Traffic Light Complaints vs. Car Crashes in NYC

Group 5

Introduction

New York is known for its chaotic traffic conditions, but how severe can it be? And are the driving conditions a significant cause? In our project, we want to explore the correlation between crashes— a major traffic issue— and the number of complaints about traffic signals and street light conditions. In case of a positive correlation, we hope to provide evidence to incentivize the city to invest in improving traffic signals and street lights and thus avoid, possibly fatal, crashes.

Our primary dataset will be the "311 Service Requests from 2010 to Present"; filtered for Traffic Signal Condition and Street Light Condition. Our second dataset will be the "Motor Vehicle Collisions - Crashes." Both datasets are from NYC Open Data and are updated frequently.

In this project, we will implement the steps to create a data warehouse to then support the analysis process, initially described. To start creating the data warehouse we first defined the Key Performance Indicators (KPIs). Then, we will develop the Dimensional Model based on the KPIs. In this document we included the several drafts that lead to the final model. After setting the final dimensional model, we will select ETL Tools and Target DBMS to then perform the ETL process. Finally, we will create a dashboard with the selected KPIs. In this document we describe the Dimensional Modeling Process, which includes the initial draft, second draft, and final Dimensional Model.

Key Performance Indicators

1. Number of crashes (by borough)
2. Number of people killed by the crashes (by borough)
3. Most prevalent 311 complaints regarding street lights (by borough)

Datasource Description

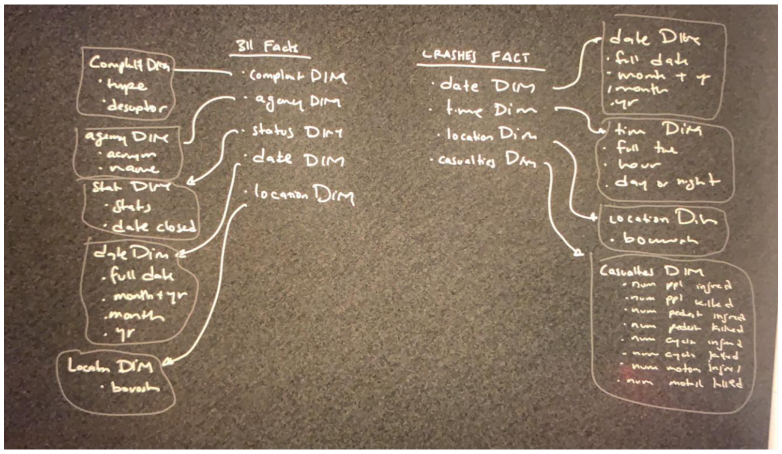
With 27 million records and 41 columns, the main dataset "311 Service Requests from 2010 to Present" includes all the 311 calls or service requests. Each row describes a specific call or service request. The most relevant columns include "created date," "agency", "complaint type", "borough", etc. The secondary dataset "Motor Vehicle Collisions - Crashes" includes 1.84 million records and 29 columns. Each row describes a specific vehicle collision incident. The most relevant columns include"number of people injured" , “crash date", "crash time", "borough", etc.

Main Datasource: 311 Service Requests from 2010 to Present
https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9
Secondary Datasource: Motor Vehicle Collisions - Crashes
https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95

DIMENSIONAL MODELING

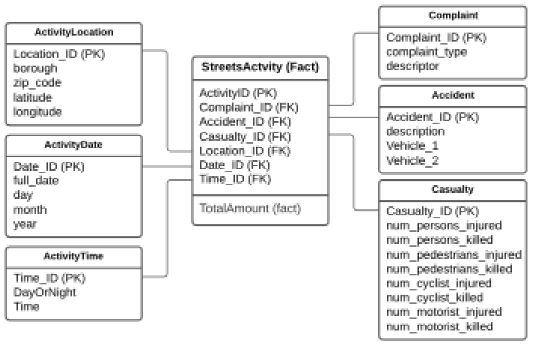
Initial Draft: Dimensional Model

Stand-alone fact tables for 311 and Crashes, which we initially thought about then linking to another fact table that would consolidate the information from each respective fact table. We quickly realized how doing so would give us issues when we link the data sets together when creating the 3rd fact table which would consolidate both fact tables.



