# Module Code & Module Title

**CC5067NI Smart Data Discovery**

**60% Individual Coursework**

**Submission : Final Submission**

**Academic Semester: Spring Semester 2025**

**Credit: 15 credit semester long module**

**Student Name: Salina Thing**

**London Met ID: 23047540**

**College ID: np01cp4a230017**

**Assignment Due Date: Thursday, May 15, 2025**

**Assignment Submission Date: Thursday, May 15, 2025**

**Submitted To: Dipeshor Silwal**

*I confirm that I understand my coursework needs to be submitted online via MST Classroom under the relevant module page before the deadline in order for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded*

# 23047540 Salina Thing_Python - Copy1.docx

Islington College,Nepal

## Document Details

Submission ID

**trn:oid:::3618:96004515**

Submission Date

**May 15, 2025, 12:48 PM GMT+5:45**

Download Date

**May 15, 2025, 12:53 PM GMT+5:45**

File Name

**23047540 Salina Thing_Python - Copy1.docx**

File Size

**45.8 KB**

**64 Pages**

**6,529 Words**

**38,647 Characters**





# 24% Overall Similarity

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

## Match Groups

- **124** Not Cited or Quoted 21%
  Matches with neither in-text citation nor quotation marks
- **6** Missing Quotations 1%
  Matches that are still very similar to source material
- **10** Missing Citation 2%
  Matches that have quotation marks, but no in-text citation
- **0** Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

## Top Sources

11% Internet sources

4% Publications

21% 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

# Table of Tables

# Table of Figures

# 1. Data Understanding

The dataset given in the **60% coursework** of module **CC5067NI Smart Data Discovery** is mainly focused on data analysis of **311 Customer service Requests** in **New York City (NYC).** Each day, NYC 311 receives complaints related to non-emergency issues reported by locals such as noise complaints, illegal parking, blocked roads, and other concerns.

NYC main goals are to provide quick services of government to the public with best customer service. These issues are noted by NYC311 and forwarded to respective agencies like police, building, transport and so on. The agency responds to the request, go through it and end it (Shubham, 2021). Each row includes a single complaint with more detail's information like date and time the request was created and closed, the types of issues, the location, the city and agency responsible for, direction of location and many more.

The dataset is used by New York City agencies to track local frequent complaints pattern, service delivery, analyse average request resolution time, performance evaluation in different city agencies and so on. Simply, it reviews the services are provided to locals on time and improve services (MotherDuck, 2025) (Shubham, 2021). This data is publicly available on the NYC Open Data Portal and typically exported as CSV file.

*Figure 1: Logo of data discover*

The table below summarizes the columns of datasets with descriptions with data types:

| S.N. | Column Name | Description | Data Type |
|---|---|---|---|
| 1 | Unique Key | Every service request has a distinct number. | Integer |
| 2 | Created Date | Time of the complaint's creation | Object / DateTime |
| 3 | Closed Date | Time of the complaint's closure | Object / DateTime |
| 4 | Agency | Agency code for handling the request | String |
| 5 | Agency Name | The complete name of the organization making the request | String |
| 6 | Complaint Type | Type or category of general complaints | String |
| 7 | Descriptor | An in-depth description of the issue | String |
| 8 | Location Type | The kind of place where the problem happened | String |
| 9 | Incident Zip | ZIP code of the incident location | Integer |
| 10 | Incident Address | Address where the incident took place | String |
| 11 | Street Name | The incident's street name | String |
| 12 | Cross Street 1 | Closest cross street | String |
| 13 | Cross Street 2 | Second closest cross street | String |
| 14 | Intersection Street 1 | Intersection detail | String |
| 15 | Intersection Street 2 | Another intersection details | String |
| 16 | Address Type | Type of address like residential, commercial | String |
| 17 | City | Name of the city | String |
| 18 | Landmark | Nearby Landmark | String |
| 19 | Facility Type | Type of city facility involved | String |
| 20 | Status | Status of complaint (e.g., Closed, Open) | String |
| 21 | Due Date | When the issue was expected to be resolved | DateTime |
| 22 | Resolution Description | Explanation of how the issue was addressed | String |
| 23 | Resolution Action updated date | Last update to the resolution | DateTime |
| 24 | Community Board | Local community board number | String |
| 25 | Borough | Borough where the complaint was made | String |

| | | | |
|---|---|---|---|
| 26 | X Coordinate (State Plane) | X coordinate in NYC State Plane projection | Float |
| 27 | Y Coordinate (State Plane) | Y coordinate in NYC State Plane projection | Float |
| 28 | Park Facility Name | Name of the park facility involved | String |
| 29 | Park Borough | Borough where park is located | String |
| 30 | School Name | Name of the school involved | String |
| 31 | School Number | School number | Integer |
| 32 | School Region | School Region | String |
| 33 | School code | Code of school | String |
| 34 | School Phone Number | School contact number | Integer |
| 35 | School Address | Address of the school | String |
| 36 | School city | City where school is located | String |
| 37 | School state | State where school is located | String |
| 38 | School zip | Zip code of the school | Integer |
| 39 | School Not Found | Indicates whether a school wasn't found | String/Boolean |
| 40 | School or Citywide Complaint | Indicates if the complaints is from school or citywide. | String |
| 41 | Vehicle Type | Type of vehicle invovled | String |
| 42 | Taxi Company Borough | Borough where taxi company is registered | String |
| 43 | Taxi Pick Up Location | Location where the taxi picked up a passenger | String |
| 44 | Bridge Highway Name | Bridge or highway involved | String |
| 45 | Bridge Highway Direction | Direction on the bridge/highway | String |
| 46 | Road Ramp | Ramp detail | String |
| 47 | Bridge Highway Segment | Segment detail | String |
| 48 | Garage Lot Name | Garage/lot name | String |
| 49 | Ferry Direction | Direction of the ferry | String |
| 50 | Ferry Terminal Name | Terminal from which ferry departs | String |
| 51 | Latitude | Geographic latitude of the complaint location | Float |
| 52 | Longitude | Geographic longitude of the complaint location | Float |
| 53 | Location | Combined latitude/longitude or address | String |

*Table 1: List of columns of datasets with descriptions and datatype*

# 2. Data Preparation

## 2.1. Library used in my project

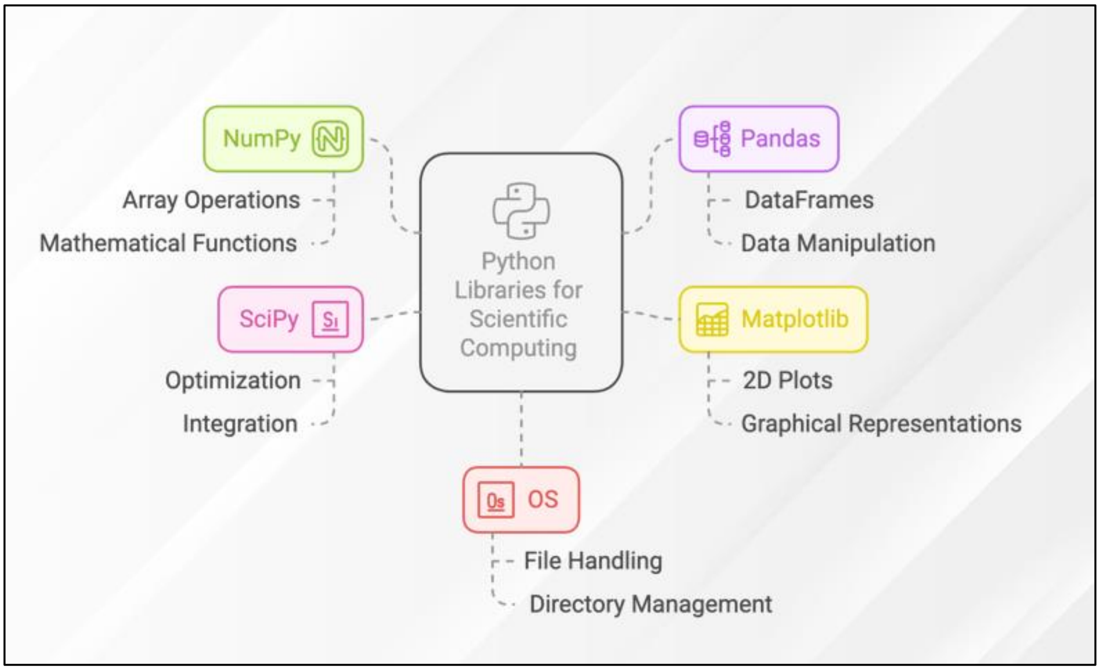



*Figure 2: Different library of python*

Normally, Python consists of a lot of libraries so that programmer can easily learn faster. It avoids redundancy and allow new user to learn it easily with the help of imported packages. It contains built-in modules that provide access to basic system functionality like I/O and some other core modules. In my code, I have used the packages like numpy, pandas, statistics and matplot.

a) **Numpy :** These days, the term "numpy" refers to the widely used library for numerical Python. Large matrices and multi-dimensional data are supported by this well-known machine learning framework. The main characteristic of this library is its array interface.

b) **Pandas:** For data scientists, the Pandas library is essential. A range of analysis tools and adaptable high-level data structures are offered by this open-source machine learning package. Pandas facilitates data conversions, iterations, visualization, and more.

c) **Matplot:** Plotting numerical data is the responsibility of this library, which is utilized in data analysis. Plotting high-definition figures like as pie charts, histograms, scatterplots, graphs, etc., it is also an open-source library.

d) **Scipy:** The acronym for "Scientific Python" is "SciPy." This library is open-source and used for complex scientific calculations. This library is based on a Numpy extension. Additionally, engineers and application developers use it extensively (GeeksforGeeks, 2024).

```
#importing packages
import pandas as pd
import numpy as np
import statistics
import scipy.stats as stats
import matplotlib.pyplot as plt
```

These libraries are imported in alias form like :

**Pandas → import pandas as pd**

**Numpy → import numpy as np**

**Matplot → import matplotlib.pyplot as plt**

**Scipy → import scipy.stats as stats**

**import statistics** of respectively packages.

Here, **pd, np, plt and stats** are the alias or short form of respective library mentioned above.

## 2.2. Import the dataset

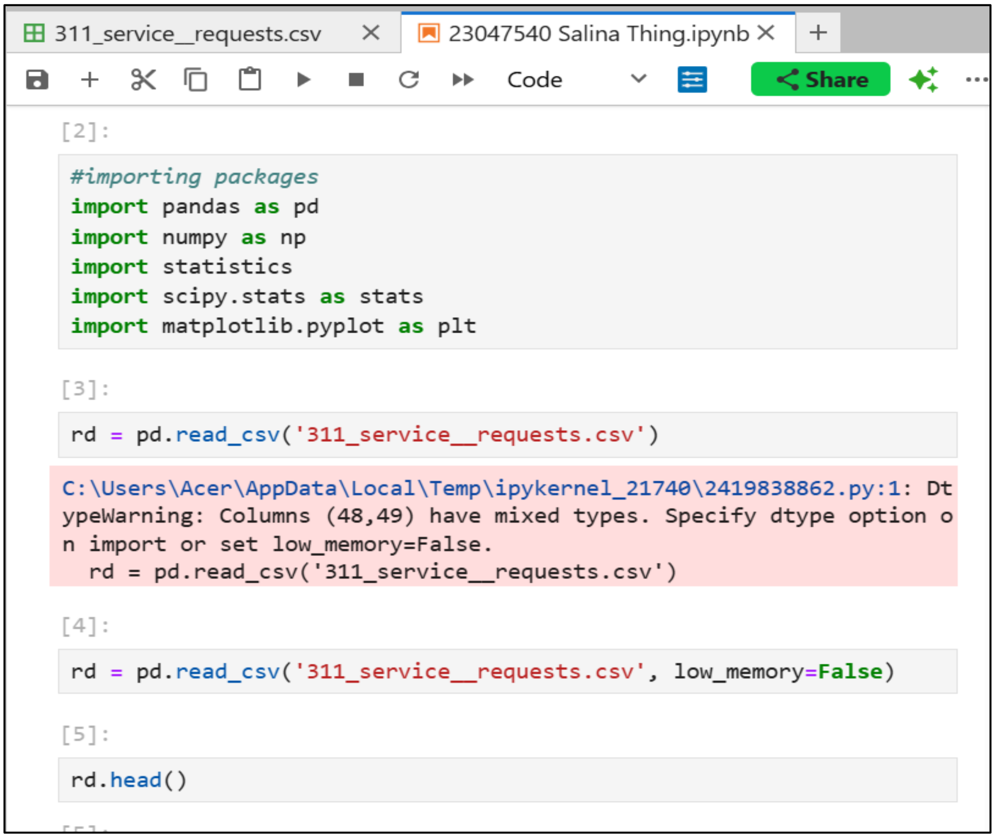



*Figure 3: Import dataset(i)*

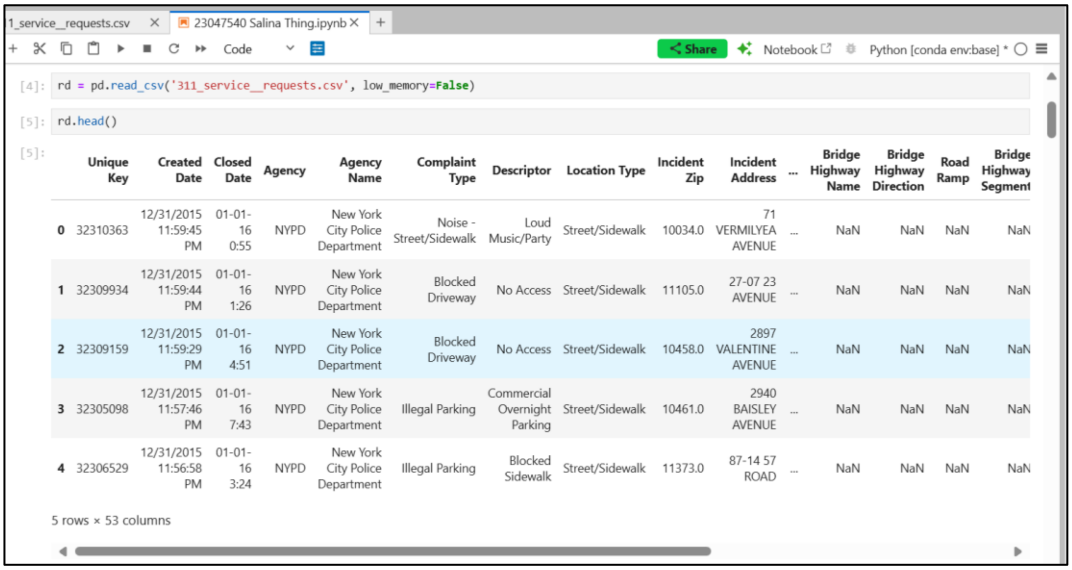
At first, we use different libraries like pandas, numpy, matplotlib, scipy and statistic for importing the datasets and for easy numeric calculation in the alias form.

### Line 1: Read the CSV file

```
rd = pd.read_csv('311_Service_Requests.csv')
```

We load given datasets in CSV **file name "311_service_request"** with variable name **'rd'** in pandas DataFrame.

**pd.read_csv()** is a pandas function that reads data from csv file and convert into dataframe.


[4]: rd = pd.read_csv('311_service__requests.csv', low_memory=False)

[5]: rd.head()

[5]:

| | Unique Key | Created Date | Closed Date | Agency | Agency Name | Complaint Type | Descriptor | Location Type | Incident Zip | Incident Address | ... | Bridge Highway Name | Bridge Highway Direction | Road Ramp | Bridge Highway Segment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 32310363 | 12/31/2015 11:59:45 PM | 01-01-16 0:55 | NYPD | New York City Police Department | Noise - Street/Sidewalk | Loud Music/Party | Street/Sidewalk | 10034.0 | 71 VERMILYEA AVENUE | ... | NaN | NaN | NaN | NaN |
| 1 | 32309934 | 12/31/2015 11:59:44 PM | 01-01-16 1:26 | NYPD | New York City Police Department | Blocked Driveway | No Access | Street/Sidewalk | 11105.0 | 27-07 23 AVENUE | ... | NaN | NaN | NaN | NaN |
| 2 | 32309159 | 12/31/2015 11:59:29 PM | 01-01-16 4:51 | NYPD | New York City Police Department | Blocked Driveway | No Access | Street/Sidewalk | 10458.0 | 2897 VALENTINE AVENUE | ... | NaN | NaN | NaN | NaN |
| 3 | 32305098 | 12/31/2015 11:57:46 PM | 01-01-16 7:43 | NYPD | New York City Police Department | Illegal Parking | Commercial Overnight Parking | Street/Sidewalk | 10461.0 | 2940 BAISLEY AVENUE | ... | NaN | NaN | NaN | NaN |
| 4 | 32306529 | 12/31/2015 11:56:58 PM | 01-01-16 3:24 | NYPD | New York City Police Department | Illegal Parking | Blocked Sidewalk | Street/Sidewalk | 11373.0 | 87-14 57 ROAD | ... | NaN | NaN | NaN | NaN |

5 rows × 53 columns

*Figure 4: Import dataset(ii)*

Here the solution has come to deal with errors as it is common **DtypeWarning** as it includes more columns (mixed data types) than its own range.i.e. 48 but the columns are 53. So, we can fix it by **setting low_memory=false** or suppress it.

```
rd = pd.read_csv('311_service_request.csv', low_memory=False)
```

**low_memory=False** → This tells pandas **not to guess data types in chunks**, which prevents warnings or mixed types. A DataFrame **rd** is created containing all the rows and columns from the CSV file.

