quantitative values (like sales or counts), it still gives an idea of the data’s magnitude.
- **Mean**: The average value, which indicates the central tendency. For example, the average zip code (Incident Zip) helps us understand where most service requests come from.
- **Standard Deviation**: This measures how spread out the data is around the mean. A low standard deviation (like for Latitude and Longitude) means most incidents are geographically close to each other—typical for a city-based dataset.
- **Skewness**: This tells us about the **asymmetry** of the distribution:
  - A skewness close to 0 means the data is fairly symmetrical.
  - Negative skewness (like in Incident Zip) means the values are mostly high, but there are some very low zip codes pulling the distribution to the left.
  - Positive skewness means the opposite—values are mostly low with a few very high outliers.
- **Kurtosis**: This describes how "peaked" or "flat" the distribution is:

- A **high kurtosis** (as seen in Incident Zip) means the data has heavy tails—many extreme values or outliers.
- A **low kurtosis** (as in Unique Key) indicates a flatter distribution with fewer outliers.

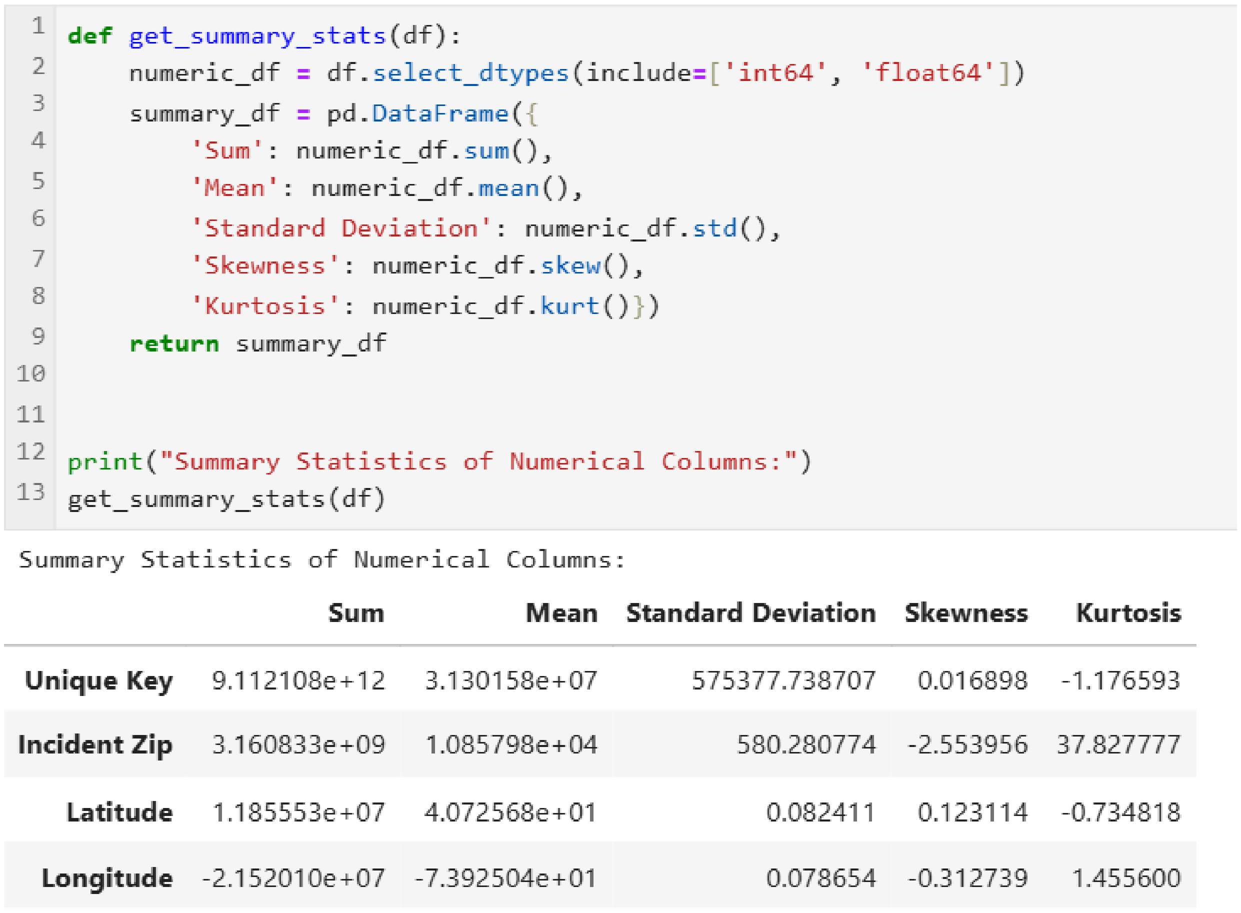
The summary statistics of the numerical columns provide valuable insights into the structure and distribution of the data.

The **'Unique Key'** column, which acts as an identifier for each complaint, shows a very large total sum and a high average value, but its low skewness (0.0169) and negative kurtosis (-1.1766) indicate that the values are fairly symmetrically distributed and relatively flat compared to a normal distribution.

The **'Incident Zip'** column reveals a mean of approximately 10,858 with a standard deviation of 580.28, suggesting that most incidents occur within a concentrated range of zip codes. However, its strong negative skewness (-2.5540) and extremely high kurtosis (37.8278) suggest that the distribution is heavily left-skewed and has a significant number of outliers, meaning a large number of incidents occur in only a few zip codes.

he **'Latitude'** and **'Longitude'** columns, representing the geographic location of complaints, have means of around 40.73 and -73.93, respectively, which aligns with New York City's coordinates. Both columns have low standard deviations, showing that the incidents are geographically concentrated. Their near-zero skewness and modest kurtosis values imply that the geographical distribution is mostly symmetrical with few extreme values. Overall, the results highlight that while identifiers and geographic data are well-distributed, the zip code distribution is more uneven, indicating potential hotspots for complaints that may require further investigation.

These statistics are foundational for understanding the distribution and quality of the data before performing further steps such as visualization, machine learning modeling, or business decision-making. They help identify whether the data has outliers, bias, or irregular distributions all of which are critical to detect early in the data analysis process.



```python
def get_summary_stats(df):
    numeric_df = df.select_dtypes(include=['int64', 'float64'])
    summary_df = pd.DataFrame({
        'Sum': numeric_df.sum(),
        'Mean': numeric_df.mean(),
        'Standard Deviation': numeric_df.std(),
        'Skewness': numeric_df.skew(),
        'Kurtosis': numeric_df.kurt()})
    return summary_df


print("Summary Statistics of Numerical Columns:")
get_summary_stats(df)
```

```
Summary Statistics of Numerical Columns:
```

	Sum	Mean	Standard Deviation	Skewness	Kurtosis
Unique Key	9.112108e+12	3.130158e+07	575377.738707	0.016898	-1.176593
Incident Zip	3.160833e+09	1.085798e+04	580.280774	-2.553956	37.827777
Latitude	1.185553e+07	4.072568e+01	0.082411	0.123114	-0.734818
Longitude	-2.152010e+07	-7.392504e+01	0.078654	-0.312739	1.455600

Figure 10: Summary Statistics of Numerical columns

3.2 Write a Python program to calculate and show correlation of all variables.

To check how strongly the numerical columns in the dataset are related to each other by calculating their correlation coefficients. This helps in spotting any patterns or connections between variables, like whether an increase in one value tends to lead to an increase or decrease in another.

Understanding these relationships is useful because it can highlight important features, reduce redundancy, and guide decisions during model building. For example, if two variables are highly correlated, we might only need one of them in our analysis to keep things simple and avoid confusion later.