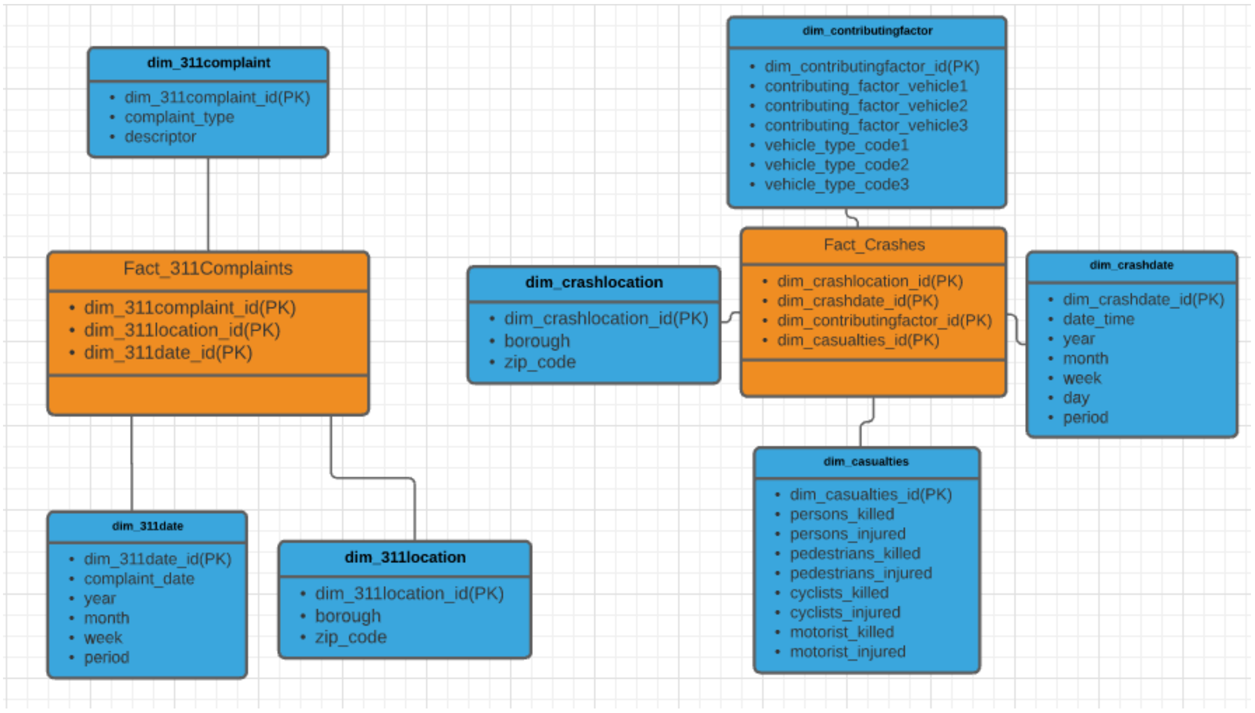
Second Draft: Dimensional Model

Since the Dimensional Modeling Process is iterative, we create a second draft. This second draft reflects closer the activity we are trying to record; summarized as *Street Activit*y. Which includes *complaints* (311 calls) and *accidents* (crashes). The main fact being recorded is *Total Amount*, both on accident and complaints. The grain selected total amount, per location (*longitude* and *latitude*), and date (*day*). The two initial datasets include many more information, street name, off street name and community board, etc. However, some of the information might not be used.