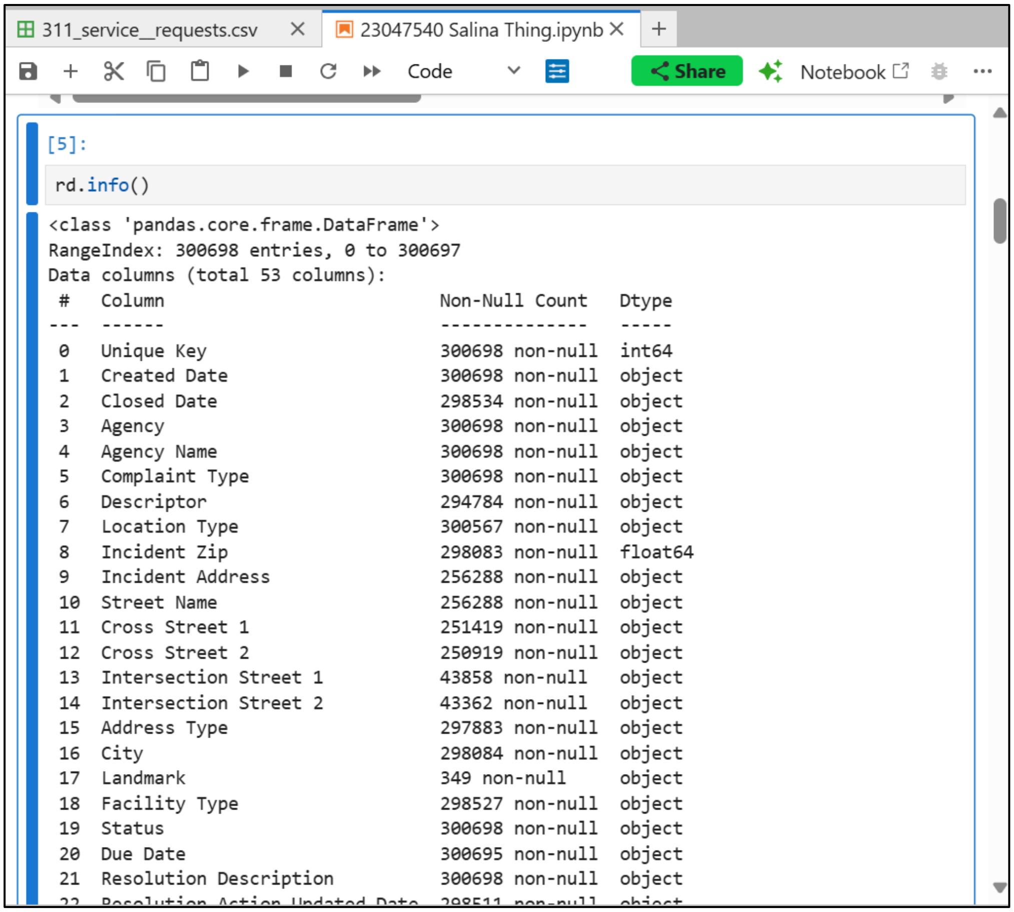
**Line 2: View the first 5 rows**

```
rd.head()
```

This is a pandas functions where head() shows first 5 rows by default but if you want more rows then you can pass the required integer to display required rows.

## 2.3. Insights on the dataset

- The dataset is about the complaints records of New York City done by the residents on 311 service requests.
- It includes a lot of complaints regarding complaint type, location, responsible agency, and service request resolution.
- Important columns include:
    - ❖ **Complaint Type**: Indicates the types of the problem.
    - ❖ **Created Date** and **Closed Date**: Includes the time of response to complete the problem.
    - ❖ **Status**: To identify whether the request is still pending or already solved or working on.
    - ❖ **Location Info**: Borough, City, Zip, street, intersection, Latitude, Longitude.
- This data is crucial for knowing public demands and needs and agency performance.
- It could make public freely to speak or report to the government services directly and get response on time.
- It also helps the respective departments to work on time due to providing the details more clearly.



311_service__requests.csv 23047540 Salina Thing.ipynb

```
[5]:
rd.info()

<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
```

*Figure 5: Screenshot of information of datasets*

It provides the summary of whole data with information like class, columns, datatype, index range and so on. Also, it helps in quick finding missing data and viewing data.

## 2.4. Convert "Created Date" and "Closed Date" to datetime + Create Request_Closing_Time

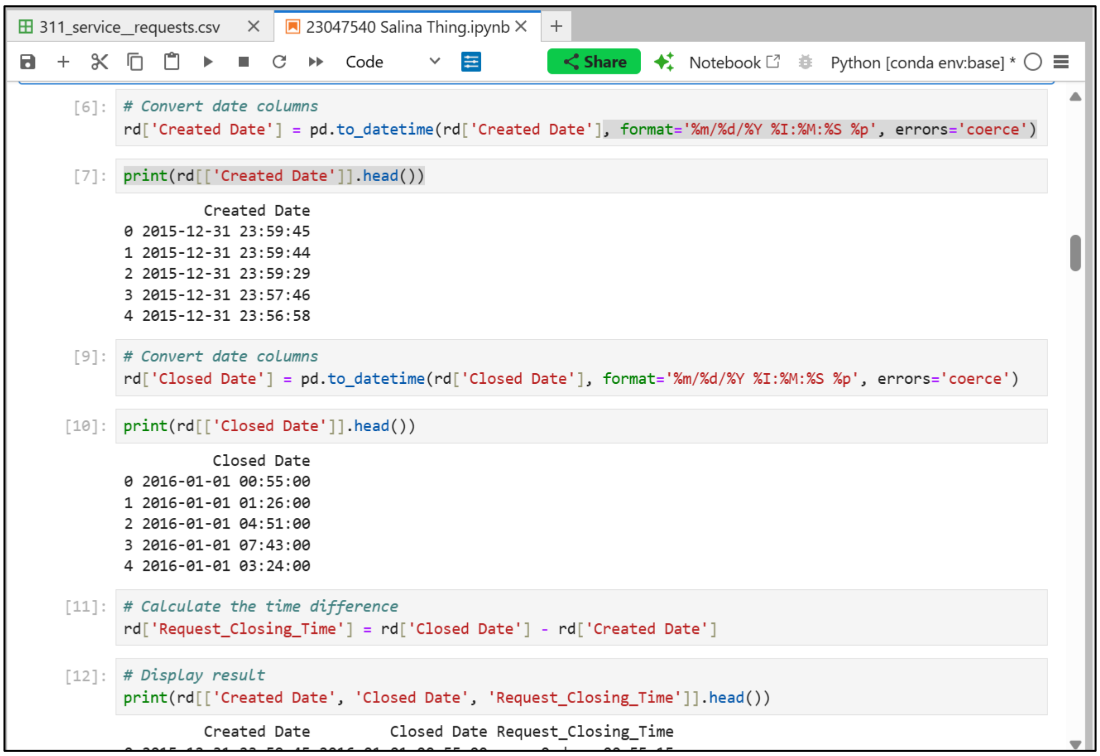


*Figure 6: Converting given data columns into "DateTime"*

```
# Convert date columns
rd['Created Date'] = pd.to_datetime(rd['Created Date'], format='%m/%d/%Y %I:%M:%S %p',
errors='coerce')
print(rd[['Created Date']].head())
```

```
# Convert date columns
rd['Closed Date'] = pd.to_datetime(rd['Closed Date'], format='%m/%d/%Y %I:%M:%S %p',
errors='coerce')
print(rd[['Closed Date']].head())
```

- At first, I convert columns into datetime of both created date and closed date.
- **rd** hold the dataset of our case- NYC service requests.
- **[‘Created Date’]** → It is columns of dataset which contains string values in the format (DD/MM/YYYY HH:MM:SS) that represents dates and times when requests were created.
- **[‘Closed Date’]** → It is columns of dataset which contains string values initially in the format (DD/MM/YYYY HH:MM:SS) that represents dates and times when requests were closed.
- **format='%m/%d/%Y %I:%M:%S %p', errors='coerce' :**

  **%I =** Twelve-hour formatted hours (01–12)

  **%M= Minute (00-59)**

  **%S= Second (00-59)**

  **%p = AM/PM**

  **errors= “coerce”:** Avoid crashing the code by replacing date with NaT **(Not a Time)**
- **pd** → alias of pandas library.
- **pd.to_datetime** → This is pandas functions which converts string into actual data time object.
- **(rd['Created Date']) , (rd['Closed Date'])** → Put the **updated date time** in the same column and same variable store ‘rd’ replacing the original column.

This allows you to perform **date-time operations** like extracting hour, day, week, month, or calculating durations.

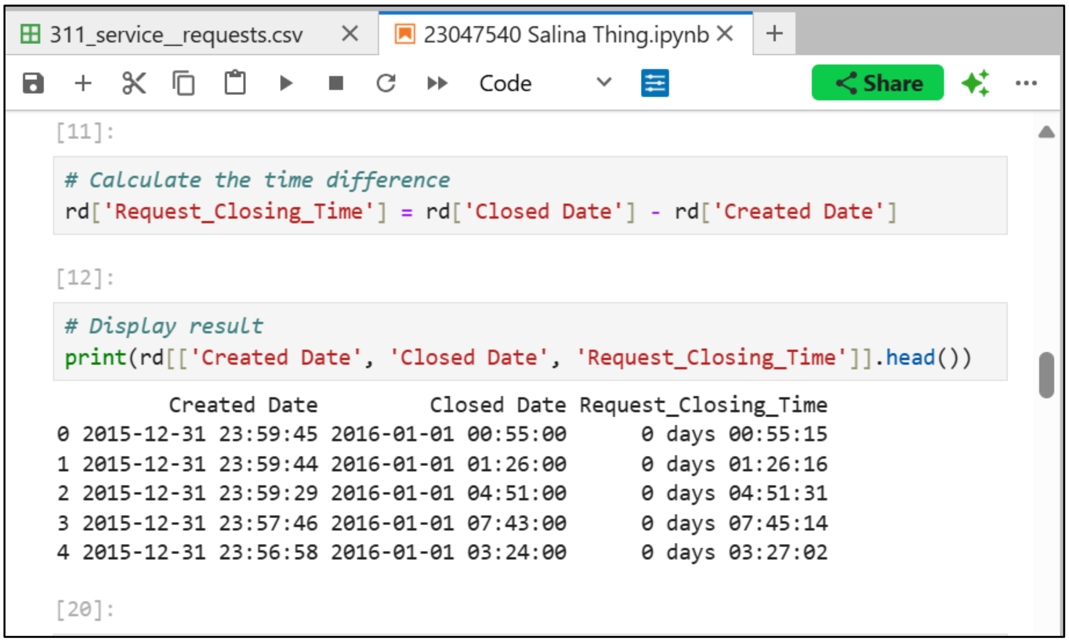



*Figure 7:Create and calculate "Request_closing_time" of datasets*

```
rd['Request_Closing_Time'] = rd['Closed Date'] - rd['Created Date']
```

After converting into date time, we create a new column "**Request_Closing_Time**" for finding the time difference between request creation and request closing. Each value is stored in '**rd' variable** respectively.

```
print(rd[['Created Date', 'Closed Date', 'Request_Closing_Time']].head())
```

- **[['Created Date', 'Closed Date', 'Request_Closing_Time']]:**

  This selects three columns only created date, closed date and request_closing_time. We use double **square big brackets [[…]]** to pass list of multiple columns together. Single square bracket is used for only one column to pass. Lastly, we print the columns and update in same column and stored in same variable(rd).

- **.head():** This is the method to display first five rows of selected columns **after conversion to datetime** which includes:

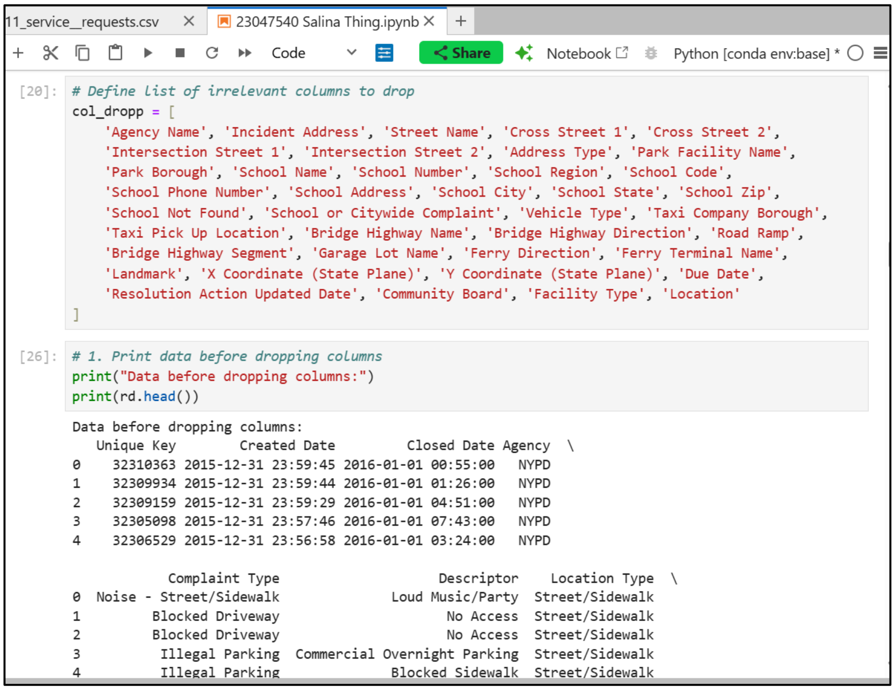
  When the complaint was created

  When it was closed

  How long it took to close

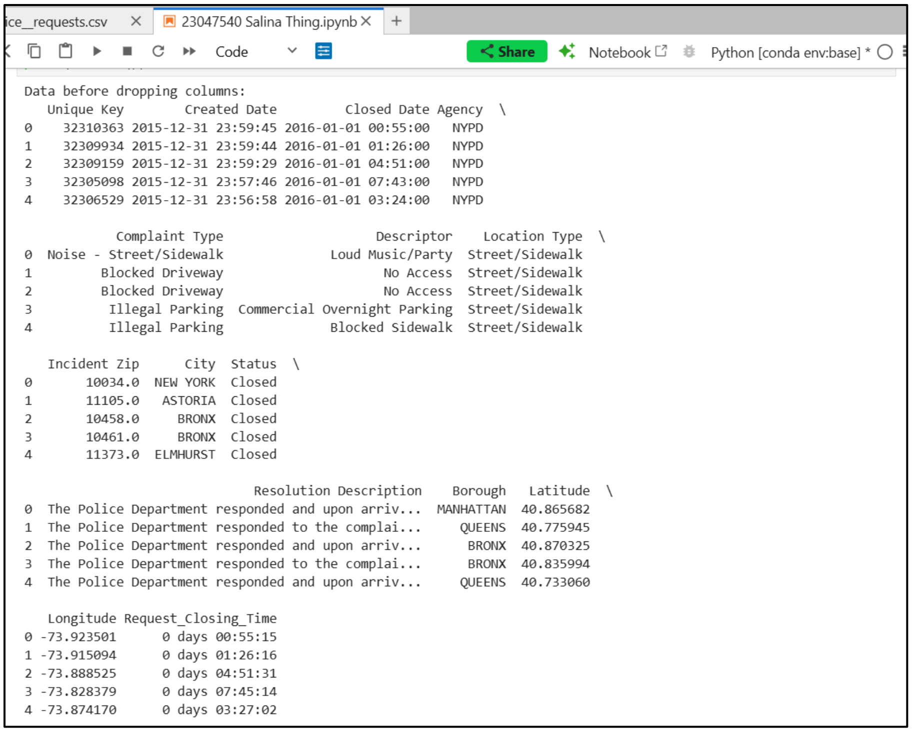
- Each entry is now a timestamp object in the standard format **YYYY-MM-DD HH:MM:SS.**

## 2.5. Drop irrelevant columns



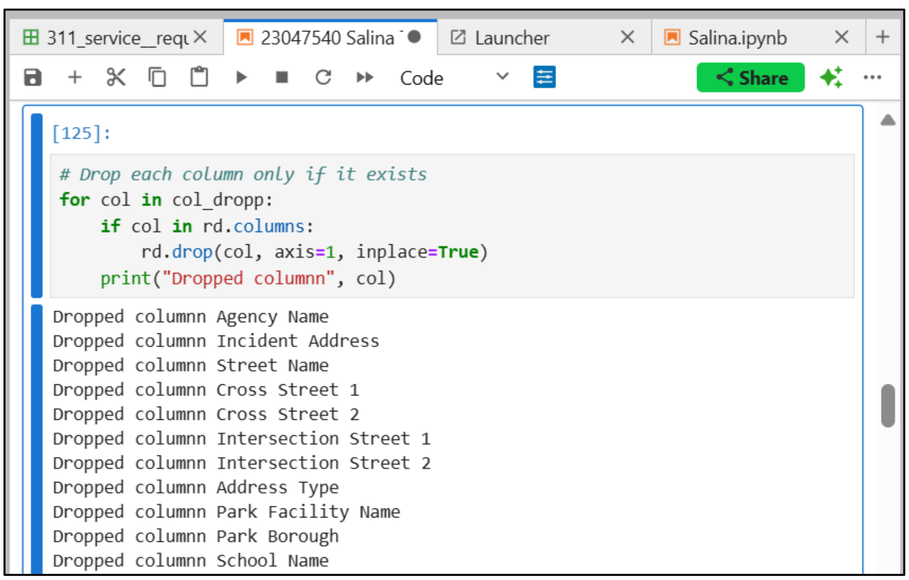

```
[20]: # Define list of irrelevant columns to drop
col_dropp = [
    'Agency Name', 'Incident Address', 'Street Name', 'Cross Street 1', 'Cross Street 2',
    'Intersection Street 1', 'Intersection Street 2', 'Address Type', 'Park Facility Name',
    'Park Borough', 'School Name', 'School Number', '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'
]

[26]: # 1. Print data before dropping columns
print("Data before dropping columns:")
print(rd.head())

Data before dropping columns:
   Unique Key         Created Date          Closed Date Agency  \
0    32310363 2015-12-31 23:59:45 2016-01-01 00:55:00   NYPD
1    32309934 2015-12-31 23:59:44 2016-01-01 01:26:00   NYPD
2    32309159 2015-12-31 23:59:29 2016-01-01 04:51:00   NYPD
3    32305098 2015-12-31 23:57:46 2016-01-01 07:43:00   NYPD
4    32306529 2015-12-31 23:56:58 2016-01-01 03:24:00   NYPD

           Complaint Type                    Descriptor    Location Type  \
0  Noise - Street/Sidewalk              Loud Music/Party  Street/Sidewalk
1         Blocked Driveway                     No Access  Street/Sidewalk
2         Blocked Driveway                     No Access  Street/Sidewalk
3          Illegal Parking  Commercial Overnight Parking  Street/Sidewalk
4          Illegal Parking              Blocked Sidewalk  Street/Sidewalk
```