To calculate correlations, the code first filters out only the **numeric columns** from the DataFrame using select_dtypes(include=['int64', 'float64']), since correlation only applies to numerical data. Then, it uses the .corr() method to compute the **Pearson correlation coefficients** between all pairs of these numeric columns.The result is a **correlation matrix**, stored in a variable named correlation_matrix, which shows how strongly each numerical column is related to the others. This matrix was then printed to get a better understanding of the relationships between different variables in the dataset. We also made a heatmap. This heatmap provides a clear visual summary of how Uniquekey, Incident zip, Latitude, and Longitude relate to each other. We can quickly see which pairs of variables have strong, weak, positive, or negative linear associations. This can be very useful for understanding the underlying structure of your data and for informing further analysis or modeling.

A **correlation matrix** shows the statistical relationship (correlation coefficient) between pairs of numeric variables. The correlation coefficient ranges from -1 to 1, where:

- **1** indicates a perfect positive correlation (as one variable increases, so does the other),
- **-1** indicates a perfect negative correlation (as one increases, the other decreases),

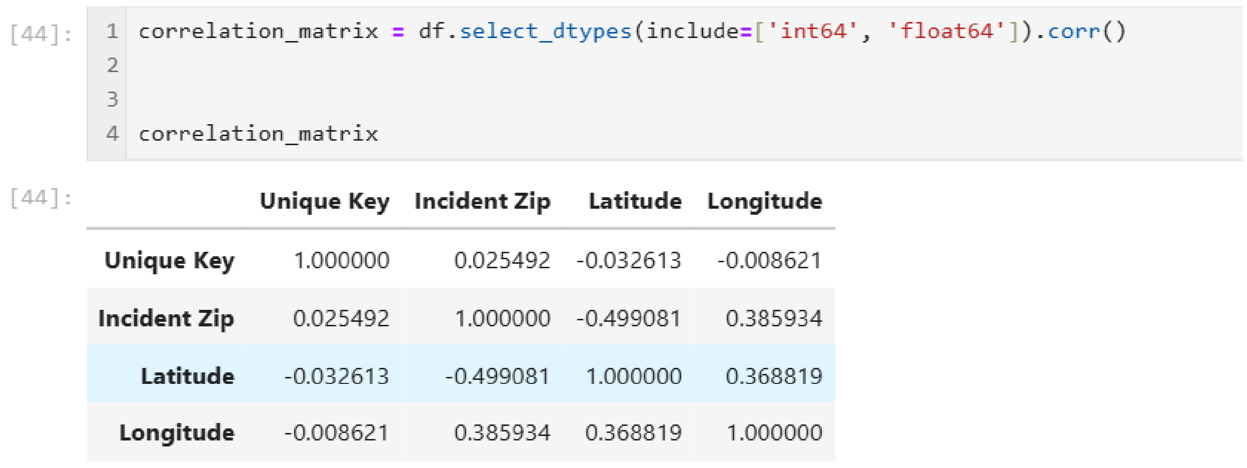
- **0** means no linear correlation. (Built In, 2023)

Results:

Unique Key:- This column shows almost **no correlation** with any of the other variables (values near 0), which is expected since "Unique Key" is just an identifier. It does not represent any measurable quantity that should vary with other features.

Incident Zip:- Shows a moderate negative correlation with Latitude (-0.4991). This suggests that as zip codes increase numerically, the latitude decreases. In NYC, this makes sense geographically northern areas generally have lower zip codes. Has a moderate positive correlation with Longitude (0.3859), meaning zip codes tend to increase as we move eastward across the city.

Latitude and Longitude:- These two have a moderate positive correlation (0.3688), indicating that northern and eastern locations often occur together in the dataset. This fits NYC’s layout, where some boroughs (like the Bronx and Queens) stretch diagonally across the city.

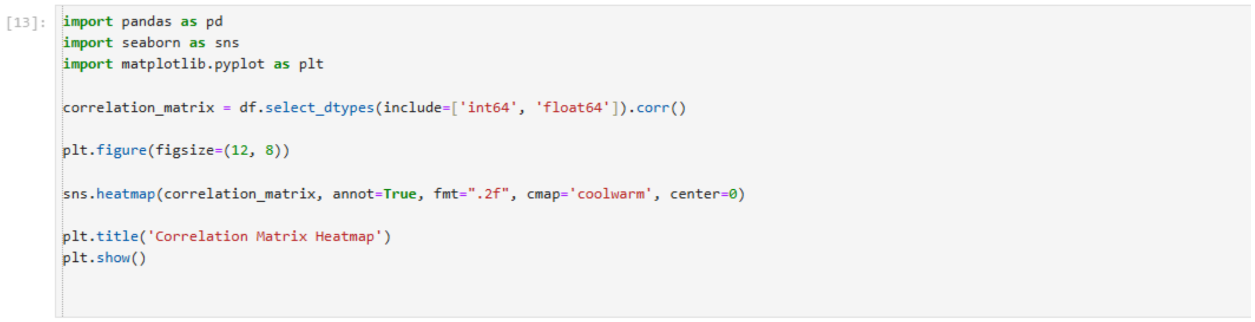


```
[44]: correlation_matrix = df.select_dtypes(include=['int64', 'float64']).corr()


      correlation_matrix
```

[44]:

	Unique Key	Incident Zip	Latitude	Longitude
Unique Key	1.000000	0.025492	-0.032613	-0.008621
Incident Zip	0.025492	1.000000	-0.499081	0.385934
Latitude	-0.032613	-0.499081	1.000000	0.368819
Longitude	-0.008621	0.385934	0.368819	1.000000

Figure 11:Calculate Correlation matrix



```
[13]: import pandas as pd
      import seaborn as sns
      import matplotlib.pyplot as plt

      correlation_matrix = df.select_dtypes(include=['int64', 'float64']).corr()

      plt.figure(figsize=(12, 8))

      sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', center=0)

      plt.title('Correlation Matrix Heatmap')
      plt.show()
```