Final Dimensional Model

Taking Professor Vaghefi's suggestions, we ended up going back to creating two stand-alone fact tables for the final version of the dimensional model. We consolidated the date and time dimension to one table for both fact tables so that it includes the datetime stamps, year, month, week (number), and period (AM/PM) for each respectively. We also decided to remove the coordinates (longitude and latitude) values from the location dimensions for both fact tables. We chose to retain information of up to three vehicles involved in the crash; the overwhelming majority of the dataset involved 3 or less vehicles in accidents. Other than that, we kept the casualty table intact but renamed it to "casualties" instead.



Extract, Transform, Load (ETL)

Tools and Software Utilized

For the ETL process, we decided to use the following tools:

- ***Socrata OpenData API:***
    - To pull data the sets from OpenData; the datasets are too large to download them locally
- ***Jupyter Notebook:***
    - For writing code in Python
- ***Github:***
    - For version control and for hosting the files to make them easier to navigate for the group (copy of zip/csv for dimension table for backup, copy of the most recent version of the ETL code, etc.)
- ***Python:***
    - For **extracting** the datasets from OpenData using the Socrata OpenData API and sodapy library
    - Utilized libraries such as pandas, datetime, and calendar to **transform** the datasets
    - For **loading** data to Google BigQuery; created connection to Google BigQuery via google.cloud (bigquery), pandas.io, and pandas-gbq
- ***Google BigQuery:***
    - Used BigQuery to house the data (dimension tables)

ETL Plan

As described in the bullets above, we will be heavily utilizing Python and its libraries and APIs to perform the bulk of the extracting, transforming, and loading of the datasets. Although we had options such as Google DataFlow and Pentaho available to us, the group came to a consensus that for the scope and scale of this project, Python would be a completely viable option. We also happen to be the most comfortable with using Python as well.

We would then load the data (dimension tables) into Google BigQuery and house it there in preparation for the ad-hoc analysis and dashboarding that we will be performing later.

ETL Process in Action

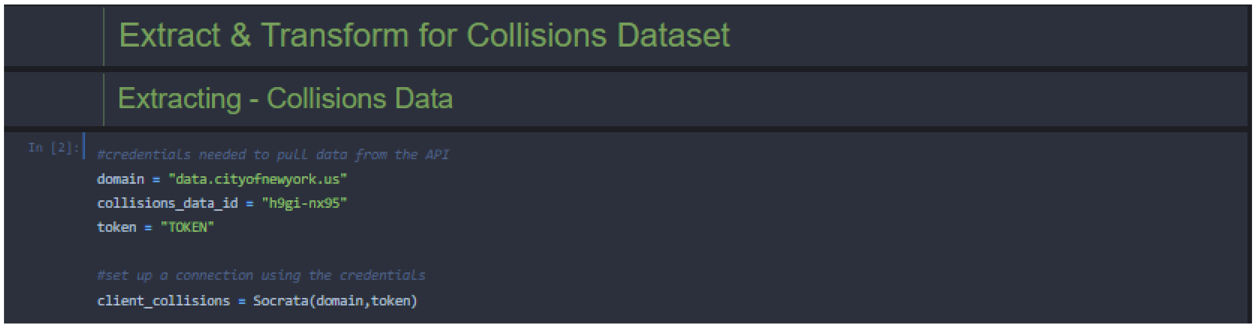
Link to Jupyter Notebook :
https://github.com/mendozadylan0/cis4400_finalproject/blob/main/CIS4400_FinalProj_ETL_CredClear.ipynb

- **Setting Up**

  First, we import the libraries and packages that we will be using for the ETL process.

- **Extract & Transform for Collisions Dataset:**
  - **Extracting - Collisions Data**

  We then proceed with setting up the credentials needed for connecting with the Socrata API and for pulling the "Motor Vehicle Collisions - Crashes" dataset from the said API. For convenience, we will be using "collisions dataset" and any variations thereof to refer to the "Motor Vehicle Collisions - Crashes" dataset. We then establish a connection with the API.