*Figure 8: Dropping Irrelevant columns(i)*




```
Data before dropping columns:
   Unique Key        Created Date         Closed Date Agency  \
0    32310363 2015-12-31 23:59:45 2016-01-01 00:55:00   NYPD
1    32309934 2015-12-31 23:59:44 2016-01-01 01:26:00   NYPD
2    32309159 2015-12-31 23:59:29 2016-01-01 04:51:00   NYPD
3    32305098 2015-12-31 23:57:46 2016-01-01 07:43:00   NYPD
4    32306529 2015-12-31 23:56:58 2016-01-01 03:24:00   NYPD

            Complaint Type                    Descriptor    Location Type  \
0  Noise - Street/Sidewalk              Loud Music/Party  Street/Sidewalk
1         Blocked Driveway                     No Access  Street/Sidewalk
2         Blocked Driveway                     No Access  Street/Sidewalk
3          Illegal Parking  Commercial Overnight Parking  Street/Sidewalk
4          Illegal Parking              Blocked Sidewalk  Street/Sidewalk

   Incident Zip      City  Status  \
0       10034.0  NEW YORK  Closed
1       11105.0   ASTORIA  Closed
2       10458.0     BRONX  Closed
3       10461.0     BRONX  Closed
4       11373.0  ELMHURST  Closed

                              Resolution Description    Borough   Latitude  \
0  The Police Department responded and upon arriv...  MANHATTAN  40.865682
1  The Police Department responded to the complai...     QUEENS  40.775945
2  The Police Department responded and upon arriv...      BRONX  40.870325
3  The Police Department responded to the complai...      BRONX  40.835994
4  The Police Department responded and upon arriv...     QUEENS  40.733060

   Longitude Request_Closing_Time
0 -73.923501      0 days 00:55:15
1 -73.915094      0 days 01:26:16
2 -73.888525      0 days 04:51:31
3 -73.828379      0 days 07:45:14
4 -73.874170      0 days 03:27:02
```

*Figure 9: Before dropping columns, print the value of it*




```
[125]:
# Drop each column only if it exists
for col in col_dropp:
    if col in rd.columns:
        rd.drop(col, axis=1, inplace=True)
    print("Dropped columnn", col)
Dropped columnn Agency Name
Dropped columnn Incident Address
Dropped columnn Street Name
Dropped columnn Cross Street 1
Dropped columnn Cross Street 2
Dropped columnn Intersection Street 1
Dropped columnn Intersection Street 2
Dropped columnn Address Type
Dropped columnn Park Facility Name
Dropped columnn Park Borough
Dropped columnn School Name
```

*Figure 10: Dropping irrelevant columns using loop(ii)*

Here, I have listed the columns to drop as **"col_dropp"** then I have printed data before dropping the irrelevant columns. Then only use **for loop** to delete the given columns only if it exists otherwise no need to change, update to the original dataset**(inplace=True)** in rd and print the dropped columns.

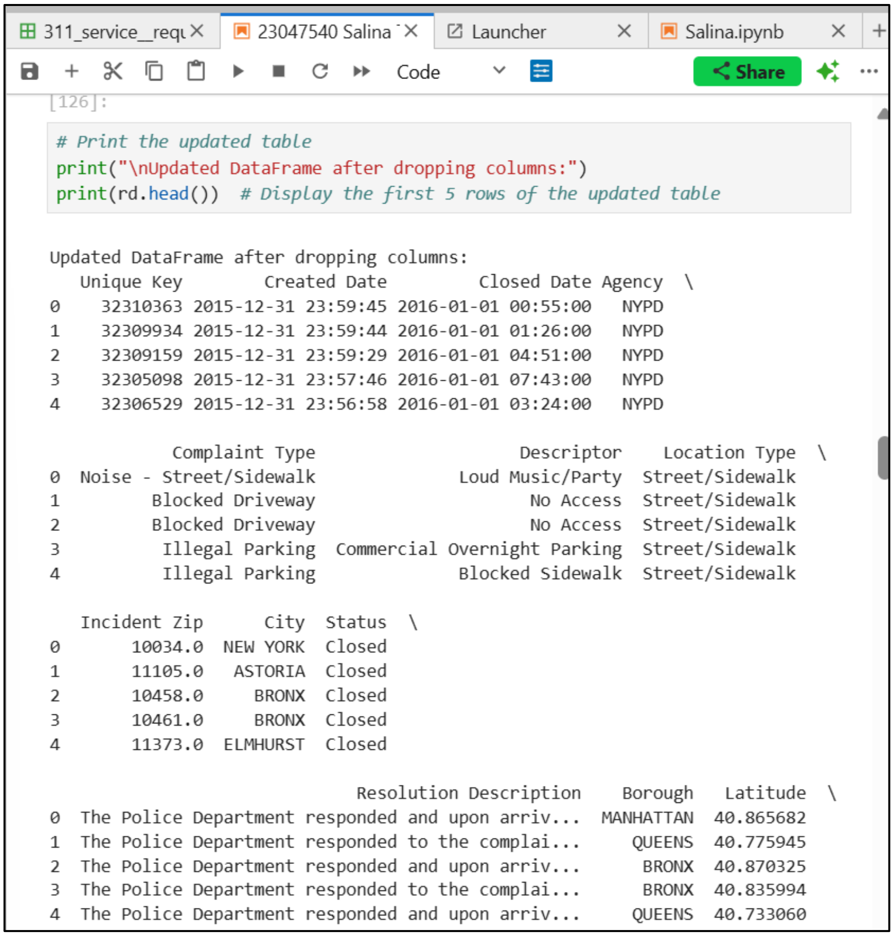
```
for col in col_dropp:
    if col in rd.columns:
        rd.drop(col, axis=1, inplace=True)
    print("Dropped column", col)
```

I have made **new variable "col"** in **for loop** in col_dropp and every time the condition meets the changes (current column) are updated to col. The condition checks if column name stored in col is present in rd or not. If yes, it goes to next time or skips to the next column in the loop.

- **rd.columns** include a list of all column names present in the DataFrame.
- **rd.drop(...)** is a method to remove columns from our DataFrame.
- **col** is the column name to remove.
- **axis=1** tells pandas to drop columns not rows.

- **inplace=True** means any changes that occurs automatically update to original dataset rd.
- **print** the dropped columns.

**Result**: All unnecessary columns are removed, and the dataset becomes more manageable.




```
# Print the updated table
print("\nUpdated DataFrame after dropping columns:")
print(rd.head())  # Display the first 5 rows of the updated table
```

```
Updated DataFrame after dropping columns:
   Unique Key        Created Date         Closed Date Agency  \
0    32310363 2015-12-31 23:59:45 2016-01-01 00:55:00   NYPD
1    32309934 2015-12-31 23:59:44 2016-01-01 01:26:00   NYPD
2    32309159 2015-12-31 23:59:29 2016-01-01 04:51:00   NYPD
3    32305098 2015-12-31 23:57:46 2016-01-01 07:43:00   NYPD
4    32306529 2015-12-31 23:56:58 2016-01-01 03:24:00   NYPD

            Complaint Type                    Descriptor    Location Type  \
0  Noise - Street/Sidewalk              Loud Music/Party  Street/Sidewalk
1         Blocked Driveway                     No Access  Street/Sidewalk
2         Blocked Driveway                     No Access  Street/Sidewalk
3          Illegal Parking  Commercial Overnight Parking  Street/Sidewalk
4          Illegal Parking              Blocked Sidewalk  Street/Sidewalk

   Incident Zip      City  Status  \
0       10034.0  NEW YORK  Closed
1       11105.0   ASTORIA  Closed
2       10458.0     BRONX  Closed
3       10461.0     BRONX  Closed
4       11373.0  ELMHURST  Closed

                               Resolution Description    Borough   Latitude  \
0  The Police Department responded and upon arriv...  MANHATTAN  40.865682
1  The Police Department responded to the complai...     QUEENS  40.775945
2  The Police Department responded and upon arriv...      BRONX  40.870325
3  The Police Department responded to the complai...      BRONX  40.835994
4  The Police Department responded and upon arriv...     QUEENS  40.733060
```

*Figure 11: Update table after dropping irrelevant columns*

```
print("\nUpdated DataFrame after dropping columns:")
print(rd.head())
```

The above code is after “droping the irrelevant columns” and update lists in the datasets(rd). To see if the code has updated or not. We use **rd.info()** to view the details of datasets in rd.

### 2.6. Remove missing values

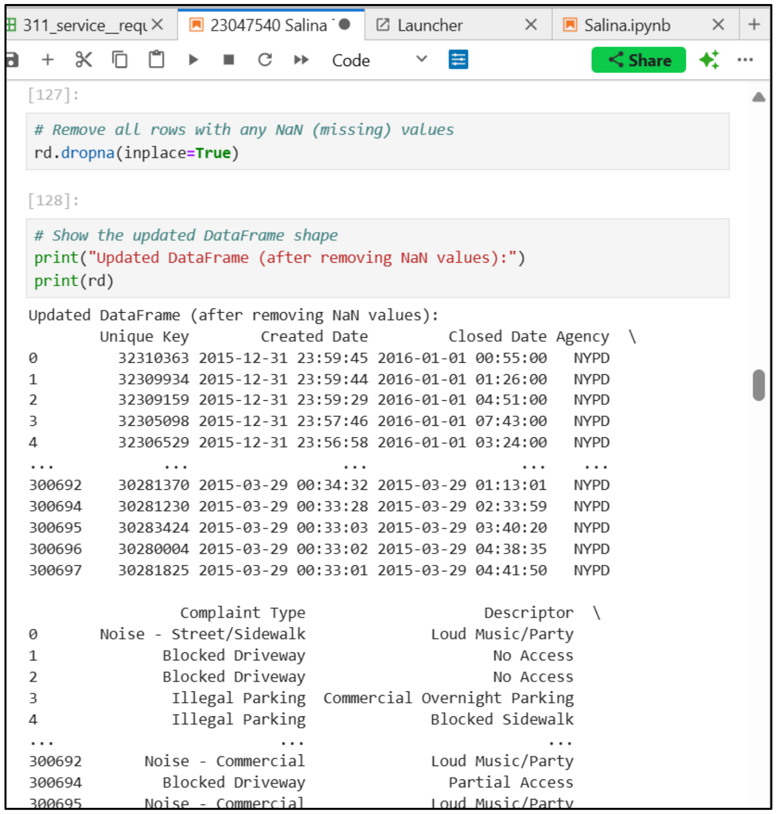
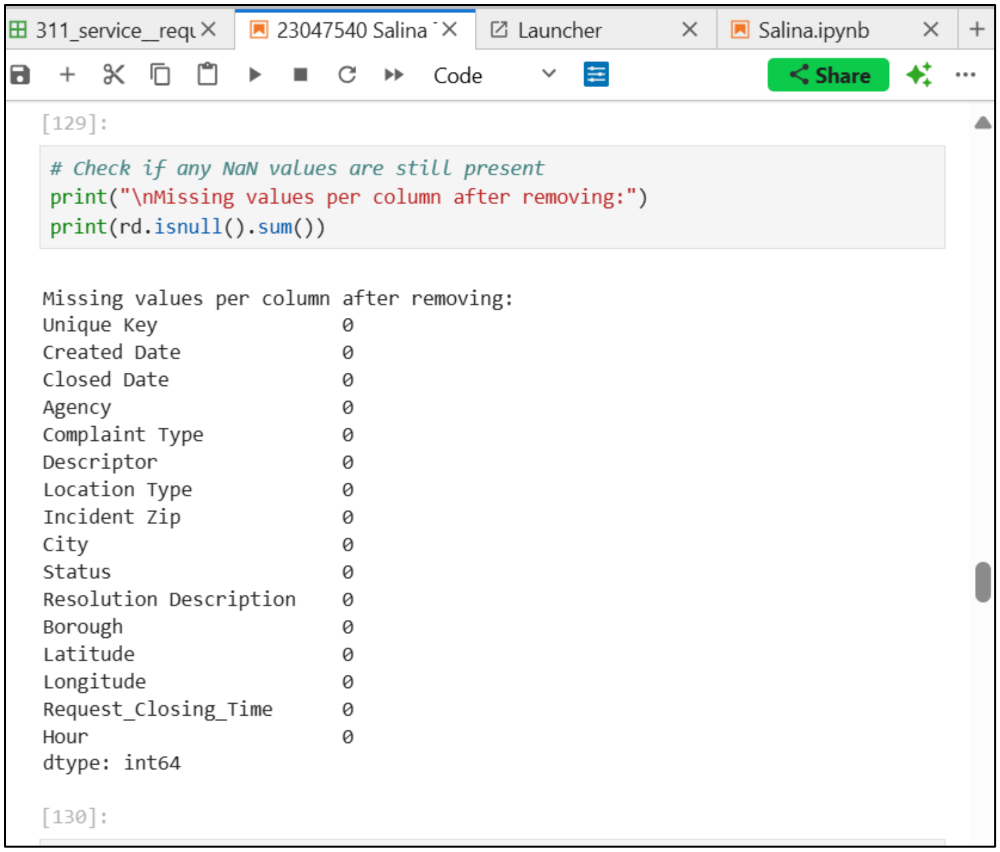



*Figure 12:Removing missing values*

```
rd.dropna(inplace=True)
```

```
print("Updated DataFrame (after removing NaN values):")
print(rd)
```

To remove missing values, I have used the code **“rd.dropna(inplace=True)”.** Here, **dropna() methods** removes the missing value and updates the changes in main datasets. Then, I show the updated dataframe by printing **“rd”.**



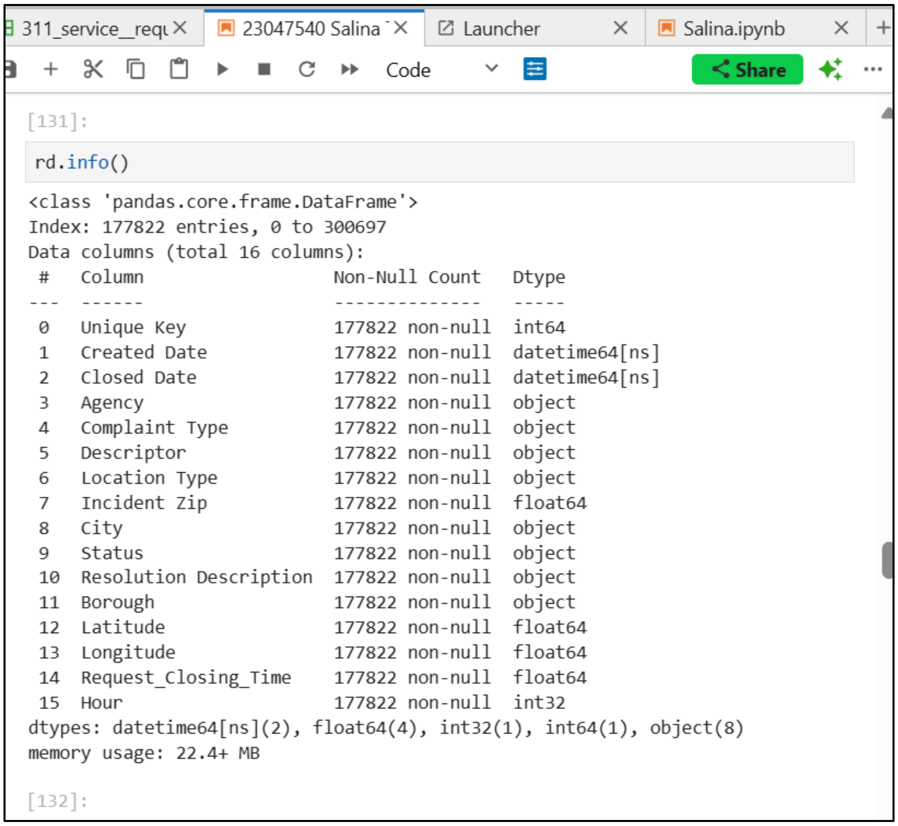

```
# Check if any NaN values are still present
print("\nMissing values per column after removing:")
print(rd.isnull().sum())
```

```
Missing values per column after removing:
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
Hour                      0
dtype: int64
```

*Figure 13: Checking if any Nan values are still present or not*

After removing missing values, I have checked if any missing values are left or not through the code **"print(rd.isnull().sum())"** if **yes** this code removes the missing value and print, if no direct print**.**

- **"is.null"** checks if there is any missing value"
- "**.sum()"** helps to add sum of each column if the condition is true.
- If all values are zero, it means the DataFrame is now fully clean (no missing data).
- **print(rd)** displays the **entire cleaned dataset.**
- **rd.head()** to show the first five rows.




```
[131]:
rd.info()

<class 'pandas.core.frame.DataFrame'>
Index: 177822 entries, 0 to 300697
Data columns (total 16 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   Unique Key              177822 non-null  int64
 1   Created Date            177822 non-null  datetime64[ns]
 2   Closed Date             177822 non-null  datetime64[ns]
 3   Agency                  177822 non-null  object
 4   Complaint Type          177822 non-null  object
 5   Descriptor              177822 non-null  object
 6   Location Type           177822 non-null  object
 7   Incident Zip            177822 non-null  float64
 8   City                    177822 non-null  object
 9   Status                  177822 non-null  object
 10  Resolution Description  177822 non-null  object
 11  Borough                 177822 non-null  object
 12  Latitude                177822 non-null  float64
 13  Longitude               177822 non-null  float64
 14  Request_Closing_Time    177822 non-null  float64
 15  Hour                    177822 non-null  int32
dtypes: datetime64[ns](2), float64(4), int32(1), int64(1), object(8)
memory usage: 22.4+ MB

[132]:
```

*Figure 14: info*

- **rd.info()** helps to show the information of whole datasets.