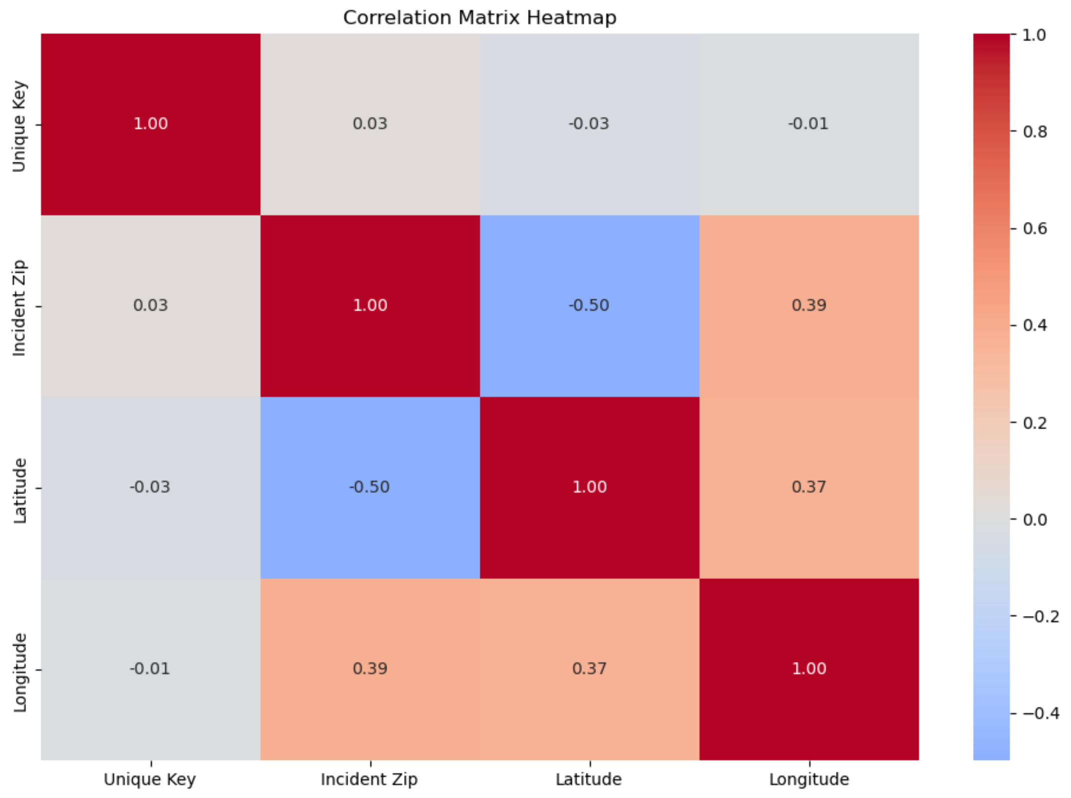


Figure 12: Heatmap of correlation

4. Data Exploration

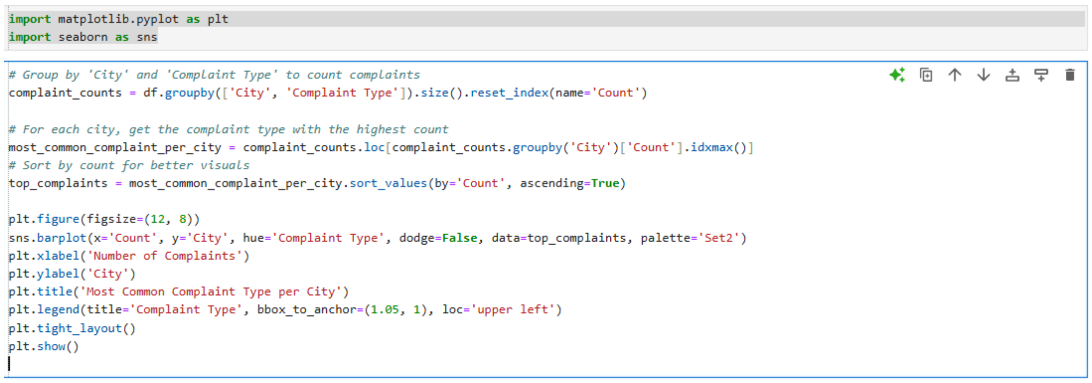
4.1 Provide four major insights through visualization that you come up after data mining.

Insights 1: Most Common Complaint Type per City

From the data we have we can get what kinds of complaints are most common in different cities. In code, it first groups the data by both 'City' and 'Complaint Type' to count how often each complaint happens in each city. Then, it picks the most common complaint type for every city and ranks the results to make the information easier to understand. Using the matplotlib libraries, the code creates a horizontal bar chart. Each bar shows a city and its most common complaint type, using different colors to make it clear. This kind of chart is useful for seeing which problems are biggest in which places, which can help city planners or local leaders decide where to focus their efforts.

I chose a **bar chart** to visualize the most common complaint types per city because it clearly compares categories across multiple locations, making it easy to identify which complaint types dominate in each area.

The graph shows that Blocked Driveway and Illegal Parking are the most common complaints in many cities, especially in places like Howard Beach, Queens, and Brooklyn. In these areas, the number of complaints can go as high as 25,000. On the other hand, complaints like Noise - Commercial and Noise - Street/Sidewalk happen much less often and are at the bottom of the list. Cities like Staten Island and the Bronx have fewer complaints overall, which might mean these problems aren’t as common there. The graph helps show that cities with more people and traffic tend to have more parking-related complaints. This analysis highlights geographic patterns in urban complaints and can guide city officials in prioritizing resources and addressing localized issues more effectively.



```python
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
# Group by 'City' and 'Complaint Type' to count complaints
complaint_counts = df.groupby(['City', 'Complaint Type']).size().reset_index(name='Count')

# For each city, get the complaint type with the highest count
most_common_complaint_per_city = complaint_counts.loc[complaint_counts.groupby('City')['Count'].idxmax()]
# Sort by count for better visuals
top_complaints = most_common_complaint_per_city.sort_values(by='Count', ascending=True)

plt.figure(figsize=(12, 8))
sns.barplot(x='Count', y='City', hue='Complaint Type', dodge=False, data=top_complaints, palette='Set2')
plt.xlabel('Number of Complaints')
plt.ylabel('City')
plt.title('Most Common Complaint Type per City')
plt.legend(title='Complaint Type', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

Figure 13:Most Common Complaint Type per City Code

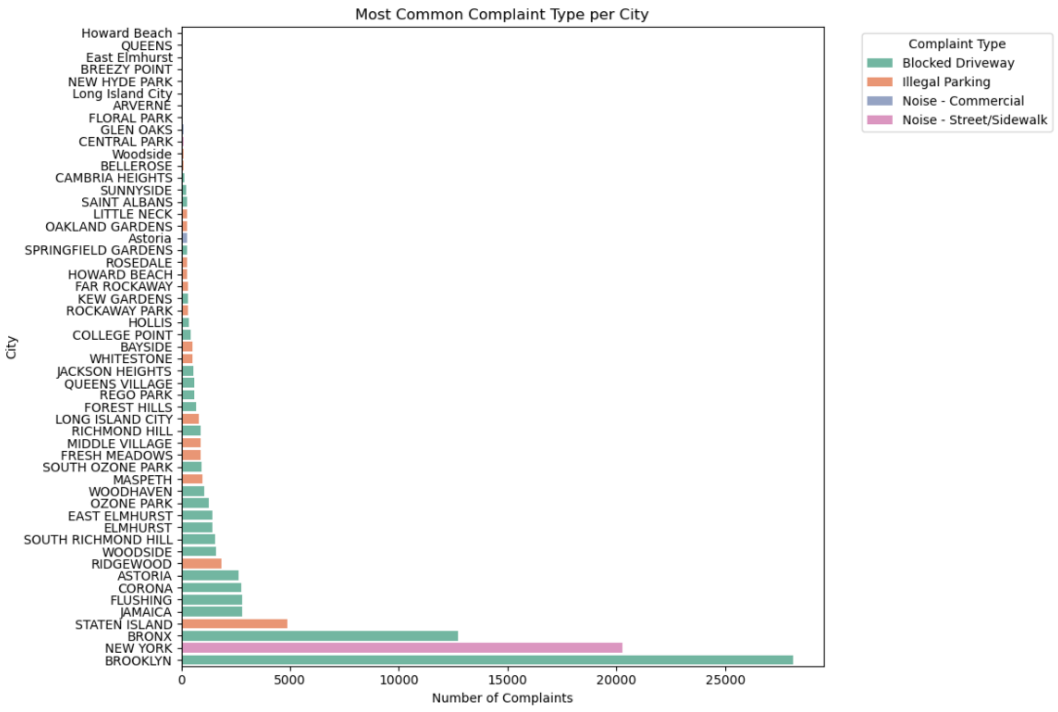
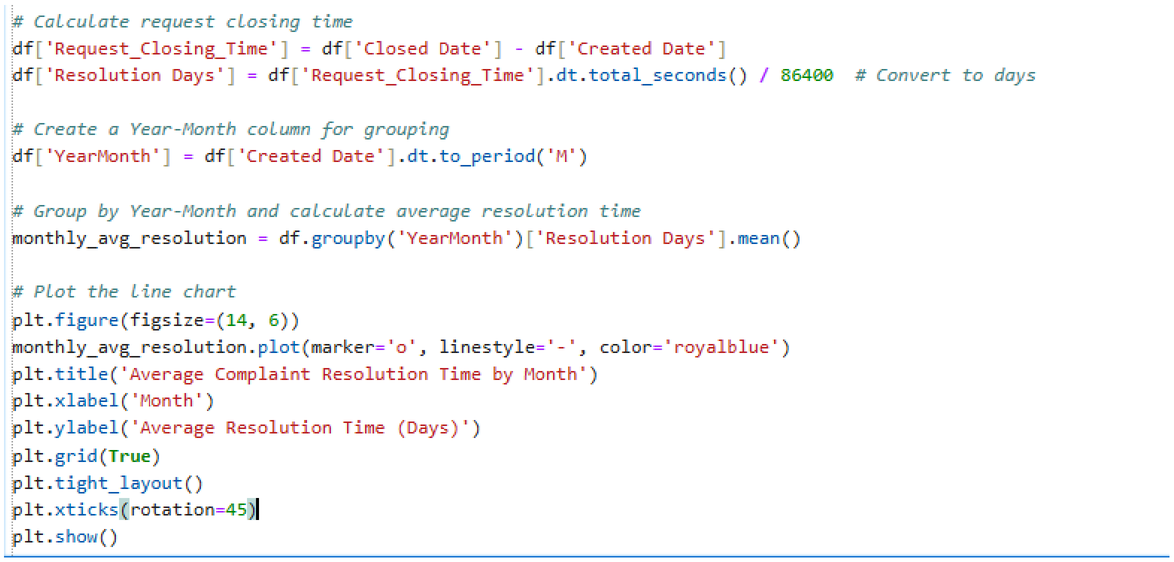


Figure 14: Most Common Complaint Type per City Graph

Insights 2: Time Series of Resolution Efficiency

The second insights we can get is Track how average resolution time has improved or worsened month by month. To get these insights first we calculate how long each complaint took to resolve by subtracting the date it was created from the date it was closed. It then converts that time into days and finds the average for each month. The code uses Matplotlib to make a line chart with markers, labels, and grid lines so it's easier to read. This kind of chart is useful for spotting which months had slower responses and can help agencies figure out how to improve their performance in the future.

The line graph was used, as it effectively highlights trends over time and makes it easy to observe changes in performance month by month. The average time to resolve complaints started around 0.16 days in March and steadily rose to nearly 0.2 days by December. A minor dip occurred in April, after which resolution times began a consistent climb, with only a slight decrease in November. This upward trend may reflect seasonal impacts, increasing complaint volume, or declining operational efficiency toward the end of the year. Overall, the data suggests that as the year progressed, the city faced growing challenges in maintaining quick resolution times. This may warrant a review of complaint handling workflows or resource allocation strategies during high-volume periods.