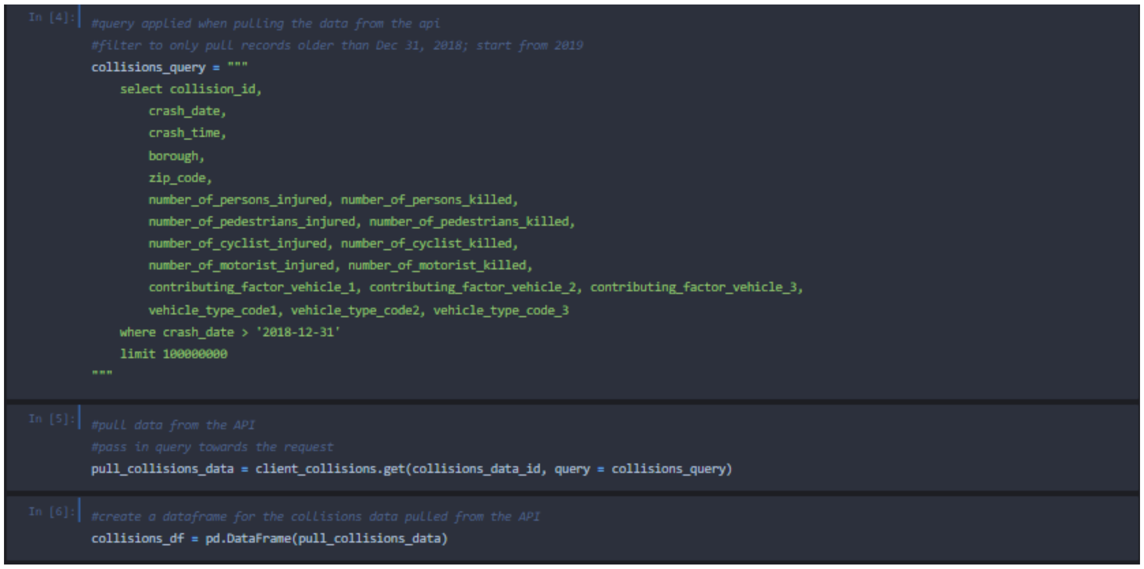


Extract & Transform for Collisions Dataset

Extracting - Collisions Data

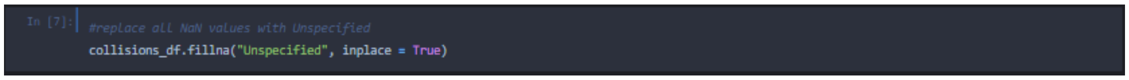
```
In [2]: #credentials needed to pull data from the API
        domain = "data.cityofnewyork.us"
        collisions_data_id = "h9gi-nx95"
        token = "TOKEN"

        #set up a connection using the credentials
        client_collisions = Socrata(domain,token)
```

After that, we passed a query to the Socrata API (it allows you to pass SQL like queries before you extract the data -- very handy for filtering the data down). In the query, we opted to only extract records from the collisions dataset dated after December 31, 2018. In other words, we're going to be using the records starting from January 1, 2019 until the present. Following this, we proceeded with pulling the collisions dataset via the API and then loading it into a pandas dataframe.



```
In [4]: #query applied when pulling the data from the api
        #filter to only pull records older than Dec 31, 2018; start from 2019
        collisions_query = """
            select collision_id,
                crash_date,
                crash_time,
                borough,
                zip_code,
                number_of_persons_injured, number_of_persons_killed,
                number_of_pedestrians_injured, number_of_pedestrians_killed,
                number_of_cyclist_injured, number_of_cyclist_killed,
                number_of_motorist_injured, number_of_motorist_killed,
                contributing_factor_vehicle_1, contributing_factor_vehicle_2, contributing_factor_vehicle_3,
                vehicle_type_code1, vehicle_type_code2, vehicle_type_code_3
            where crash_date > '2018-12-31'
            limit 100000000
        """

In [5]: #pull data from the API
        #pass in query towards the request
        pull_collisions_data = client_collisions.get(collisions_data_id, query = collisions_query)

In [6]: #create a dataframe for the collisions data pulled from the API
        collisions_df = pd.DataFrame(pull_collisions_data)
```