## 2.7. Show unique values in each column.

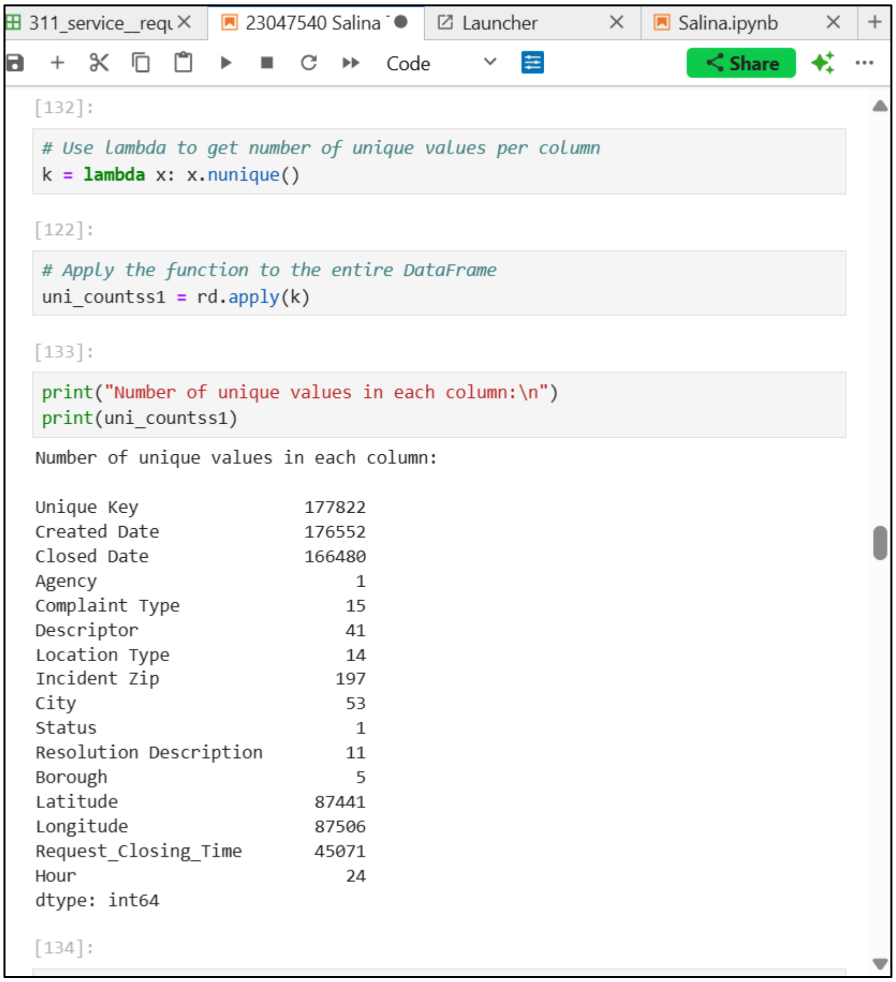



*Figure 15: Show unique values in each column(i)*

```
# Use lambda to get number of unique values per column
k = lambda x: x.nunique()
```

I have used lamda to get unique values per column.

- **x** represents each column passed to the function.
- **x.nunique()** includes the unique values of that specific column.

```
uni_countss1 = rd.apply(k)
print("Number of unique values in each column:\n")
print(uni_countss1)
```

- **.apply()** function is used to execute the function to every dataset column.
- It creates a **Series** where:

  Index = Column names

  Values = The quantity of distinct entries in every columns
- k counts each column's distinct values.
- Output each column's distinct values.

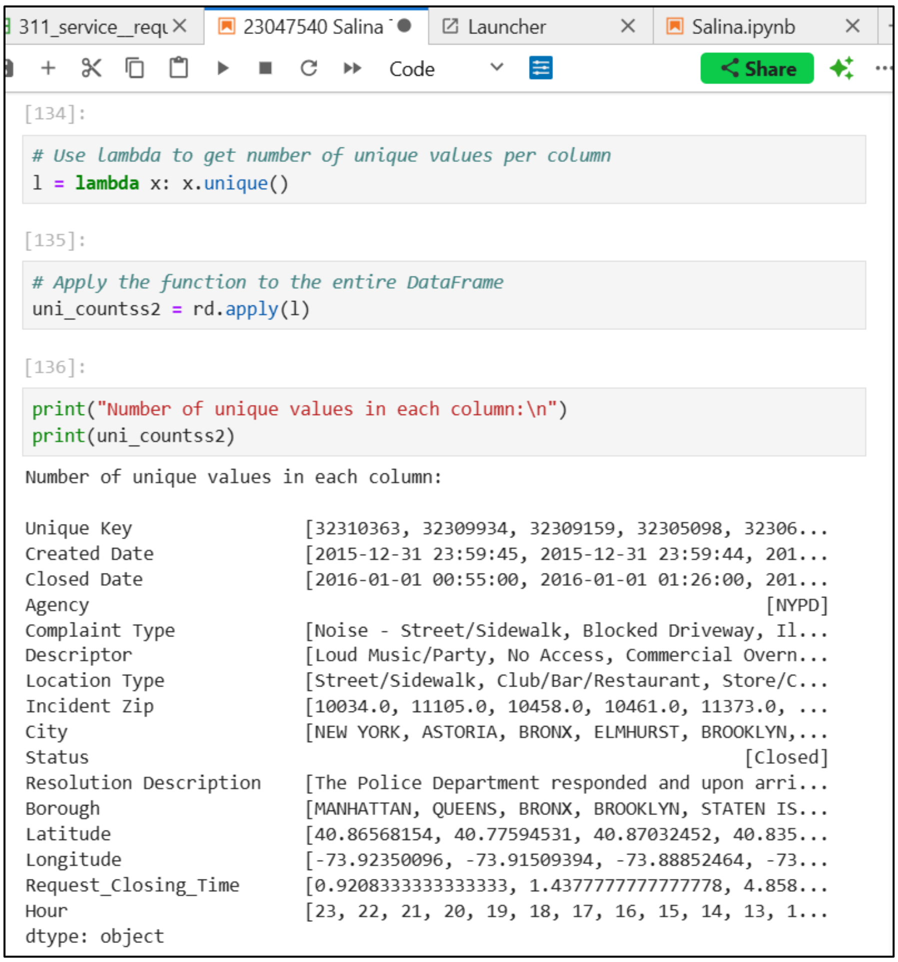


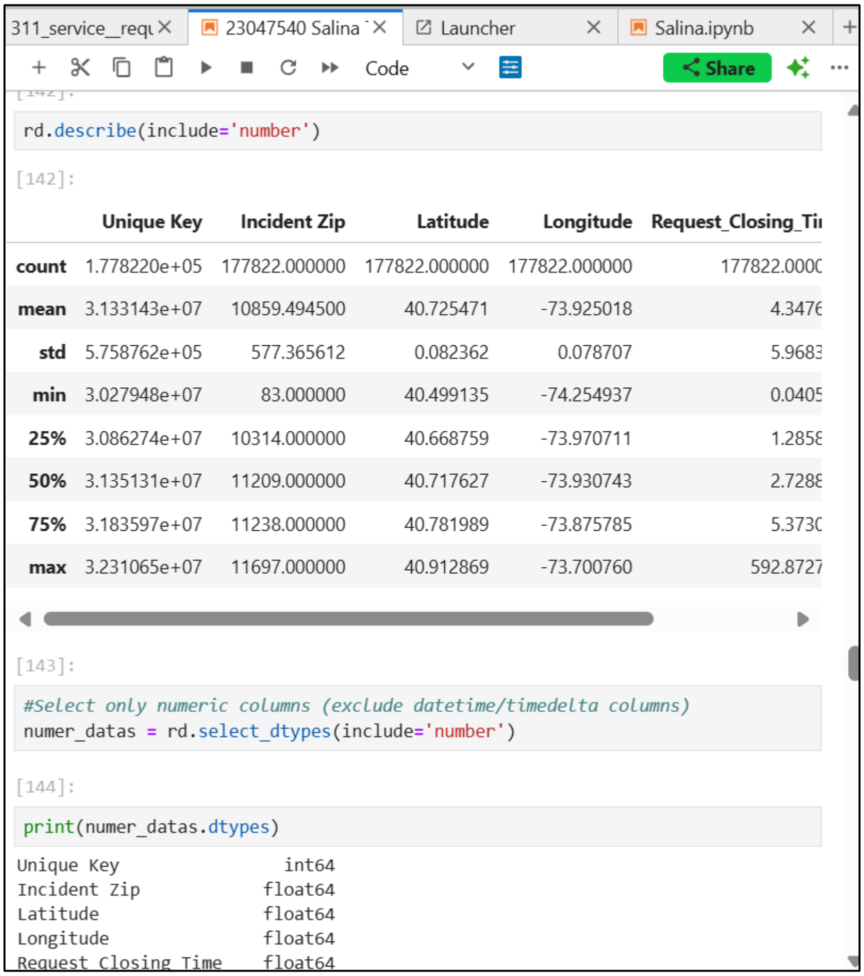

*Figure 16: Show unique values in each column(ii)*

Here, All are same like **nunique** and both have differences of as following:

**unique** write unique values in array list in a Series using **numpy.ndarray** where as **nunique** use to count unique values in **int.**

# 3. Data Analysis

## 3.1. To display the data frame's sum, mean, standard deviation, skewness, and kurtosis as summary statistics.




| | Unique Key | Incident Zip | Latitude | Longitude | Request_Closing_Ti |
|---|---|---|---|---|---|
| count | 1.778220e+05 | 177822.000000 | 177822.000000 | 177822.000000 | 177822.000C |
| mean | 3.133143e+07 | 10859.494500 | 40.725471 | -73.925018 | 4.3476 |
| std | 5.758762e+05 | 577.365612 | 0.082362 | 0.078707 | 5.9683 |
| min | 3.027948e+07 | 83.000000 | 40.499135 | -74.254937 | 0.0405 |
| 25% | 3.086274e+07 | 10314.000000 | 40.668759 | -73.970711 | 1.2858 |
| 50% | 3.135131e+07 | 11209.000000 | 40.717627 | -73.930743 | 2.7288 |
| 75% | 3.183597e+07 | 11238.000000 | 40.781989 | -73.875785 | 5.373C |
| max | 3.231065e+07 | 11697.000000 | 40.912869 | -73.700760 | 592.8727 |

*Figure 17: summary statistics(i)*

**Code:**

```
rd.describe(include='number')
```

At first, we display the specific columns which contains only “number” then through **.describe**, we display the descriptive statistics.

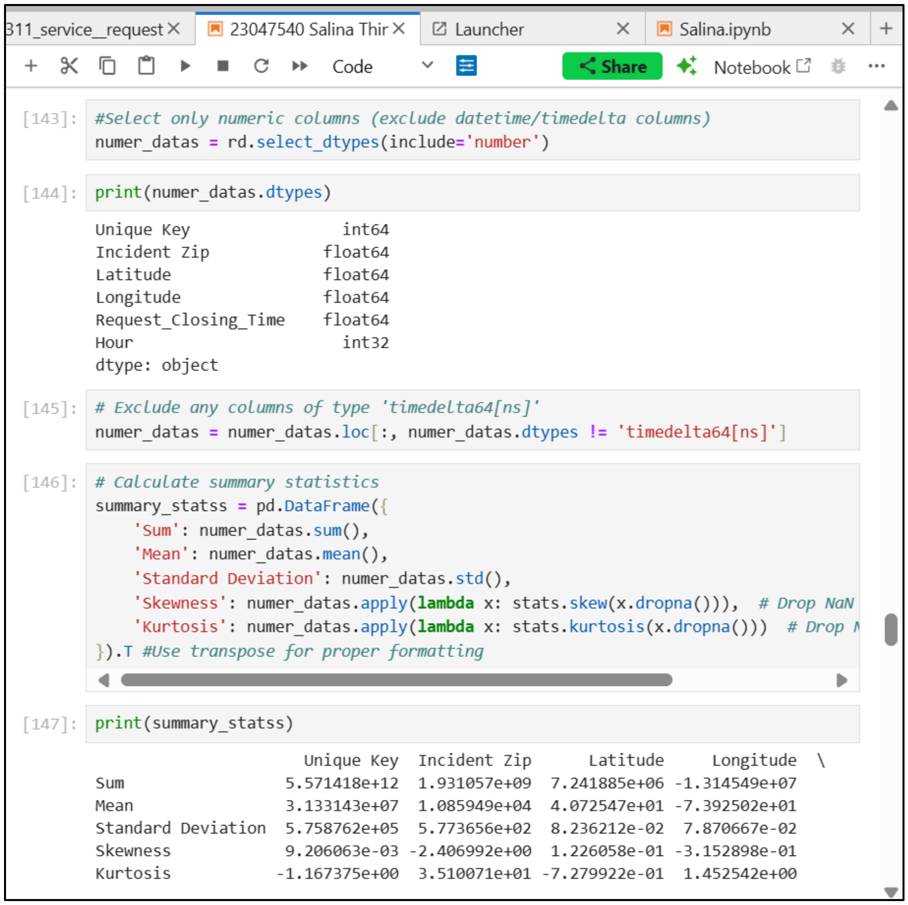



*Figure 18: Summary statistic(ii)*

```
numer_datas = rd.select_dtypes(include='number')
```

```
print(numer_datas.dtypes)
```

- **.select_dtype()** = methods which select the specific column based on the parameters inside it.
- **numer_datas** -new dataframe that contains only numeric columns data.
- It automatically excludes datetime or categorical columns.
- **print(numer_datas.dtype)=** This code print only numerical data with its datatype. Gives a quick overview of which columns are int64, float64, etc.

```
numer_datas = numer_datas.loc[:, numer_datas.dtypes != 'timedelta64[ns]']
```

- **numer_datas.loc[:, ..] =**It selects all the rows (choose the columns with conditions only numeric value and remove datetime. We need to remove datetime for statistical functions like skewness, mean, kurtosis,etc as we cannot calculate date time.

```
summary_statss = pd.DataFrame({
    'Sum': numer_datas.sum(),
    'Mean': numer_datas.mean(),
    'Standard Deviation': numer_datas.std(),
    'Skewness': numer_datas.apply(lambda x: stats.skew(x.dropna())),
    'Kurtosis': numer_datas.apply(lambda x: stats.kurtosis(x.dropna()))
}).T
```

- I have created **'summary_statss'** with multiple rows of sum(), mean(), Standard deviation, skewness and kurtosis.

  **numer_datas.sum()=** .sum() add all the values of each numeric column.

  **Mean=**average of each column

  **Standard deviation** = data varies from mean

  For **skewness** and **kutosis,** we import scipy.stats. Also, we also use dropna() and apply(). We apply .apply(), x.dropna() and lamba for skewness and kurtosis.

**.T** is transpose which converts rows into columns and viceversa.

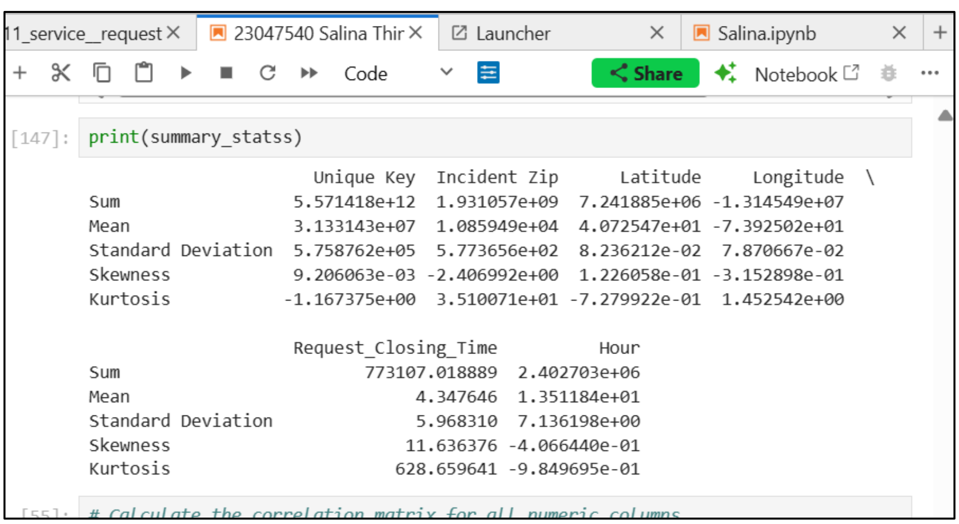



*Figure 19: Summary statistic(iii)*

```
print(summary_statss)
```

It prints the value inside the summary_stats.

**Result:** A table where:

Rows = Statistical metrics (Sum, Mean, etc.)

Columns = Each numeric column in your dataset

DataFrame using summary_statss, which includes:

**a) Sum**

Total value of any column.

**b) Mean**

The **average value**:

- Latitude ≈ 40.73 → consistent with NYC location.
- Longitude ≈ -73.94 → again, aligns with NYC geography.
- Incident Zip ≈ 10857 → a plausible ZIP code average in NYC.
- Unique Key average isn't meaningful; it’s just an ID.

**c) Standard Deviation**

Measures variability of data (National Library Of Medicine, 2025):

- Latitude and Longitude have small SDs (geographic locations are tightly clustered).
- Incident Zip has more variability (ZIP codes vary more across boroughs).
- Unique Key again isn't useful for interpretation.

**d) Skewness**

Measures **asymmetry** of the data (Turney, 2022):

- Incident Zip = 2.55: **high positive skew** (most values are low, few are high).
- Latitude = 1.23: **moderate positive skew**.
- Longitude = -0.13: **slightly negatively skewed**, close to symmetric.
- Skewness > 1 or < -1 typically signals significant skew.

**e) Kurtosis**

Measures the **tailedness** (Kenton, 2024):

- Incident Zip = 37.83: **extremely peaked distribution**.
- Latitude = -7.35: **flat distribution**.
- Kurtosis ≈ 0 is normal; large positive = sharp peak; large negative = flatter spread.

- ❖ This descriptive analysis helps identify data quality and distribution issues.
- ❖ **Incident Zip** is highly skewed and kurtotic → may need normalization or binning.
- ❖ **Latitude/Longitude** are consistent with real-world spatial data.
- ❖ **Unique Key** is **not suitable for statistical analysis** – drop it for modeling.