```
# Calculate request closing time
df['Request_Closing_Time'] = df['Closed Date'] - df['Created Date']
df['Resolution Days'] = df['Request_Closing_Time'].dt.total_seconds() / 86400  # Convert to days

# Create a Year-Month column for grouping
df['YearMonth'] = df['Created Date'].dt.to_period('M')

# Group by Year-Month and calculate average resolution time
monthly_avg_resolution = df.groupby('YearMonth')['Resolution Days'].mean()

# Plot the line chart
plt.figure(figsize=(14, 6))
monthly_avg_resolution.plot(marker='o', linestyle='-', color='royalblue')
plt.title('Average Complaint Resolution Time by Month')
plt.xlabel('Month')
plt.ylabel('Average Resolution Time (Days)')
plt.grid(True)
plt.tight_layout()
plt.xticks(rotation=45)
plt.show()
```

Figure 15: Time Series of Resolution Efficiency code

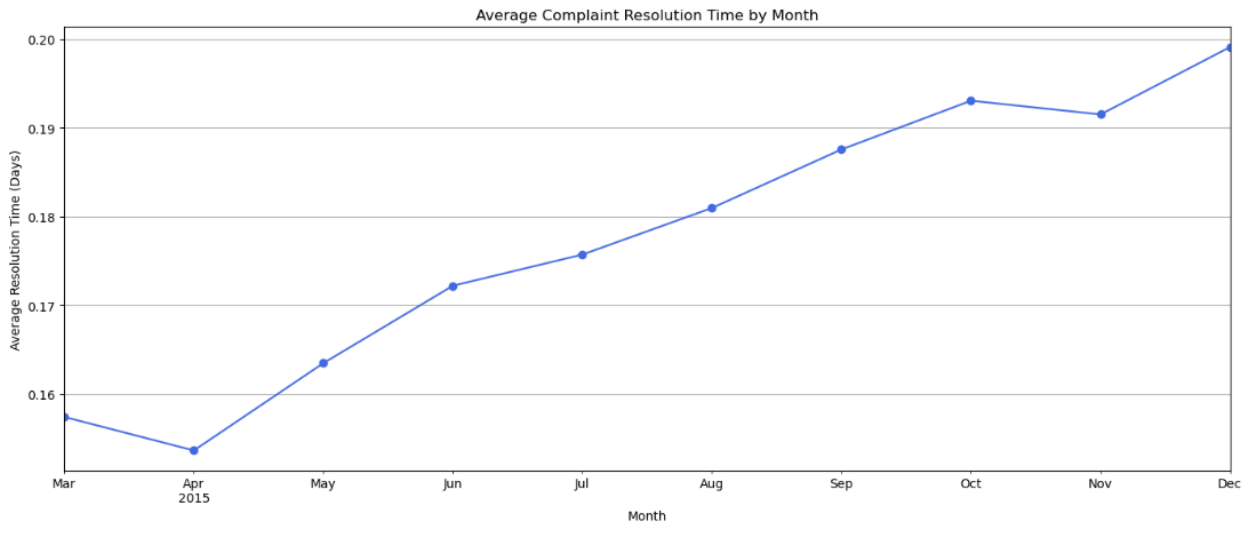


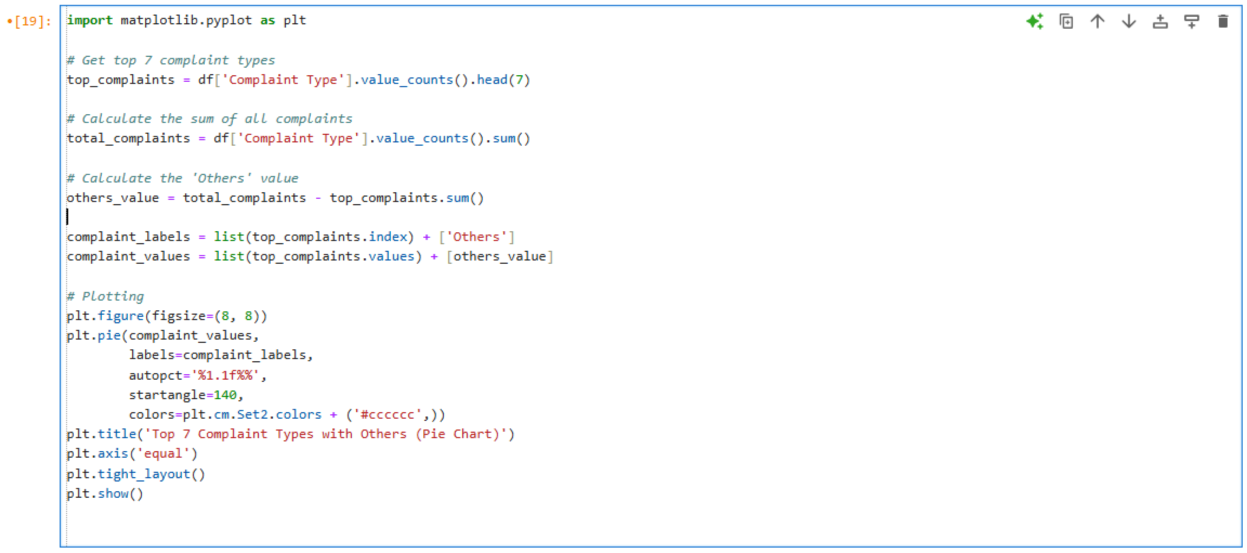
Figure 16: Time Series of Resolution Efficiency Graph

Insights 3: Top complaints type

These insights give us which issues occur most often. The Python code that creates this chart works by counting how often each type of complaint shows up, picking the top 7, and then making a pie chart. It uses autopct to show the exact percentages, sets a starting angle for better layout, and uses nice colors with plt.cm.Set2.colors to make it easy to read. This kind of chart helps decision-makers see which problems affect the most people and where they should put their time and resources.

The pie chart shows the distribution of the top 7 most common complaint types submitted, along with an aggregated “Others” category that includes all less frequent complaints. The chart clearly shows that **"Blocked Driveway" (26.3%)** and **"Illegal Parking" (25.4%)** are by far the most frequently reported issues, together making up over half of all complaints. These are followed by **"Noise - Street/Sidewalk" (16.4%)** and **"Noise - Commercial" (12.1%)**, indicating that noise-related disturbances are also a significant concern in the community. **"Derelict Vehicle" (6.0%)**, **"Noise - Vehicle" (5.8%)**, and **"Animal Abuse" (2.7%)** appear much less frequently in comparison. The **“Others”** category accounts for **5.3%**, capturing all remaining complaint types not in the top seven.

This chart is helpful for city officials because it shows what problems people care about most. Since over half the complaints are about parking, the city could focus on fixing parking issues or enforcing the rules better. Also, since many people complained about noise, there might need to be stricter noise rules or better enforcement in busy areas. Overall, this data helps leaders make better decisions about where to focus their time and resources.