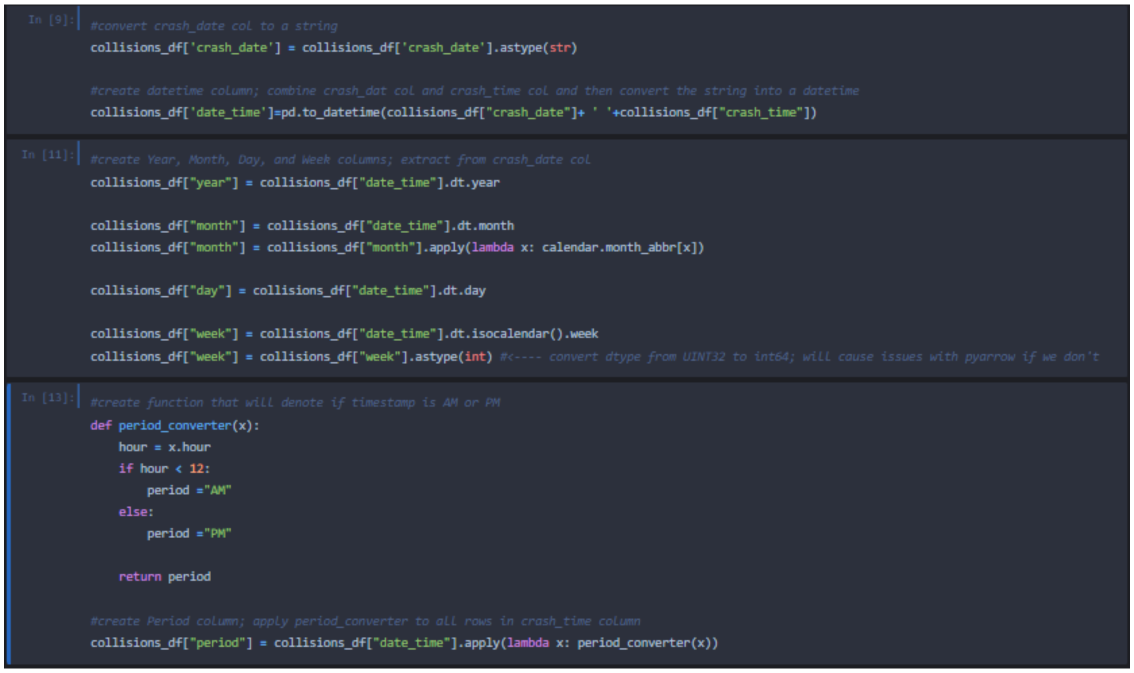
Upon loading the collisions dataset into a pandas dataframe, we replaced all "NaN" values with "Unspecified". After careful deliberation, we came to the conclusion that the nature of the information that's recorded in the dataset makes it so that there is value in knowing that the value for a particular column is missing. Since some records showed entries of "Unspecified" as inputs to columns where values were missing or unknown, we chose to use "Unspecified" as the substitute value to bring it inline with the data set as well as to make data validation easier.



```
In [7]: #replace all NaN values with Unspecified
        collisions_df.fillna("Unspecified", inplace = True)
```

- **Transforming - Collisions Data**

We first converted the crash_date column to a *string* so that we can combine it with the crash_time column and then convert it into a *datetime* object. The combination of the two will create a datetime stamp and this will be stored in a new column called date_time. We then proceeded to extract the year, month, week (number), and period (AM/PM) from the date_time column. We used Python's *datetime* and *calendar* libraries to do so. Before moving on, we coerced the "week" column to an int64 data type to make sure that we don't run into issues later on when we load the data into BigQuery. With that out of the way, we then created the "period" column to denote whether the collision/crash took place during the morning (AM) or in the afternoon and evening (PM).