### 3.2. To calculate and show correlation of all the variables of the data frame.

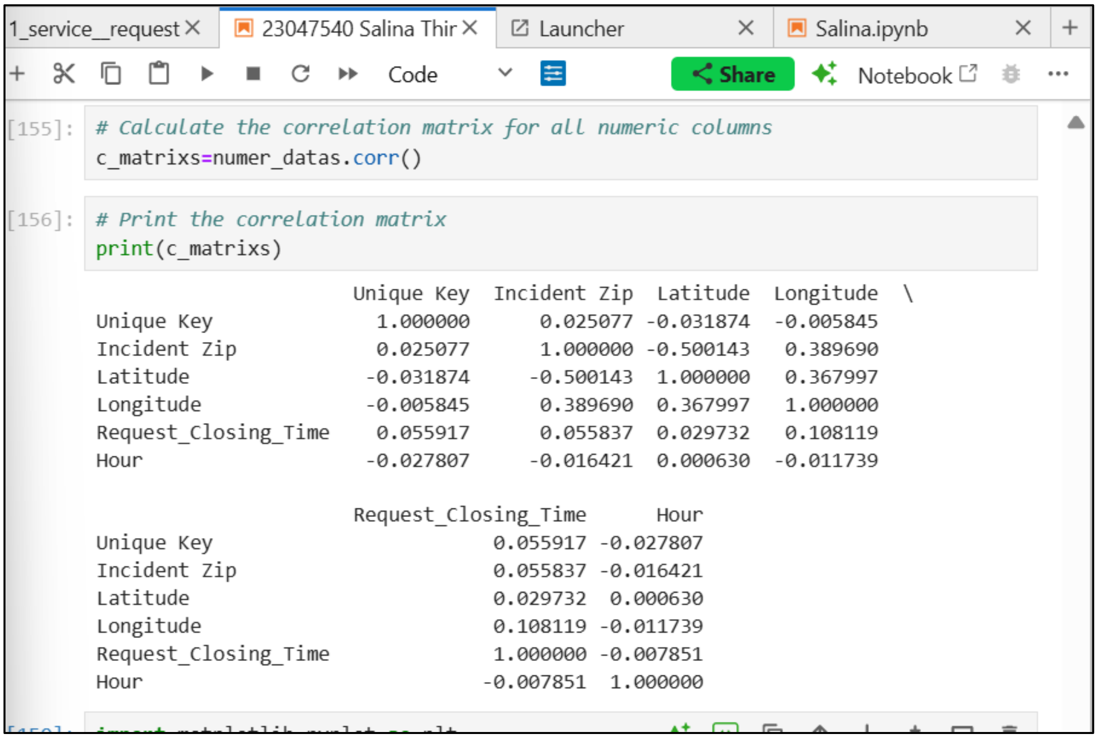



*Figure 20: Calculate correlation matrix for all the variables of datasets*

**Code:**

```
# Calculate the correlation matrix for all numeric columns
c_matrixs=numer_datas.corr()
# Print the correlation matrix
print(c_matrixs)
```

**corr() methods** calculates the relations between each column present in the datasets(rd). It values ranges from -1 to +1.

- ➔1 =perfect positive correlations
- ➔0= no correlation
- ➔-1=perfect negative correlations

Here, This methods directly ignores “non-numerics” columns (W3 schools, 2025).

```
c_matrixs=numer_datas.corr()
```

Here, we make variable “c_matrixs” to store data from ‘numer_datas.corr()’. As, **numer_datas** is also variable store of “summary_statics” which includes only numeric columns and calculates the relationship between two columns.

```
print(c_matrixs)
```

Then, we print the correlation matrix to display above variable (c_matrixs).

**Result:**

The matrix compares:

- Unique Key
- Incident Zip
- Latitude
- Longitude

```
                     Unique Key  Incident Zip      Latitude     Longitude  \
Sum                5.571418e+12  1.931057e+09  7.241885e+06 -1.314549e+07
Mean               3.133143e+07  1.085949e+04  4.072547e+01 -7.392502e+01
Standard Deviation 5.758762e+05  5.773656e+02  8.236212e-02  7.870667e-02
Skewness           9.206063e-03 -2.406992e+00  1.226058e-01 -3.152898e-01
Kurtosis          -1.167375e+00  3.510071e+01 -7.279922e-01  1.452542e+00

                   Request_Closing_Time          Hour
Sum                       773107.018889  2.402703e+06
Mean                           4.347646  1.351184e+01
Standard Deviation             5.968310  7.136198e+00
Skewness                      11.636376 -4.066440e-01
Kurtosis                     628.659641 -9.849695e-01
```

**Unique Key vs Itself:** The value is 1.0, which just means it's perfectly related to itself — that’s always true for any column.

**Unique Key vs Incident Zip:** The correlation is 0.025 — this is very close to 0, meaning the Unique Key has nothing to do with ZIP codes. It’s just an ID number.

**Incident Zip vs Latitude:** The value is -0.499. This means that as you move north (higher latitude), ZIP codes tend to get smaller. This likely reflects how NYC organizes ZIP codes from south to north.

**Incident Zip vs Longitude:** The value is 0.386, which shows a weak link — as ZIP codes increase, the location shifts a bit eastward.

**Latitude vs Longitude:** The value is 0.369, which means places that are further north also tend to be a bit more east. It just shows a mild relationship between nearby locations.

**Latitude vs Incident Zip:** Again, it's -0.499 — this repeats the earlier idea that going north means ZIP codes usually get smaller.

**Unique Key vs All Other Columns:** All these values are close to 0, which means the Unique Key doesn't affect or relate to location or ZIP code. It's just a unique label for each record.

**Final:**

Unique Key is an identifier → **drop from correlation analysis**

Incident Zip, Latitude, Longitude show **mild spatial correlation**

Strongest insight: **Zip code is moderately associated with geographic coordinates**, useful if you're modeling complaint density or mapping.

# 4. Data Exploration

## 4.1. Main four major insight through visualisation

### a) Insight 1: Most Common Complaint Types

**Visualisations:** Bar chart of complaint type frequencies.

**Insights:** Noise, Heating, and Illegal Parking are among the most frequent complaints.

311_service_request × | 23047540 Salina Thir | Launcher | Salina.ipynb

[157]:
```python
import matplotlib.pyplot as plt

# Get top 10 complaint types
top_complaintsss = rd['Complaint Type'].value_counts().head(10)
```

[158]:
```python
# Prepare data for plotting
complaint_namesss = top_complaintsss.index
complaint_countsss = top_complaintsss.values
```

[159]:
```python
# Plot using matplotlib
plt.figure(figsize=(8,4))
plt.barh(complaint_namesss, complaint_countsss, color='skyblue')
plt.title(" NYC's Top Ten Complaint Types")
plt.xlabel('Complaint Count')
plt.ylabel('Type of Complaints')
plt.grid(axis='x', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

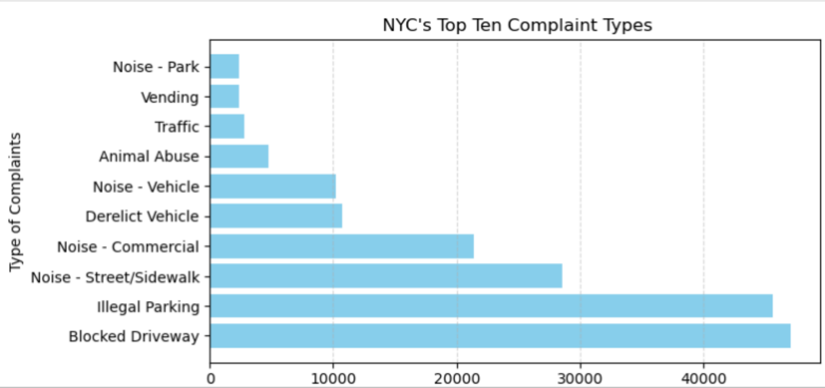



*Figure 21: Top most common complaint types(i)*

At first, I have imported **mat plot library** as plt (alias) specially of pyplot module which is used for creating plots and charts in python.

```
top_complaintsss = rd['Complaint Type'].value_counts().head(10)
```

**top_complaintsss=** new variable name which holds the top 10 complaint categories with their counts.

**rd['Complaint Type']=**Access “Complaint type” colums from rd dataframe

**. value_counts()=** It counts how many times each unique complaint type appears.

.**head(10)=** Display top 10 most frequent complaint types.

```
# Prepare data for plotting
complaint_namesss = top_complaintsss.index
complaint_countsss = top_complaintsss.values
```

**top_complaintsss.index:** Gets the names of the top 10 complaint types. **top_complaintsss.values**: Gets the corresponding counts for each complaint type.

These two arrays will be used to **label the bars** and define their **lengths**.

```
# Plot using matplotlib
plt.figure(figsize=(8,4))
```

Creates a new figure for the plot.

**figsize=(8,4):** Set a size of 8 inches wide and 4 inches tall, giving wide appearance.

```
plt.barh(complaint_namesss, complaint_countsss, color='skyblue')
```

**plt.barh():** Creates a **horizontal bar chart**.

**complaint_namesss:** Used as labels on the y-axis.

**complaint_countsss:** Determines the **length** of each bar (how many complaints).

**color='skyblue':** Sets the bar color to a light blue for better visualization.

```
plt.title(' NYC's Top Ten Complaint Type')
```

Adding a title to barchart.

```
plt.xlabel('Complaint Count')
plt.ylabel('Type of Complaints')
```

Adds labels to the x-axis and y-axis as mentioned in the code.

```
plt.grid(axis='x', linestyle='--', alpha=0.5)
```

Adds a light **dashed grid** along the x-axis to improve readability.

**alpha=0.5**: sets the grid transparency to 50%.

```
plt.tight_layout()
```

Automatically adjusts plot so that labels and tittle aren't cut off or without overlapping.

```
plt.show()
```

Displays the final plot.

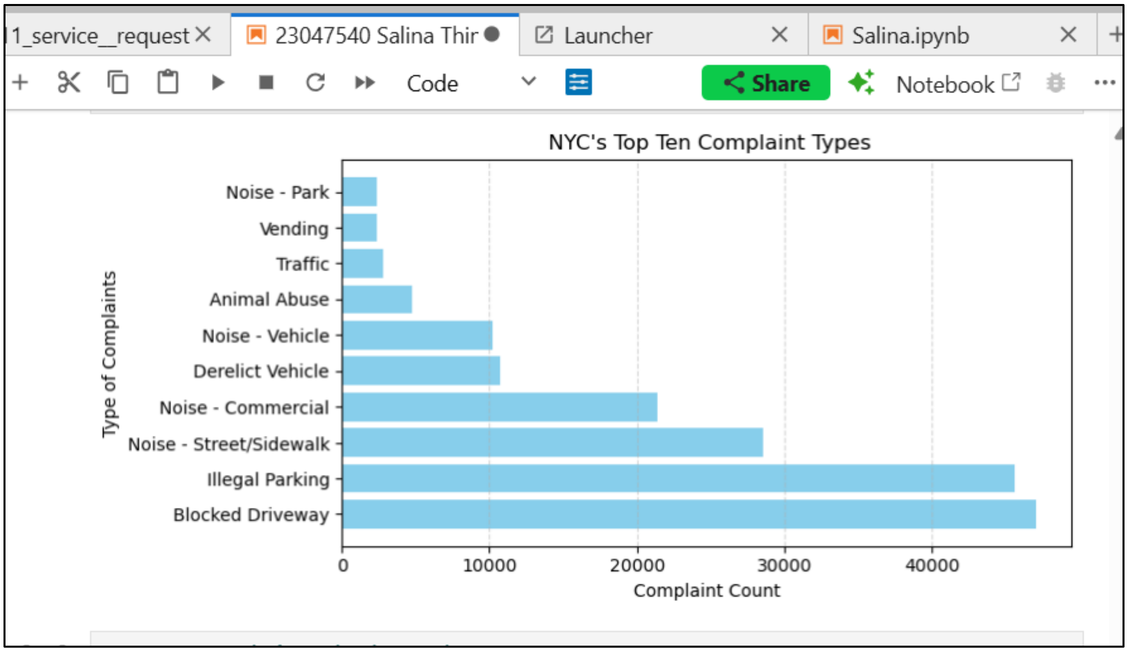



*Figure 22: Top most common complaints(ii)*

**Result:**

Here, the top 10 most common complaint types are represented in horizontal bar chart format:

- Each bar represents one complaint type.
- The **length of the bar** indicates how **frequently that complaint occurs**.
- Complaint types are sorted from most frequent to least (top to bottom).
- The top 10 most common complaint type is “Blocked Driveway” nearly 80,000 number of complaints.
- The least common complaint type is “Vending” with nearly 4000 number of complaints.

**b) Insight 2: Complaints Distribution Across Boroughs**

Visualisations: Pie chart or bar chart of complaints per borough.

Insights: Brooklyn and Manhattan receive the highest number of complaints.

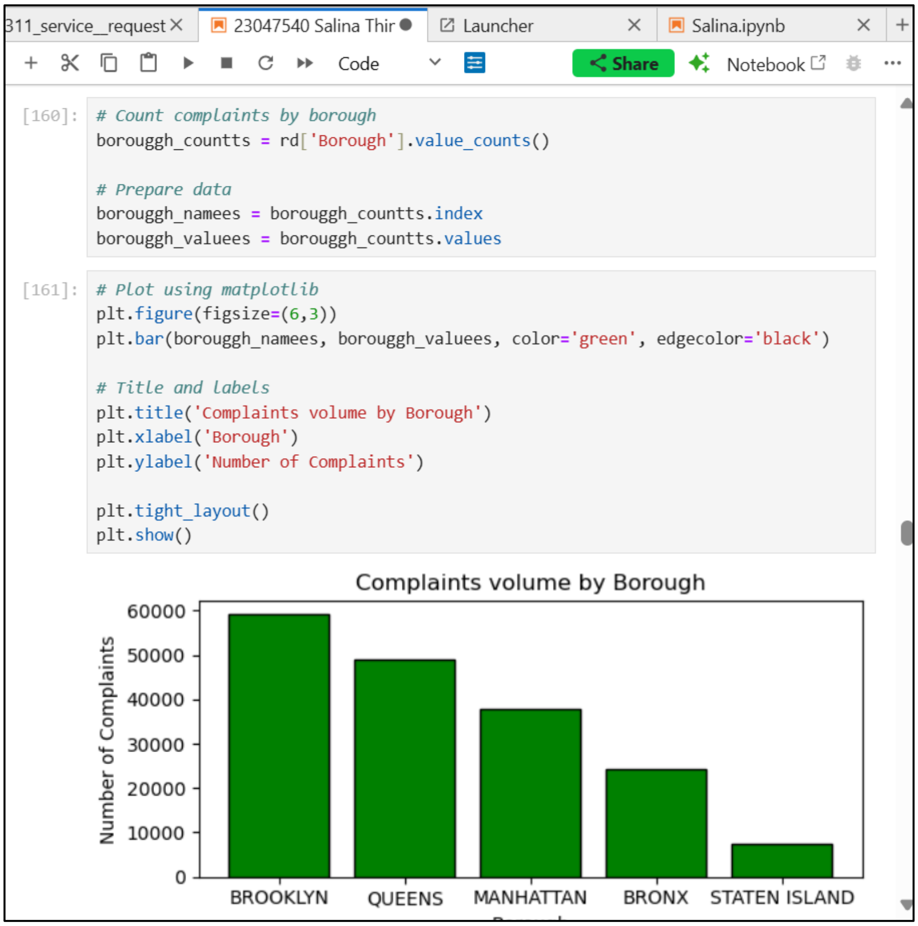



*Figure 23: Complaints Distribution Across Boroughs(i)*

```
# Count complaints by borough
borouggh_countts = rd['Borough'].value_counts()
```

**rd['Borough']:** Accesses the 'Borough' column from the DataFrame rd.

**.value_counts():** Counts how many complaints were recorded for each unique borough.

**borouggh_countts** now contains borough names (e.g., Manhattan, Brooklyn) and their corresponding number of complaints.

```
# Prepare data
borouggh_namees = borouggh_countts.index
borouggh_valuees = borouggh_countts.values
```

**borouggh_countts.index:** Extracts the borough names.

**borouggh_countts.values:** Extracts the number of complaints per borough.

```
# Plot using matplotlib
plt.figure(figsize=(6,3))
```

This line makes a new figure with a width of 6 inches and height of 3 inches in the plot.

This sets the overall size of the output chart.

```
plt.bar(borouggh_namees, borouggh_valuees, color='green', edgecolor='black')
```

Plots a vertical bar chart:

**borouggh_namees** go on the x-axis.

**borouggh_valuees** (complaint counts) go on the y-axis.

Bars are filled with green color.

**edgecolor='black'** outlines each bar with a black border for better visual distinction.

```
# Title and labels
plt.title(' Complaints volume by Borough ')
plt.xlabel('Borough')
plt.ylabel('Number of complaints')
```

Sets the chart **title** and **axis labels** to describe what the plot represents.

```
plt.tight_layout()
```

This lines actually adjusts the space to prevent overlaps or cut-off labels in the plot.