```
[19]:
import matplotlib.pyplot as plt

# Get top 7 complaint types
top_complaints = df['Complaint Type'].value_counts().head(7)

# Calculate the sum of all complaints
total_complaints = df['Complaint Type'].value_counts().sum()

# Calculate the 'Others' value
others_value = total_complaints - top_complaints.sum()

complaint_labels = list(top_complaints.index) + ['Others']
complaint_values = list(top_complaints.values) + [others_value]

# Plotting
plt.figure(figsize=(8, 8))
plt.pie(complaint_values,
        labels=complaint_labels,
        autopct='%1.1f%%',
        startangle=140,
        colors=plt.cm.Set2.colors + ('#cccccc',))
plt.title('Top 7 Complaint Types with Others (Pie Chart)')
plt.axis('equal')
plt.tight_layout()
plt.show()
```

Figure 17: Top complaints type code

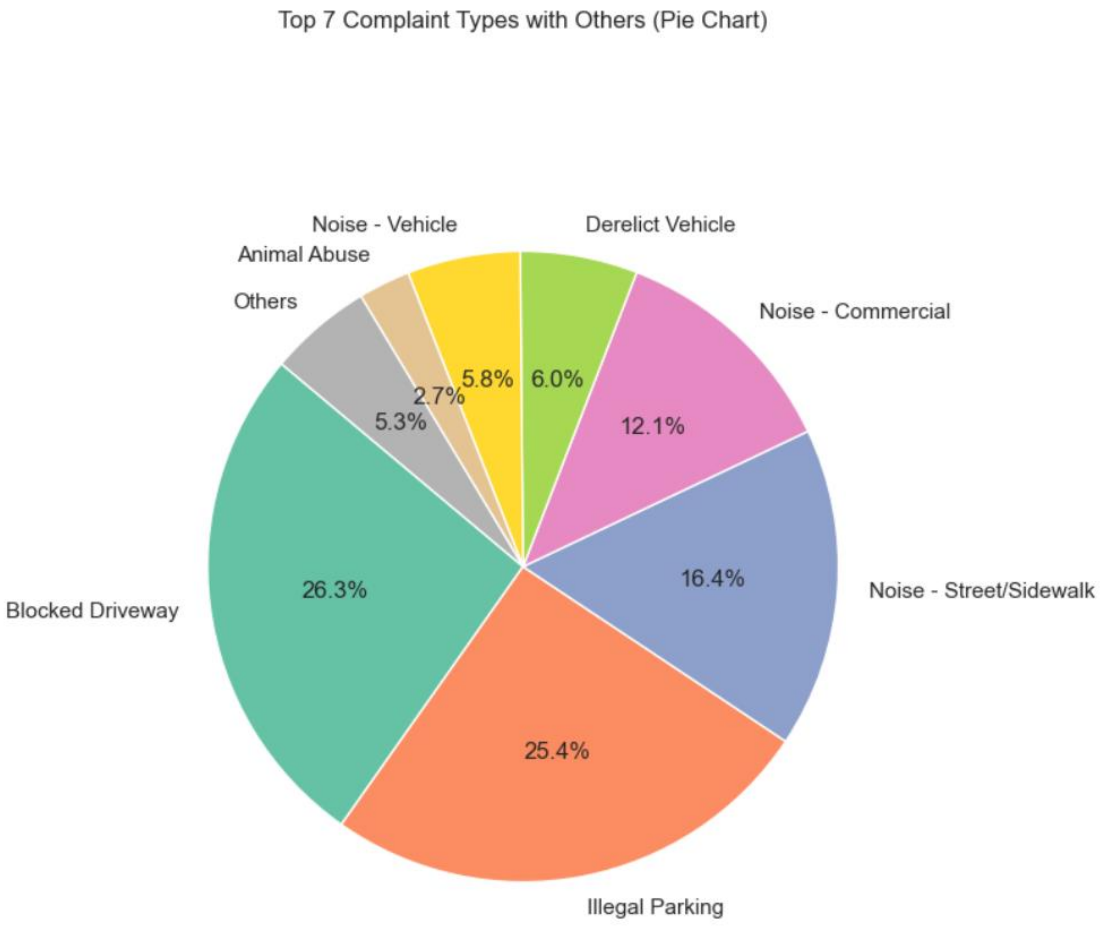
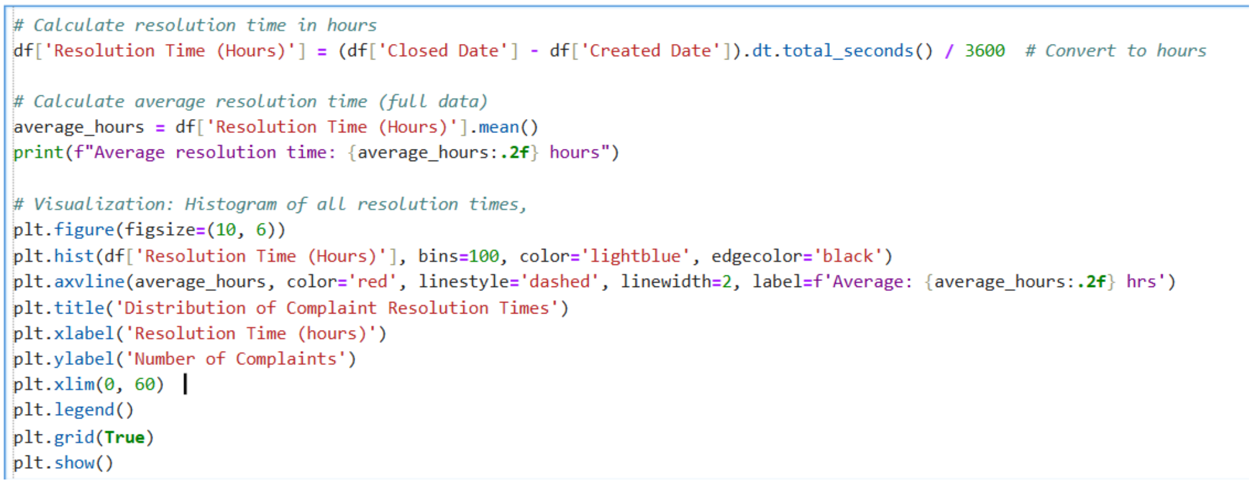


Figure 18: Top complaints type graph

Insights 4: Average Request Closing Time

This code calculates how long it takes to resolve complaints by finding the time difference between when each complaint was created and when it was closed, then converting that into hours. It figures out the average resolution time about 4.31 hours and creates a histogram with 100 bins to show how these times are spread out. A red dashed line shows where the average falls, and things like grid lines, axis labels, and clear limits make the chart easy to read.

From the histogram, it’s clear that most complaints (probably more than 90%) get resolved pretty quickly usually within 5 to 10 hours. The highest bar is near the start of the timeline, which matches the average time of 4.31 hours. By cutting off the x-axis at 60 hours, the graph leaves out extreme outliers (like those that took 600+ hours from earlier analysis). This helps highlight what normally happens rather than focusing on rare cases. The chart shows that while most complaints are handled efficiently, there are still some that take longer and may need further review. This kind of visual is helpful for tracking performance and figuring out where the system could be improved.



```
# Calculate resolution time in hours
df['Resolution Time (Hours)'] = (df['Closed Date'] - df['Created Date']).dt.total_seconds() / 3600  # Convert to hours

# Calculate average resolution time (full data)
average_hours = df['Resolution Time (Hours)'].mean()
print(f"Average resolution time: {average_hours:.2f} hours")

# Visualization: Histogram of all resolution times,
plt.figure(figsize=(10, 6))
plt.hist(df['Resolution Time (Hours)'], bins=100, color='lightblue', edgecolor='black')
plt.axvline(average_hours, color='red', linestyle='dashed', linewidth=2, label=f'Average: {average_hours:.2f} hrs')
plt.title('Distribution of Complaint Resolution Times')
plt.xlabel('Resolution Time (hours)')
plt.ylabel('Number of Complaints')
plt.xlim(0, 60)
plt.legend()
plt.grid(True)
plt.show()
```

Figure 19: Average Request Closing Time Code

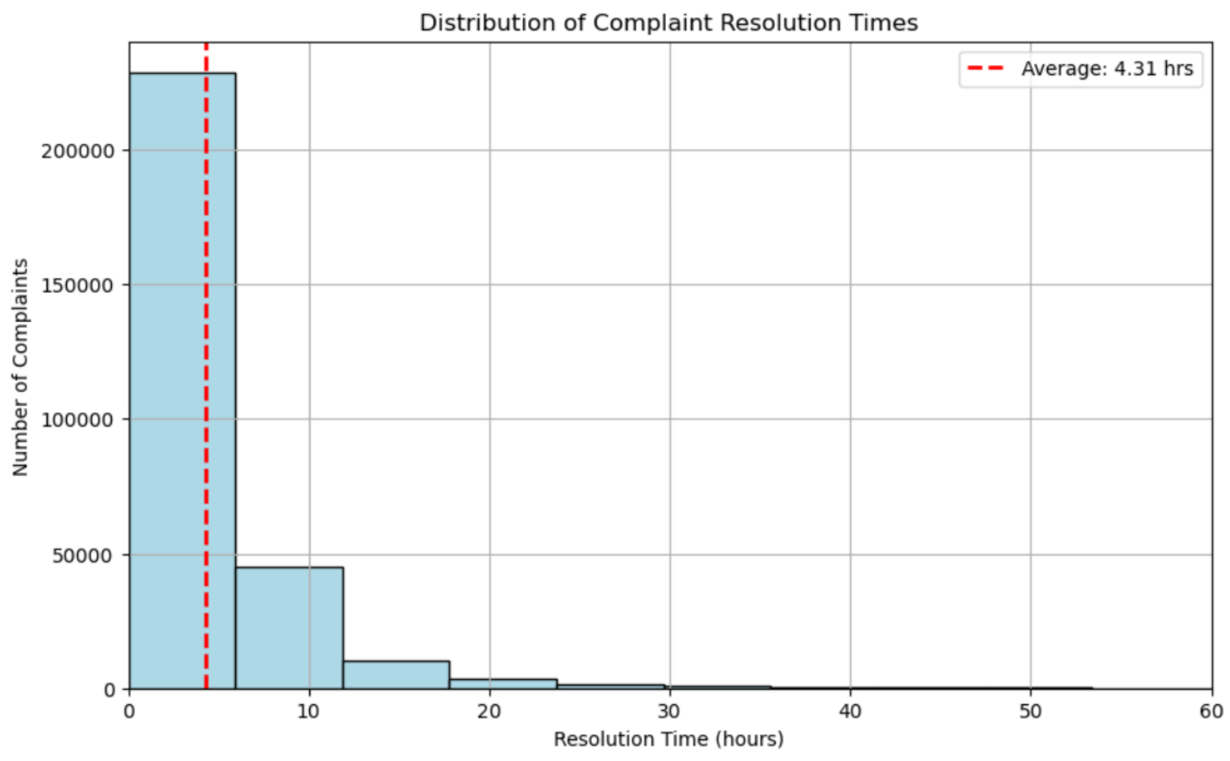


Figure 20: Average Request Closing Time graph

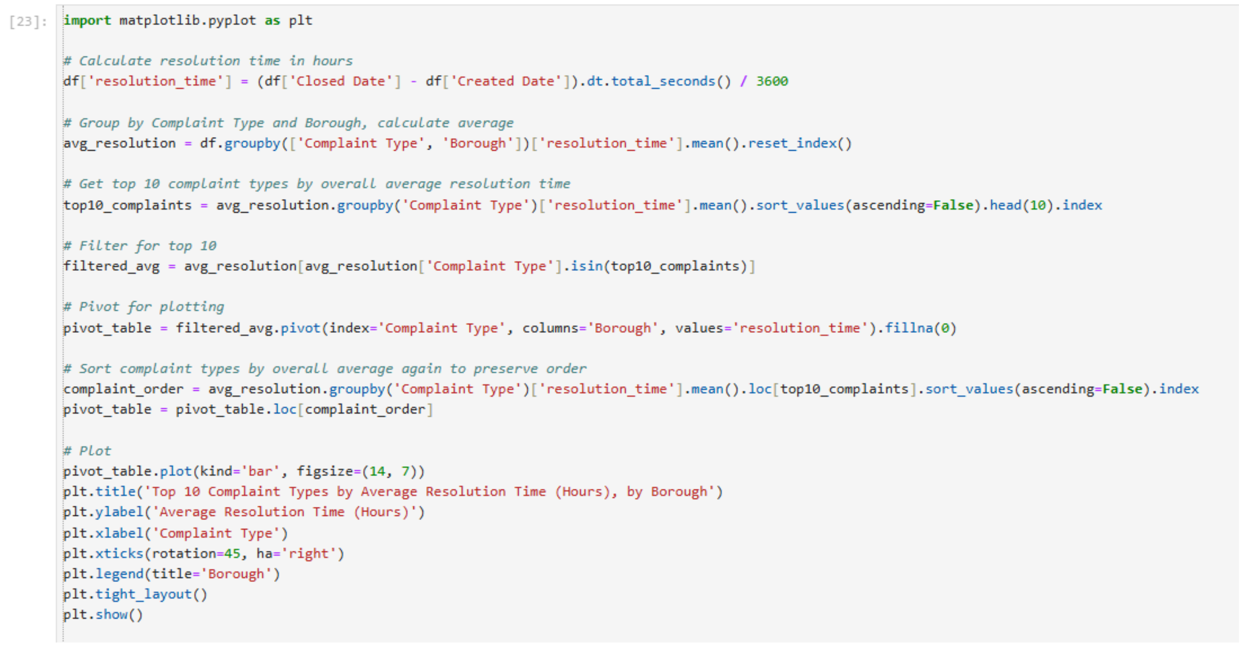
4.2 Arrange the complaint types according to their average 'Request_Closing_Time', categorized by various locations. Illustrate it through graph as well.

We need to make a graph chart for complaint types according to their average 'Request_Closing_Time', categorized by various locations. The analysis starts by calculating the resolution time for each complaint, measured in hours from when it was created to when it was closed. The data is then grouped by complaint type and borough to find the average resolution time for each combination. To highlight delays, the top 10 complaint types with the highest overall average resolution times are identified. The data is filtered to include only these complaints and reshaped into a table showing the average resolution time per borough for each complaint type. A grouped bar chart is created to compare how each borough handles these slowest complaint types. This visual makes it easy to spot which issues take the longest to resolve and how performance varies across boroughs, helping identify areas needing improvement.

The bar graph shown "Top 10 Complaint Types by Average Resolution Time (Hours), by Borough" shows how different types of complaints are handled across New York City’s boroughs. It reveals that some complaints, like "Griffiti" and "Derelict vehicle," take much longer to resolve than others, suggesting issues in how these are managed. There are clear differences between boroughs complaints in the Bronx and Staten

Island tend to take longer, while Manhattan and Queens often resolve the same issues faster, possibly because of better resources or more efficient systems.

Some complaints, like "Blocked driveway," show big time gaps between boroughs, which points to unequal service delivery. The grouped bar chart clearly compares these differences and shows where improvements are needed. To make the system fairer and faster, the city could invest more in slower boroughs like Staten Island, focus on long-resolution complaints like "graffiti," and try to make the process more consistent across all areas. This kind of analysis helps city officials make smarter decisions to improve public services.