```
plt.show()
```

It displays the bar chart.

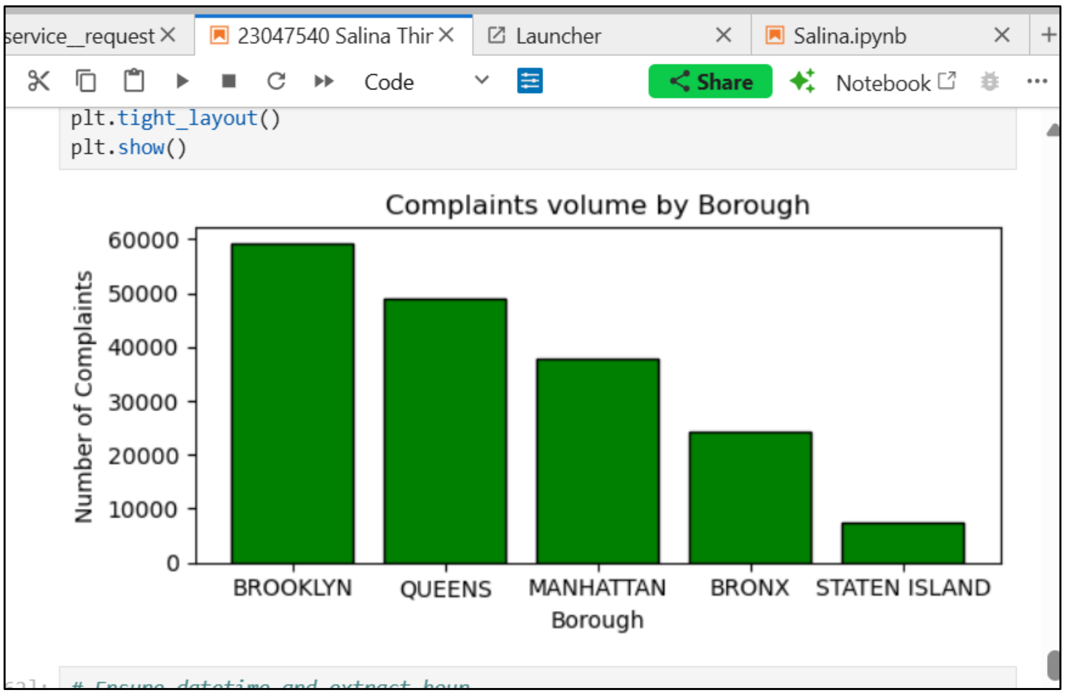



*Figure 24: Complaints Distribution Across Boroughs(ii)*

The chart shows a **comparison of complaint volume across NYC boroughs:**

- Each **bar height** shows the **total number of complaints** from that borough.
- The **tallest bar** shows the borough with the **highest number of complaints**.
- A shorter bar indicates **fewer complaints** from that borough.
- If **Brooklyn** has the highest bar, it suggests:

  Residents of Brooklyn submitted the most complaints.

- If **Staten Island** has the lowest bar:

  It had the fewest complaints logged.

  Possibly due to a smaller population or fewer service issues.

- **Complaint volume is uneven** across boroughs.
- Could correlate with **population density**, **service needs**, or **local government efficiency**.

## c) Insight 3: Complaint Volume by Hour of the Day

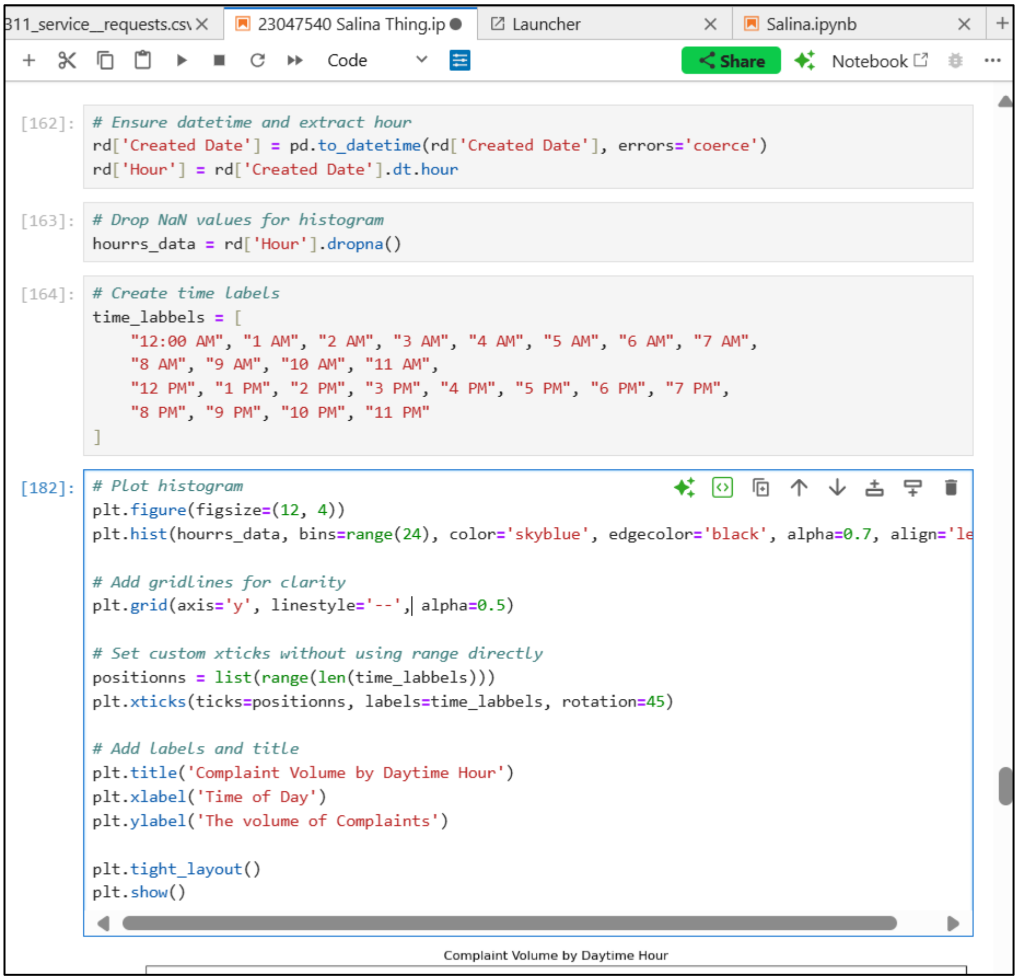



*Figure 25: Complaint Volume by Hour of the Day(i)*

```
# Ensure datetime and extract hour
rd['Created Date'] = pd.to_datetime(rd['Created Date'], errors='coerce')
rd['Hour'] = rd['Created Date'].dt.hour

# Drop NaN values for histogram
hourrs_data = rd['Hour'].dropna()

time_labbels = [
    "12:00 AM", "1 AM", "2 AM", "3 AM", "4 AM", "5 AM", "6 AM", "7 AM",
    "8 AM", "9 AM", "10 AM", "11 AM",
    "12 PM", "1 PM", "2 PM", "3 PM", "4 PM", "5 PM", "6 PM", "7 PM",
    "8 PM", "9 PM", "10 PM", "11 PM"
]
```

- This line **converts the 'Created Date' column** into proper datetime format.
- **errors='coerce'**: If any value cannot be converted, it becomes NaT (Not a Time), avoiding errors.
- This step is important because datetime operations require proper formatting.

  Extracts the **hour (0 to 23)** from each timestamp in 'Created Date'.

  Stores the result in a new column called 'Hour'.
- This is used to understand **what time of day** complaints are most frequently submitted.

  Some rows may have NaT in 'Created Date', which leads to NaN in 'Hour'.

  This line removes those rows to clean the data before plotting.
- **Time_labbels** : defines custom human time of each hour to improve chart readability.

```
# Plot histogram using matplotlib
plt.figure(figsize=(12,4))
```

- Creates a **new figure** for the plot.
- figsize=(12, 4) gives it a **wide and short appearance**, suitable for time-based data.

```
plt.hist(hourrs_data, bins=range (24), color='skyblue', edgecolor='black', alpha=0.7, align='left')
```

- Plots a **histogram** of the hours when complaints were created.
- hourrs_data: the list of all hours (0–23).
- bins=range(25): Creates 24 bins (one for each hour of the day).
- edgecolor='black': Adding a black border to each bar.
- color='skyblue': Sets the bar color with skyblue.
- align='left': Aligns each bar to the left of the bin value for clean labeling.

```
# Set custom xticks without using range directly
positionns = list(range(len(time_labbels)))
plt.xticks(ticks=positionns, labels=time_labbels, rotation=45)
```

This sets the x-tick positions to 0 through 23 using list, applies time_labbels as xustom x-axis labels and for better readability rotates in 45 degree.

```
# Title and axis labels
plt.title('Complaint Volume by Daytime Hour')
plt.xlabel('Time of Day')
plt.ylabel('The volume of Complaints')
```

Adds a clear **title** and **axis labels** for understanding what the chart represents.

```
plt.tight_layout()
plt.show()
```

Adjusts spacing so that labels don’t overlap.

**Displays** the final histogram.

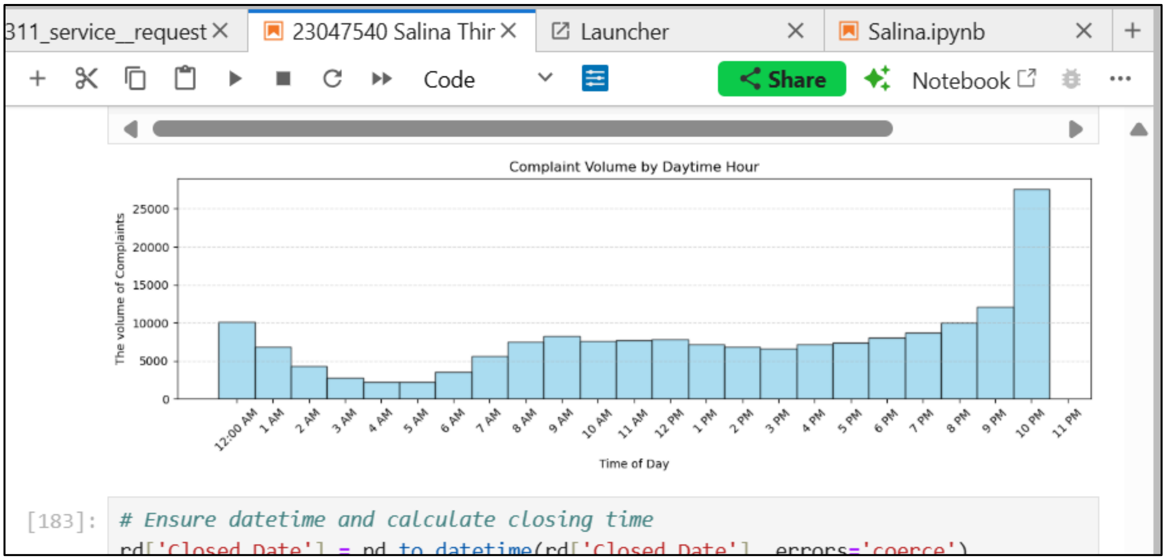



*Figure 26:Complaint Volume by Hour of the Day(ii)*

- A histogram with 24 bars — one for each **hour of the day**.
- Each bar shows how many complaints were made during that hour across the dataset.
- Ta**llest bars** show the hours when **most complaints were created**.
- Highest Complaint Volume: The number of complaints peaks at around 10 PM, surpassing 25,000. Later in the evening, this can be a sign of an increase in noise complaints or disturbances.
- Early Morning (2 AM–5 AM): Since most people are asleep and fewer problems arise, the early morning hours have the lowest complaint volumes.
- Gradual Increase from 6 AM: As the city awakens, the number of complaints begins to steadily increase at 6 AM.
- Plateau Midday (10 AM–4 PM): During business hours, the volume is comparatively constant, with a moderate number of complaints.
- Evening Spike (6 PM–10 PM): Complaints are clearly on the rise in the evening, most likely as a result of residential problems like parking, noise, and public disruptions.

## d) Insight 4: Response Time Distribution

Visualization: Bar chart of average 'Request_Closing_Time' by complaint type.

Insight: Certain complaints like Water System or Electric tend to take significantly longer to resolve.

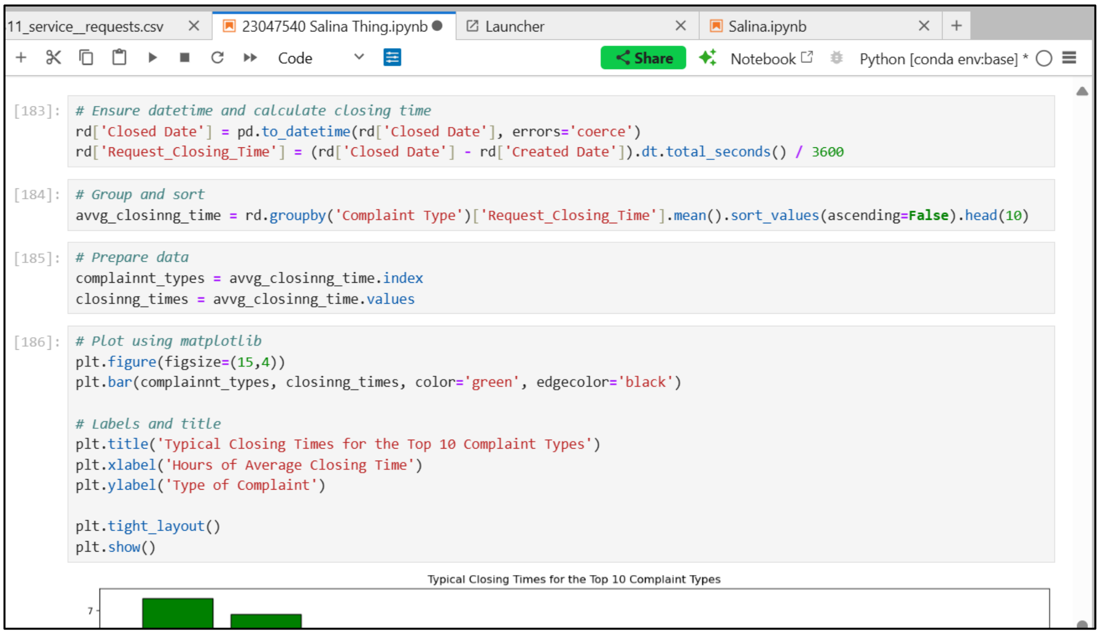

```
[183]: # Ensure datetime and calculate closing time
rd['Closed Date'] = pd.to_datetime(rd['Closed Date'], errors='coerce')
rd['Request_Closing_Time'] = (rd['Closed Date'] - rd['Created Date']).dt.total_seconds() / 3600

[184]: # Group and sort
avvg_closinng_time = rd.groupby('Complaint Type')['Request_Closing_Time'].mean().sort_values(ascending=False).head(10)

[185]: # Prepare data
complainnt_types = avvg_closinng_time.index
closinng_times = avvg_closinng_time.values

[186]: # Plot using matplotlib
plt.figure(figsize=(15,4))
plt.bar(complainnt_types, closinng_times, color='green', edgecolor='black')

# Labels and title
plt.title('Typical Closing Times for the Top 10 Complaint Types')
plt.xlabel('Hours of Average Closing Time')
plt.ylabel('Type of Complaint')

plt.tight_layout()
plt.show()
```


*Figure 27: Response Time Distribution(i)*

```
rd['Closed Date'] = pd.to_datetime(rd['Closed Date'], errors='coerce')
rd['Request_Closing_Time'] = (rd['Closed Date'] - rd['Created Date']).dt.total_seconds() / 3600
```

- Converts the 'Closed Date' column to datetime format.
- Calculates the **time it took to resolve each complaint**, in **hours**.
  - Subtracts 'Created Date' from 'Closed Date'.
  - .dt.total_seconds() gets the difference in seconds.
  - Dividing by 3600 converts it to hours.
- Stores the result in a new column 'Request_Closing_Time'.

```
avvg_closing_time = rd.groupby('Complaint
Type')['Request_Closing_Time'].mean().sort_values(ascending=False).head(10)
```

Groups the data by 'Complaint Type'.

Calculates the **average closing time (in hours)** for each complaint type.

Sorts it in **descending order**, so the types with the **longest average resolution time** come first.

Takes the **top 10** complaint types with the highest average closing time.

```
complainnt_types = avvg_closinng_time.index
closinng_times = avvg_closinng_time.values
```

**complainnt_types:** The names of the 10 complaint types.

**closinng_times:** Their corresponding average closing times (in hours).

```
plt.figure(figsize=(15,4))
plt.bar(complainnt_types, closinng_times, color='green', edgecolor='black')
```

Creates a horizontal plot canvas of size 15x4.

Draws a **bar chart** with:

- Complaint types on the x-axis.
- Average resolution times on the y-axis.
- Bars colored green with black edges for clarity.

```
plt.title('Typical Closing Times for the Top 10 Complaint Types')
plt.xlabel('Hours of Average Closing Time')
plt.ylabel('Type of Complaint')
```

Adds a title and axis labels to explain the data.

```
plt.tight_layout()
plt.show()
```

Adjusts layout for better spacing and displays the plot.

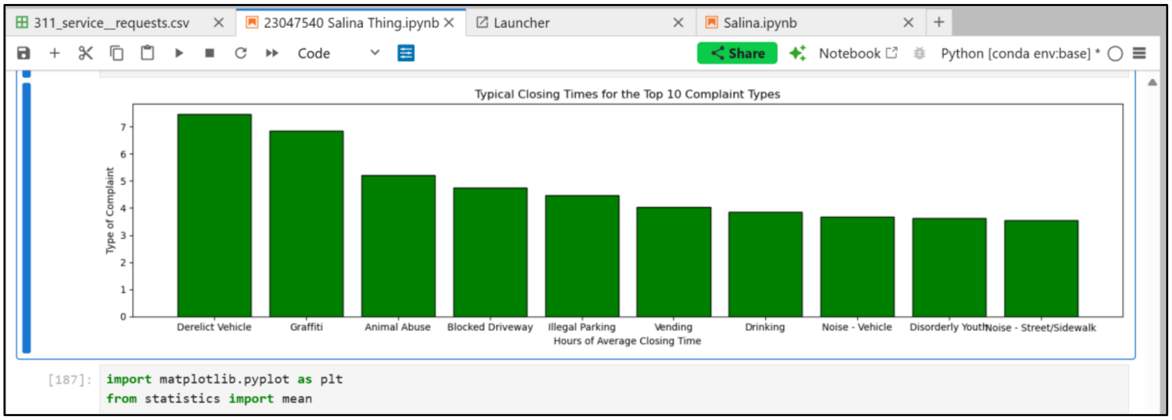



*Figure 28: Response Time Distribution(ii)*

- A bar chart with the **10 complaint types** that take the **longest time to close** on average.

  X-axis: The **complaint type** (e.g., "Water System", "General Construction").

  Y-axis: **Average closing time in hours**.

  - The **longest bars** indicate **slowest response or resolution times**.
  - These might be complex or lower-priority issues.
- Helps **identify bottlenecks** in the city's service system.
- Guides agencies on where to **allocate more resources** or streamline processes.
- Can improve **citizen satisfaction** by reducing delay in high-closing-time complaints.

### 4.2. Complaint Types by Average Request_Closing_Time Across Locations

**Process:**

- Calculate the average request closing time (in hours or days) for each complaint type grouped by location (e.g., borough or precinct).
- Use groupby(['Complaint Type', 'Borough'])['Request_Closing_Time'].mean() in Pandas.

**Visualization:**

- Recommended Graph: Heatmap or grouped bar chart.
- This allows visual comparison across boroughs for each complaint type.

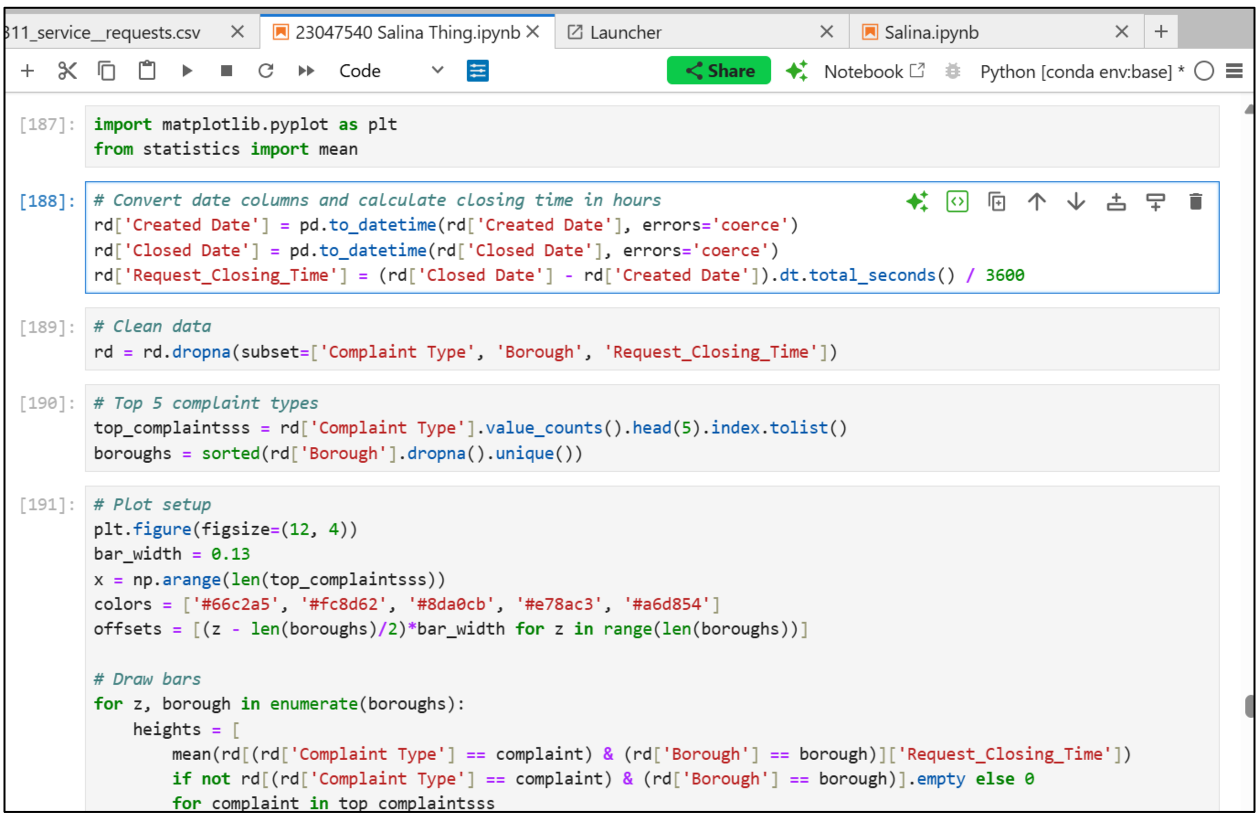



*Figure 29: Complaint Types by Average Request_Closing_Time Across Locations(i)*