```
[23]: import matplotlib.pyplot as plt

# Calculate resolution time in hours
df['resolution_time'] = (df['Closed Date'] - df['Created Date']).dt.total_seconds() / 3600

# Group by Complaint Type and Borough, calculate average
avg_resolution = df.groupby(['Complaint Type', 'Borough'])['resolution_time'].mean().reset_index()

# Get top 10 complaint types by overall average resolution time
top10_complaints = avg_resolution.groupby('Complaint Type')['resolution_time'].mean().sort_values(ascending=False).head(10).index

# Filter for top 10
filtered_avg = avg_resolution[avg_resolution['Complaint Type'].isin(top10_complaints)]

# Pivot for plotting
pivot_table = filtered_avg.pivot(index='Complaint Type', columns='Borough', values='resolution_time').fillna(0)

# Sort complaint types by overall average again to preserve order
complaint_order = avg_resolution.groupby('Complaint Type')['resolution_time'].mean().loc[top10_complaints].sort_values(ascending=False).index
pivot_table = pivot_table.loc[complaint_order]

# Plot
pivot_table.plot(kind='bar', figsize=(14, 7))
plt.title('Top 10 Complaint Types by Average Resolution Time (Hours), by Borough')
plt.ylabel('Average Resolution Time (Hours)')
plt.xlabel('Complaint Type')
plt.xticks(rotation=45, ha='right')
plt.legend(title='Borough')
plt.tight_layout()
plt.show()
```

Figure 21: Top 10 Complaint Types by Average Resolution Time (Hours), by Borough Code

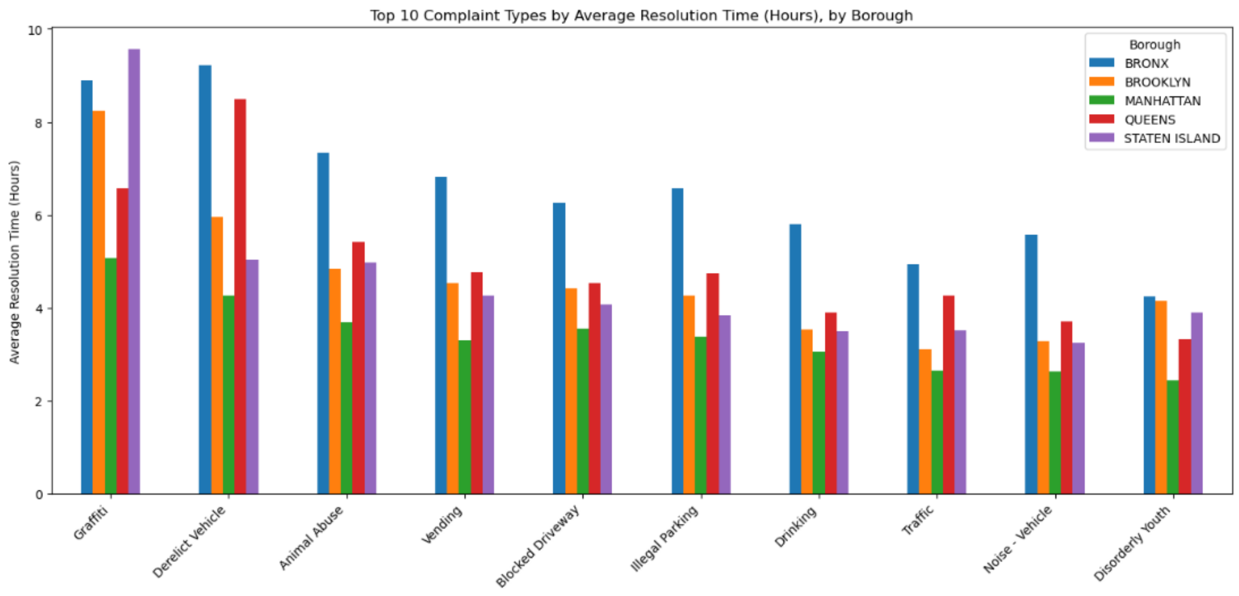


Figure 22: Top 10 Complaint Types by Average Resolution Time (Hours), by Borough graph

5. Statistical Testing

Test 1: Whether the average response time across complaint types is similar or not.

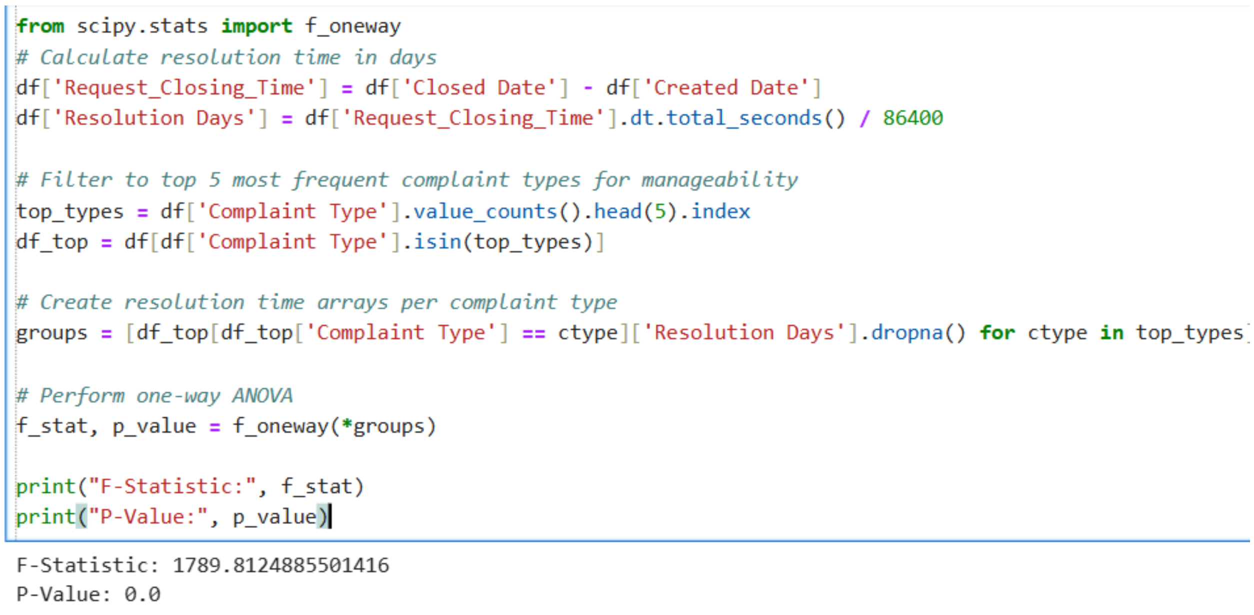
The **null hypothesis** (H_0) suggests there is no effect, difference, or relationship between variables and serves as the starting assumption in hypothesis testing. It is rejected only if strong evidence supports the **alternative hypothesis** (H_1 or Ha), which proposes that a significant effect or relationship exists. (Turney, 2022)

H_0 (Null Hypothesis): The average response time is the same across all complaint types.

H_1 (Alternate Hypothesis): At least one complaint type has a different average response time.

The **p-value** measures the strength of evidence against the null hypothesis. A **low p-value** (≤ 0.05) indicates strong evidence to reject the null hypothesis, suggesting a statistically significant result. A **high p-value** (> 0.05) suggests weak evidence against the null hypothesis, meaning the result is not statistically significant. (Bevans, 2020)

The p-value is 0.0 in our result it means we reject the null hypothesis. This means there is a statistically significant difference in the average response times among different complaint types.



```python
from scipy.stats import f_oneway
# Calculate resolution time in days
df['Request_Closing_Time'] = df['Closed Date'] - df['Created Date']
df['Resolution Days'] = df['Request_Closing_Time'].dt.total_seconds() / 86400

# Filter to top 5 most frequent complaint types for manageability
top_types = df['Complaint Type'].value_counts().head(5).index
df_top = df[df['Complaint Type'].isin(top_types)]

# Create resolution time arrays per complaint type
groups = [df_top[df_top['Complaint Type'] == ctype]['Resolution Days'].dropna() for ctype in top_types

# Perform one-way ANOVA
f_stat, p_value = f_oneway(*groups)

print("F-Statistic:", f_stat)
print("P-Value:", p_value)
```

```
F-Statistic: 1789.8124885501416
P-Value: 0.0
```

Figure 23:Checking similarities between average response time and complaint types

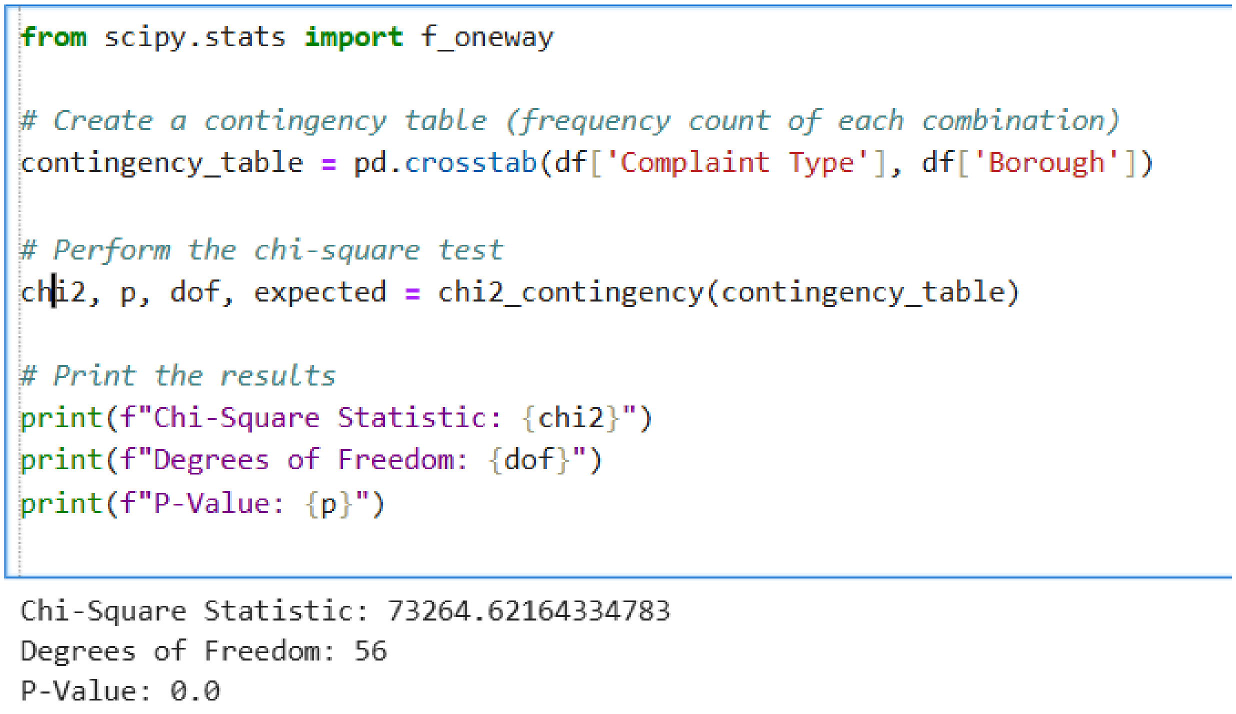
Test 2: Whether the type of complaint or service requested and location are related.

Null Hypothesis (H_0): Complaint type and location are independent (not related).

Alternate Hypothesis (H_1): Complaint type and location are dependent (related).

The Chi-Square (χ^2) test is a statistical method used to determine whether there is a significant relationship between two categorical variables. If the difference between observed and expected values is large, the χ^2 statistic will be high. A low p-value (typically < 0.05) means you reject the null hypothesis → the two variables are related. In a Chi-Square Test of Independence, the degrees of freedom (df) represent the number of values in the calculation that are free to vary. It affects the Chi-Square distribution used to calculate the p-value. More degrees of freedom means the test is more sensitive to differences between observed and expected values(Avijeet Biswal, 2021).

We tested whether complaint type is related to location using a Chi-Square test of independence. The result gave a Chi-Square Statistic of 73,264.62, degrees of freedom = 56, and a p-value = 0.0. Since the p-value is much less than 0.05, it reject the null hypothesis. This means there is a significant relationship between complaint type and location certain complaints are more common in specific areas.


```
from scipy.stats import f_oneway

# Create a contingency table (frequency count of each combination)
contingency_table = pd.crosstab(df['Complaint Type'], df['Borough'])

# Perform the chi-square test
chi2, p, dof, expected = chi2_contingency(contingency_table)

# Print the results
print(f"Chi-Square Statistic: {chi2}")
print(f"Degrees of Freedom: {dof}")
print(f"P-Value: {p}")
```

```
Chi-Square Statistic: 73264.62164334783
Degrees of Freedom: 56
P-Value: 0.0
```

Figure 24: Check the complaint type and borough are related or not.

References

Bevans, R. (2020). *Understanding P values* | *Definition and Examples*. [online] Scribbr. Available at: https://www.scribbr.com/statistics/p-value/ [Accessed 15 May 2025].

Turney, S. (2022). *Null and Alternative Hypotheses* | *Definitions & Examples*. [online] Scribbr. Available at: https://www.scribbr.com/statistics/null-and-alternative-hypotheses/ [Accessed 15 May 2025].

Avijeet Biswal (2021). *Chi-Square Test: Formula, Types, & Examples*. [online] Simplilearn.com. Available at: https://www.simplilearn.com/tutorials/statistics-tutorial/chi-square-test [Accessed 15 May 2025].

Built In. (2023). *Introduction to the Correlation Matrix* | *Built In*. [online] Available at: https://builtin.com/data-science/correlation-matrix [Accessed 15 May 2025].