```python
top_complaintsss = rd['Complaint Type'].value_counts().head(5).index.tolist()
boroughs = sorted(rd['Borough'].dropna().unique())
```

```python
# Plot setup
plt.figure(figsize=(12, 4))
bar_width = 0.13
x = np.arange(len(top_complaintsss))
colors = ['#66c2a5', '#fc8d62', '#8da0cb', '#e78ac3', '#a6d854']
offsets = [(z - len(boroughs)/2)*bar_width for z in range(len(boroughs))]

# Draw bars
for z, borough in enumerate(boroughs):
    heights = [
        mean(rd[(rd['Complaint Type'] == complaint) & (rd['Borough'] == borough)]['Request_Closing_Time'])
        if not rd[(rd['Complaint Type'] == complaint) & (rd['Borough'] == borough)].empty else 0
        for complaint in top_complaintsss
    ]
    plt.bar(x + offsets[z], heights, width=bar_width, label=borough,
            color=colors[z % len(colors)], edgecolor='black', alpha=0.8)

# Final plot settings
plt.title('The Top 5 Complaint Types by Borough and Their Average Closing Time (in hours)')
plt.xlabel('Type of Complaints')
plt.ylabel('Hours of Average Closing Time')
plt.xticks(x, top_complaintsss)
plt.legend(title='Borough')
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

*Figure 30:Complaint Types by Average Request_Closing_Time Across Locations(ii)*

```python
rd['Created Date'] = pd.to_datetime(rd['Created Date'], errors='coerce')
rd['Closed Date'] = pd.to_datetime(rd['Closed Date'], errors='coerce')
rd['Request_Closing_Time'] = (rd['Closed Date'] - rd['Created Date']).dt.total_seconds() / 3600
```

Converts 'Created Date' and 'Closed Date' to datetime format.

Calculates the **total time (in hours)** between when the request was created and when it was closed.

Saves this in a new column 'Request_Closing_Time'.

```python
rd = rd.dropna(subset=['Complaint Type', 'Borough', 'Request_Closing_Time'])
```

Removes rows where any of these fields are missing:

- 'Complaint Type'
- 'Borough'
- 'Request_Closing_Time'

Ensures only valid, complete data is used in the plot

```
top_complaintsss = rd['Complaint Type'].value_counts().head(5).index.tolist()
boroughs = sorted(rd['Borough'].dropna().unique())
```

Gets the **top 5 most frequent complaint types**.

Collects a **sorted list of all boroughs** (e.g., Bronx, Brooklyn, etc.).

```
plt.figure(figsize=(12, 4))
bar_width = 0.13
x = np.arange(len(top_complaints))
```

Initializes the plot with a size of **12 inches wide by 4 inches high**.

Sets each bar's width to 0.13 (narrow so grouped bars don’t overlap).

x contains positions for the **x-axis ticks** (one for each complaint type).

```
colors = ['#66c2a5', '#fc8d62', '#8da0cb', '#e78ac3', '#a6d854']
offsets = [(z - len(boroughs)/2)*bar_width for z in range(len(boroughs))]
```

Assigns a list of visually distinct **colors** for different boroughs.

offsets: Shifts each borough’s bar slightly left or right so bars for the same complaint type don’t overlap — they appear **side-by-side**.

```
for z, borough in enumerate(boroughs):
    heights = [
        mean(rd[(rd['Complaint Type'] == complaint) & (rd['Borough'] ==
borough)]['Request_Closing_Time'])
        if not rd[(rd['Complaint Type'] == complaint) & (rd['Borough'] == borough)].empty
else 0
        for complaint in top_complaints
    ]
    plt.bar(x + offsets[z], heights, width=bar_width, label=borough,
            color=colors[z % len(colors)], edgecolor='black', alpha=0.8)
```

For each **borough**, it:

- Loops through all top complaint types.
- Filters the dataset to get average closing time **specific to that complaint and borough**.
- Adds a bar at the correct horizontal offset.

Bars are color-coded by borough and aligned for comparison.

```
plt.title('The Top 5 Complaint Types by Borough and Their Average Closing Time (in hours)')
plt.xlabel('Type of Complaints')
plt.ylabel('Hours of Average Closing Time')
plt.xticks(x, top_complaintsss)
plt.legend(title='Borough')
plt.tight_layout()
plt.show()
```

Adds a title, axis labels, and legend.

plt.xticks() labels the x-axis with complaint names.

plt.tight_layout() avoids overlaps.

Finally, the chart is displayed

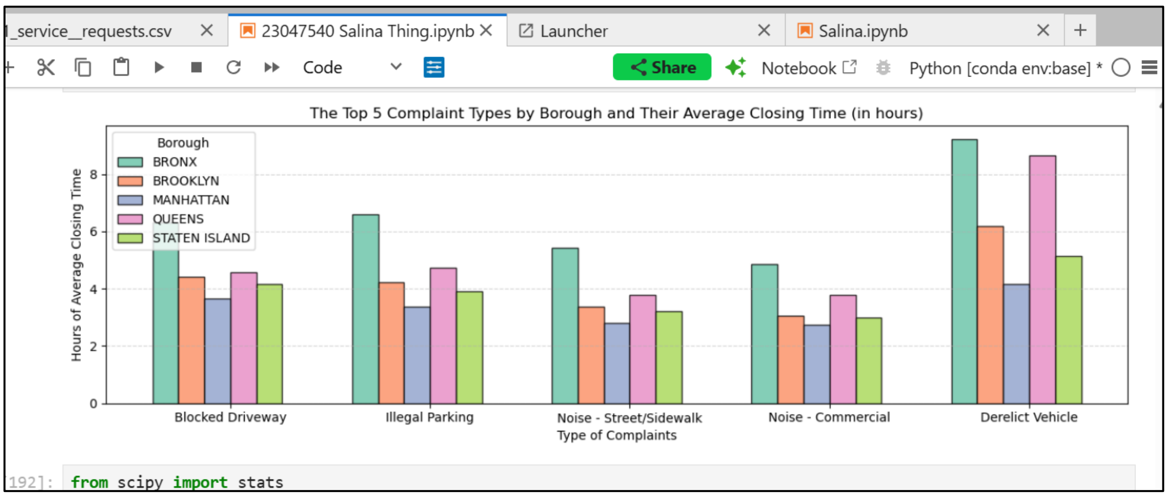



*Figure 31: Complaint Types by Average Request_Closing_Time Across Locations(iii)*

As a result, I get a **grouped bar chart**:

- **X-axis**: Top 5 complaint types.
- **Y-axis**: Average time (in hours) to close those complaints.
- **Bars**: One for each borough, color-coded and grouped by complaint type.

  - If “Noise - Residential” takes **longer in Queens** than in Brooklyn → potential inefficiencies.
  - Some boroughs might **consistently have longer bars** → indicates slower response/resolution times.
  - **Balanced bars** across boroughs mean service is more consistent.
  - Reveals **inequality or inconsistency** in city response times by borough.
  - Help public service teams **prioritize borough-specific process improvements**.

# 5. Statistical Testing

## 5.1. Test 1 Objective

To determine whether **average request closing times** are **significantly different** between various **complaint types**.

### a) Null and Alternate Hypotheses

For each pairwise t-test comparison between two complaint types:

- **Null Hypothesis ($H_0$):**
  Null hypothesis is a type of statistical hypothesis that is the default position (Newcastle University, 2025).

  The **average closing times** for the two complaint types are **equal**.
  (No significant difference.)

- **Alternate Hypothesis ($H_1$):**

  The alternative hypothesis is the hypothesis that includes sample observations which are influenced by a non-random cause (Newcastle University, 2025).

  The **average closing times** for the two complaint types are **not equal**.
  (There is a significant difference.)

## b) Statistical Test Used

```
[192]: from scipy import stats
       # Calculate the mean closing time for each complaint type
       complainnt_meanns = rd.groupby('Complaint Type')['Request_Closing_Time'].mean()

[193]: # Print the means for each complaint type
       print(complainnt_meanns)

       Complaint Type
       Animal Abuse               5.212391
       Blocked Driveway           4.763904
       Derelict Vehicle           7.471085
       Disorderly Youth           3.624137
       Drinking                   3.867348
       Graffiti                   6.857846
       Illegal Parking            4.464806
       Noise - Commercial         3.170920
       Noise - House of Worship   3.042583
       Noise - Park               3.338813
       Noise - Street/Sidewalk    3.542232
       Noise - Vehicle            3.692305
       Posting Advertisement      1.786489
       Traffic                    3.450760
       Vending                    4.048535
       Name: Request_Closing_Time, dtype: float64
```

*Figure 32: average response time across complaint types is similar or not(i)*

### Step 1: Import Required Libraries

```
from scipy import stats
```

You're importing scipy.stats to use the **independent t-test** function ttest_ind(), which compares the means of two groups to see if they differ significantly.

### Step 2: Calculate Mean Closing Time per Complaint Type

```
complainnt_meanns=rd.groupby('ComplaintType')['Request_Closing_Time'].mean()
```

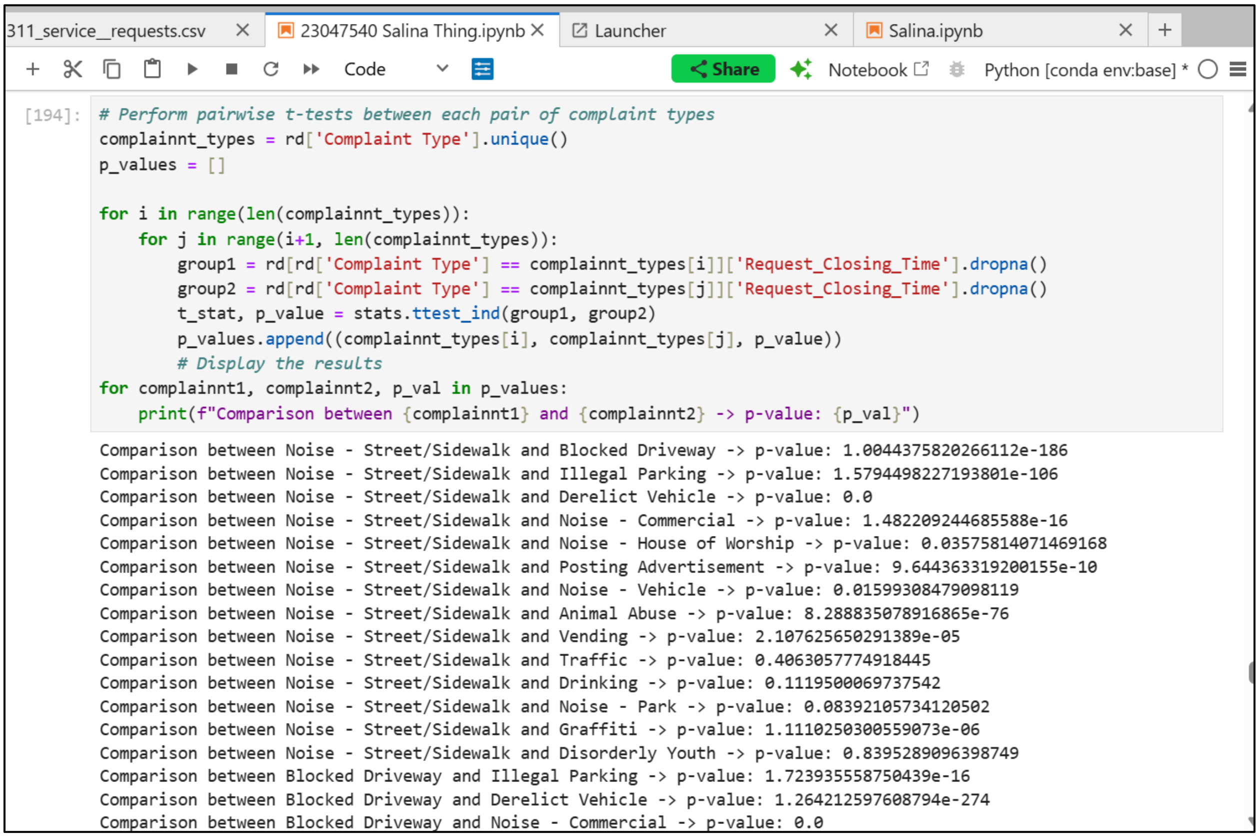
This groups the dataset by 'Complaint Type' and calculates the **average closing time** (in hours).

Result: A pandas.Series showing the **mean closing time** for each complaint type.

### Step 3: Display Mean Closing Time

```
print(complainnt_meanns)
```

This prints the **average request closing time** for each type of complaint to see overall differences before testing.



```
# Perform pairwise t-tests between each pair of complaint types
complainnt_types = rd['Complaint Type'].unique()
p_values = []

for i in range(len(complainnt_types)):
    for j in range(i+1, len(complainnt_types)):
        group1 = rd[rd['Complaint Type'] == complainnt_types[i]]['Request_Closing_Time'].dropna()
        group2 = rd[rd['Complaint Type'] == complainnt_types[j]]['Request_Closing_Time'].dropna()
        t_stat, p_value = stats.ttest_ind(group1, group2)
        p_values.append((complainnt_types[i], complainnt_types[j], p_value))
        # Display the results
for complainnt1, complainnt2, p_val in p_values:
    print(f"Comparison between {complainnt1} and {complainnt2} -> p-value: {p_val}")
```

```
Comparison between Noise - Street/Sidewalk and Blocked Driveway -> p-value: 1.0044375820266112e-186
Comparison between Noise - Street/Sidewalk and Illegal Parking -> p-value: 1.5794498227193801e-106
Comparison between Noise - Street/Sidewalk and Derelict Vehicle -> p-value: 0.0
Comparison between Noise - Street/Sidewalk and Noise - Commercial -> p-value: 1.482209244685588e-16
Comparison between Noise - Street/Sidewalk and Noise - House of Worship -> p-value: 0.03575814071469168
Comparison between Noise - Street/Sidewalk and Posting Advertisement -> p-value: 9.644363319200155e-10
Comparison between Noise - Street/Sidewalk and Noise - Vehicle -> p-value: 0.01599308479098119
Comparison between Noise - Street/Sidewalk and Animal Abuse -> p-value: 8.288835078916865e-76
Comparison between Noise - Street/Sidewalk and Vending -> p-value: 2.107625650291389e-05
Comparison between Noise - Street/Sidewalk and Traffic -> p-value: 0.4063057774918445
Comparison between Noise - Street/Sidewalk and Drinking -> p-value: 0.1119500069737542
Comparison between Noise - Street/Sidewalk and Noise - Park -> p-value: 0.08392105734120502
Comparison between Noise - Street/Sidewalk and Graffiti -> p-value: 1.111025030055907 3e-06
Comparison between Noise - Street/Sidewalk and Disorderly Youth -> p-value: 0.8395289096398749
Comparison between Blocked Driveway and Illegal Parking -> p-value: 1.723935558750439e-16
Comparison between Blocked Driveway and Derelict Vehicle -> p-value: 1.264212597608794e-274
Comparison between Blocked Driveway and Noise - Commercial -> p-value: 0.0
```

*Figure 33: average response time across complaint types is similar or not(ii)*

### Step 4: Setup for Pairwise t-tests

```
complainnt_types = rd['Complaint Type'].unique()
p_values = []
```

complainnt_types: An array of all **unique complaint types**.

p_values: A list to store the **pairwise comparison results**.

### Step 5: Perform Pairwise t-tests Between Complaint Types

```
for i in range(len(complainnt_types)):
```

```
    for j in range(i+1, len(complainnt_types)):
        group1 = rd[rd['Complaint Type'] ==
complainnt_types[i]]['Request_Closing_Time'].dropna()
        group2 = rd[rd['Complaint Type'] ==
complainnt_types[j]]['Request_Closing_Time'].dropna()
        t_stat, p_value = stats.ttest_ind(group1, group2)
        p_values.append((complainnt_types[i], complainnt_types[j], p_value))
```

Loops through each **pair of complaint types**.

Extracts their respective Request_Closing_Time values (excluding NaN).

Performs an **independent two-sample t-test** (assumes unequal samples but equal variance by default).

Stores the result (complainnt1, complainnt2, p-value) in the list.

### Step 6: Print the Pairwise p-values

```
for complainnt1, complainnt2, p_val in p_values:
    print(f"Comparison between {complainnt1} and {complainnt2} -> p-value: {p_val}")
```

This prints the **statistical test results** for each pair.

Helps identify if there is a **statistically significant difference** in average closing time between specific complaint types.

**If p-value < 0.05**, then the difference is statistically significant.

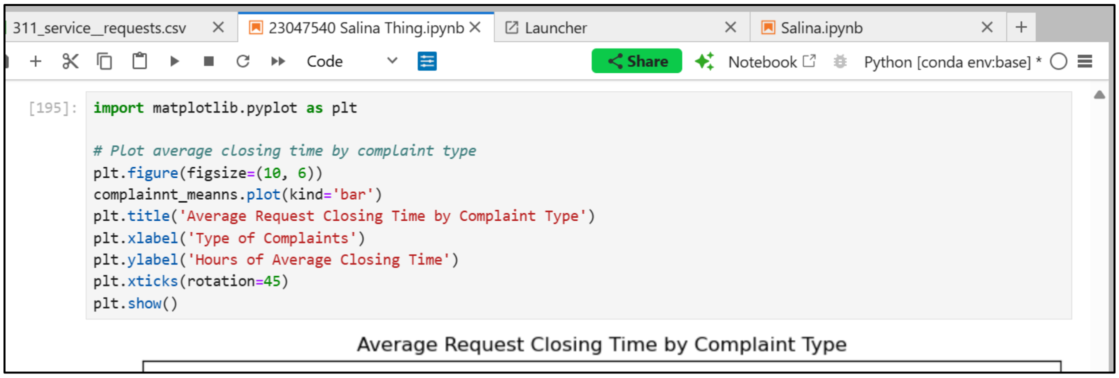



*Figure 34: average response time across complaint types is similar or not(iii)*

### Step 7: Visualize Average Closing Time by Complaint Type

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
complainnt_meanns.plot(kind='bar')
plt.title('Average Request Closing Time by Complaint Type')
plt.xlabel('Complaint Type')
plt.ylabel('Average Closing Time')
plt.xticks(rotation=45)
plt.show()
```

plt.figure(figsize=(10, 6)): Sets the size of the plot.

complainnt_meanns.plot(kind='bar'): Creates a **bar chart** of the average closing time per complaint type.

Adds title and axis labels for clarity.

plt.xticks(rotation=45): Rotates x-axis labels to prevent overlap.

plt.show(): Displays the plot.

### c) Interpretation of Results

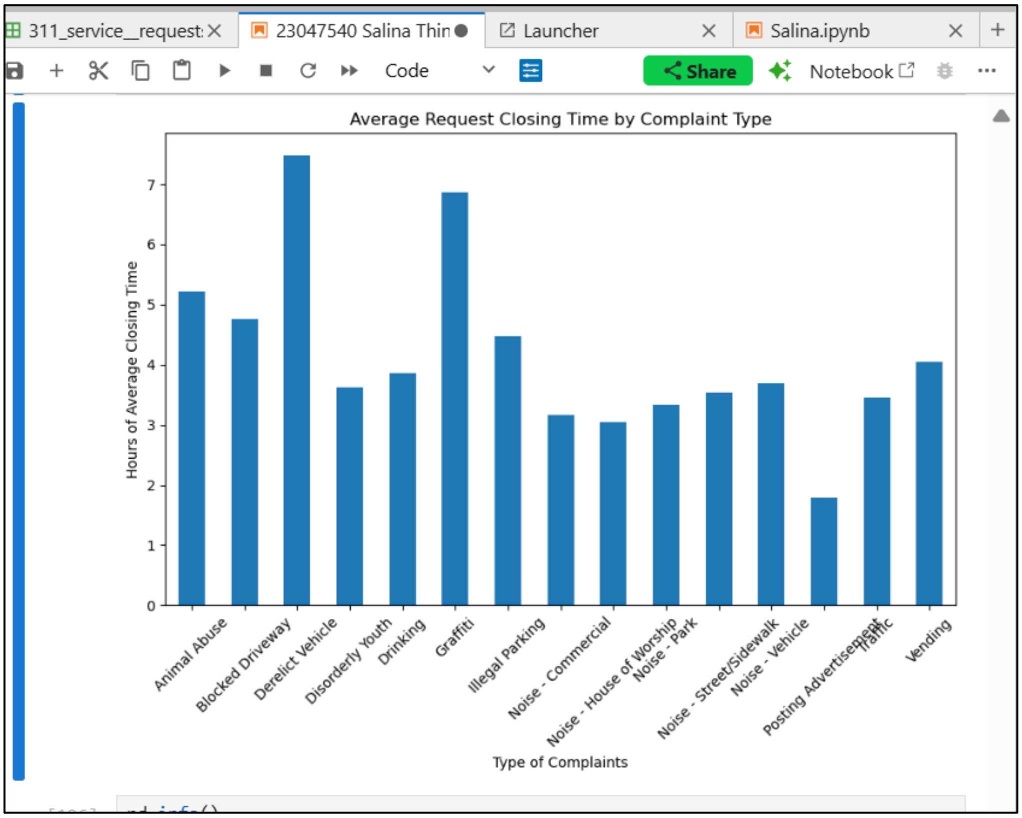



*Figure 35: average response time across complaint types is similar or not(iv)*

- The **bar chart** visually shows which complaints take the longest (or shortest) time to close.
- The **pairwise p-values** reveal **which complaint types have significantly different resolution times**.

**Interpreting the p-values**

- **If p-value < 0.05**: Reject the null hypothesis. → There **is a statistically significant difference** in average closing time between those complaint types.
- **If p-value ≥ 0.05**: Fail to reject the null hypothesis. → There **is no statistically significant difference** in average closing time between those complaint types.

If Noise - Street/Sidewalk has a much higher average closing time and a **low p-value when compared to others**, that implies a **real difference** in how long such complaints take to resolve.

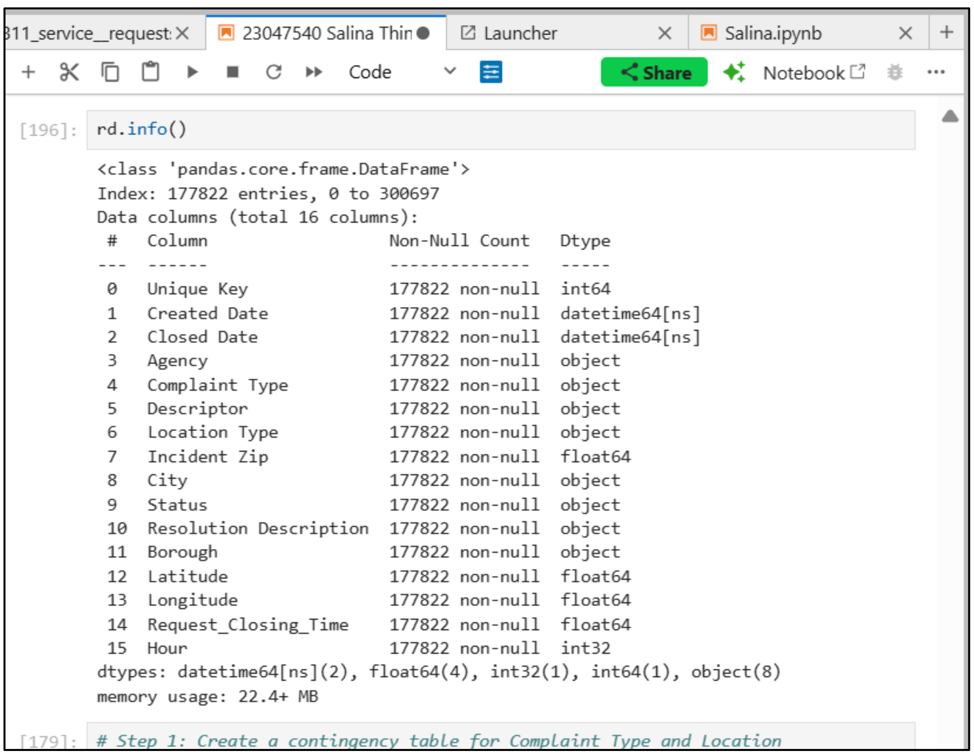



*Figure 36: Information of data in dataset*

I used this code here to see the columns name present in the datasets for further analysis.

## 5.2. Test 2 Objective

Whether the type of complaint or service requested, and location are related.

### a) Null and Alternative Hypotheses

- **Null Hypothesis ($H_0$)**:
  The average response time is the same across all complaint types.
  (No significant difference in mean closing time.)

- **Alternative Hypothesis ($H_1$)**:
  At least one complaint type has a significantly different average response time.

### b) Perform the Statistical Test

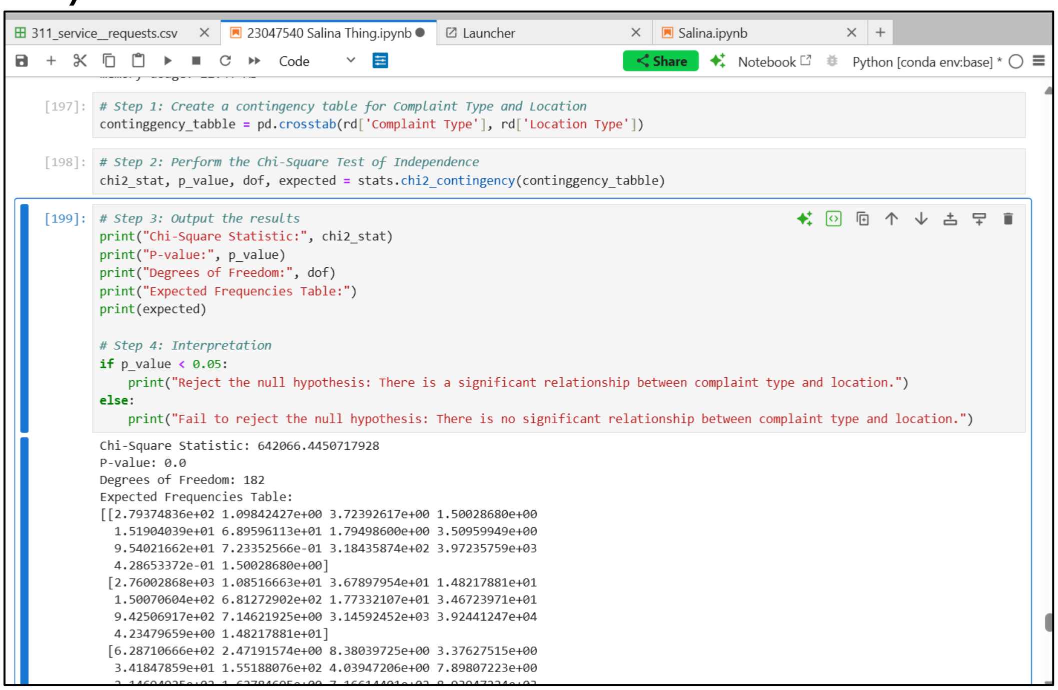



*Figure 37: Whether the type of complaint or service requested, and location are related(i)*

**Step 1: Create a contingency table**

```
continggency_tabble = pd.crosstab(rd['Complaint Type'], rd['Location Type'])
```

A **continggency tabble** shows how frequently each complaint type occurs at each location type.

**Step 2: Perform the Chi-Square Test**

```
chi2_stat, p_value, dof, expected = stats.chi2_contingency(continggency_tabble)
```

This tests **whether the distribution of complaint types is independent of location types**.

**chi2_stat:** The Chi-Square test statistic (how far the observed values deviate from expected values).

**p_value**: Probability that the observed distribution could occur under the null hypothesis.

**dof:** Degrees of freedom, calculated as (rows−1)×(columns−1)(rows - 1) × (columns - 1)(rows−1)×(columns−1)

**expected:** The **expected frequencies** table — what the counts would look like if complaint type and location were truly independent.

**Step 3: Output Results**

```
print("Chi-Square Statistic:", chi2_stat)
print("P-value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies Table:")
print(expected)
```

Whether the actual counts **significantly deviate** from what we'd expect under independence.

If the p-value is small, the deviation is too large to be random — meaning **they're likely related**.

## c) Interpret of Results

```
if p_value < 0.05:
    print("Reject the null hypothesis: There is a significant relationship between complaint type and location.")
else:
    print("Fail to reject the null hypothesis: There is no significant relationship between complaint type and location.")
```

- **If p-value < 0.05** → Reject the null hypothesis: There **is a significant difference** in average response time across complaint types.
- **If p-value ≥ 0.05** → Fail to reject the null: No significant difference; complaint types have similar average response times.

Since **p-value ≈ 0.000** (far less than 0.05), we **reject the null hypothesis**.

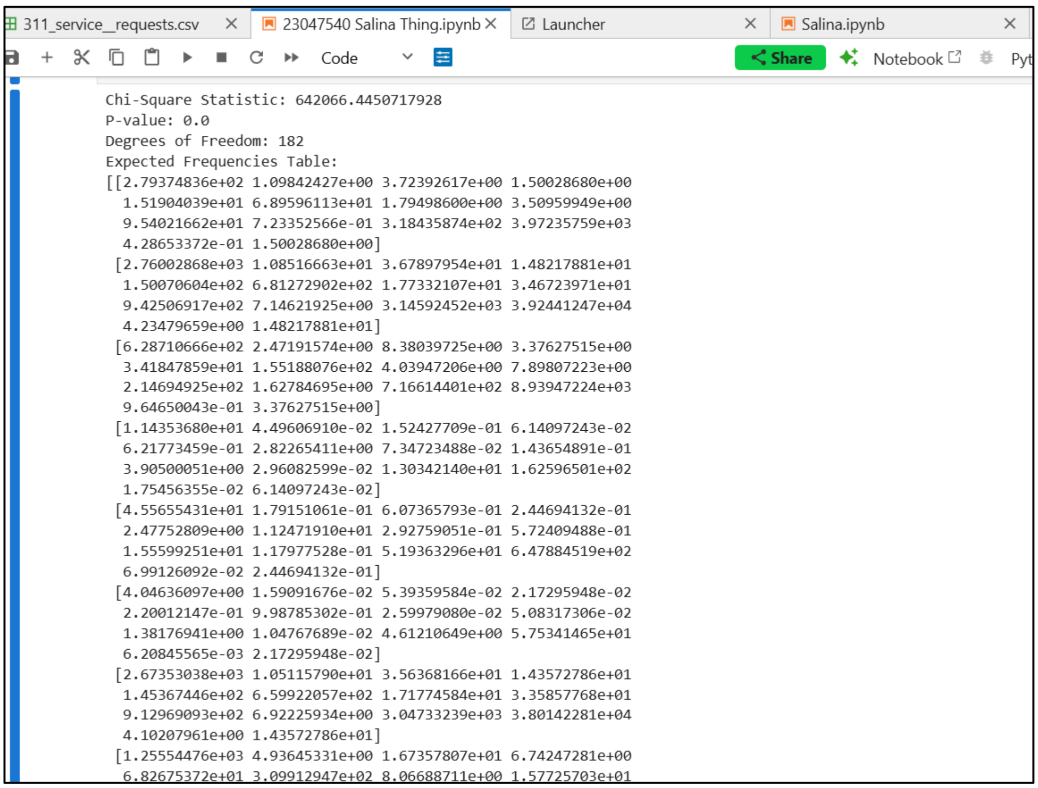



*Figure 38: Whether the type of complaint or service requested, and location are related(ii)*

**Since p-value < 0.05**, you **reject the null hypothesis**. This suggests there **is a statistically significant relationship** between the type of complaint and the type of location.
For example, **noise complaints** might be more frequent in **residential areas**, while **parking issues** are more common on **streets**.

There is a **significant relationship** between the **type of complaint** and the **location type**.
Certain complaints are more likely to occur in certain locations (e.g., noise in residential buildings, illegal parking on streets).

## References

GeeksforGeeks, 2024. *Libraries in Python.* [Online]
Available at: https://www.geeksforgeeks.org/libraries-in-python/
[Accessed 3 May 2025].

Kenton, W., 2024. *Investopedia.* [Online]
Available at:
https://www.investopedia.com/terms/k/kurtosis.asp#:~:text=Kurtosis%20describes%20how%20much%20of,shape%20of%20the%20distribution%20accordingly.
[Accessed 15 May 2025].

MotherDuck, 2025. *MotherDuck.* [Online]
Available at: https://motherduck.com/docs/getting-started/sample-data-queries/nyc-311-data/
[Accessed 11 April 2025].

National Library Of Medicine, 2025. *Standard Deviation.* [Online]
Available at: https://www.nlm.nih.gov/oet/ed/stats/02-900.html
[Accessed 15 May 2025].

Newcastle University, 2025. *Null and Alternative Hypotheses.* [Online]
Available at: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/hypothesis-testing/null-and-alternative-hypotheses.html#:~:text=The%20null%20hypothesis%20H0,type%20of%20test%20to%20use.
[Accessed 3 May 2025].

Shubham, 2021. *Kaggle.* [Online]
Available at: https://www.kaggle.com/datasets/shubhammore12/nyc-311-customer-service-requests-analysis
[Accessed 11 April 2025].

Turney, S., 2022. *Scribbr.* [Online]
Available at:
https://www.scribbr.com/statistics/skewness/#:~:text=Skewness%20is%20a%20measure%20of,negative)%2C%20or%20zero%20skewness.
[Accessed 15 May 2025].

W3 schools, 2025. *W3 schools.* [Online]
Available at: https://www.w3schools.com/python/pandas/pandas_correlations.asp
[Accessed 11 April 2025].