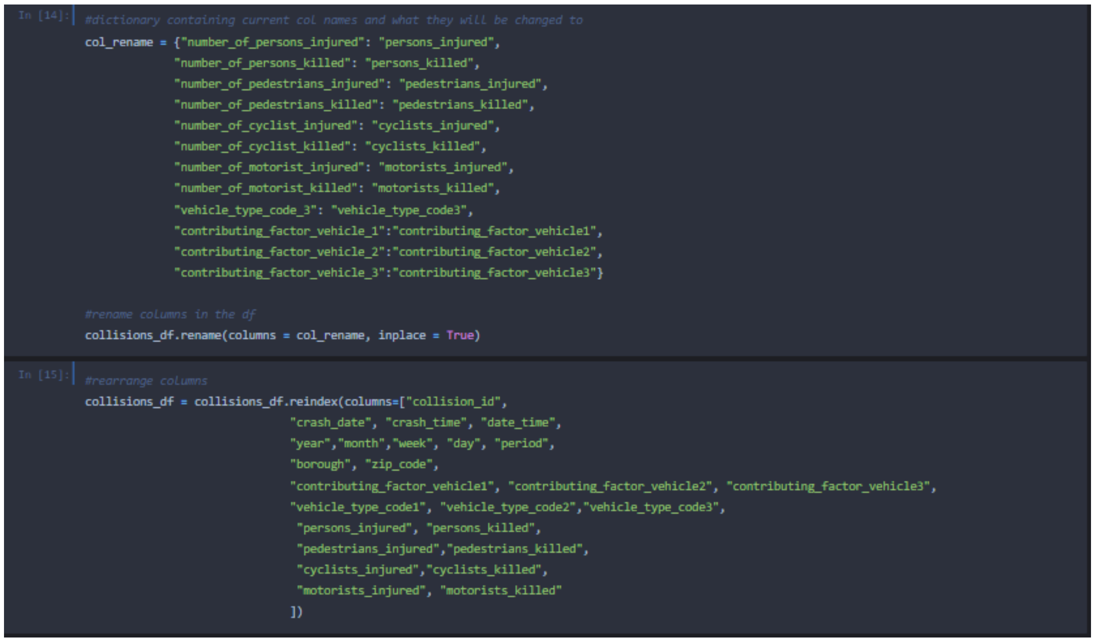


```
In [9]:  #convert crash_date col to a string
         collisions_df['crash_date'] = collisions_df['crash_date'].astype(str)

         #create datetime column; combine crash_dat col and crash_time col and then convert the string into a datetime
         collisions_df['date_time']=pd.to_datetime(collisions_df["crash_date"]+ ' '+collisions_df["crash_time"])
```

```
In [11]: #create Year, Month, Day, and Week columns; extract from crash_date col
         collisions_df["year"] = collisions_df["date_time"].dt.year

         collisions_df["month"] = collisions_df["date_time"].dt.month
         collisions_df["month"] = collisions_df["month"].apply(lambda x: calendar.month_abbr[x])

         collisions_df["day"] = collisions_df["date_time"].dt.day

         collisions_df["week"] = collisions_df["date_time"].dt.isocalendar().week
         collisions_df["week"] = collisions_df["week"].astype(int) #<---- convert dtype from UINT32 to int64; will cause issues with pyarrow if we don't
```

```
In [13]: #create function that will denote if timestamp is AM or PM
         def period_converter(x):
             hour = x.hour
             if hour < 12:
                 period ="AM"
             else:
                 period ="PM"

             return period

         #create Period column; apply period_converter to all rows in crash_time column
         collisions_df["period"] = collisions_df["date_time"].apply(lambda x: period_converter(x))
```

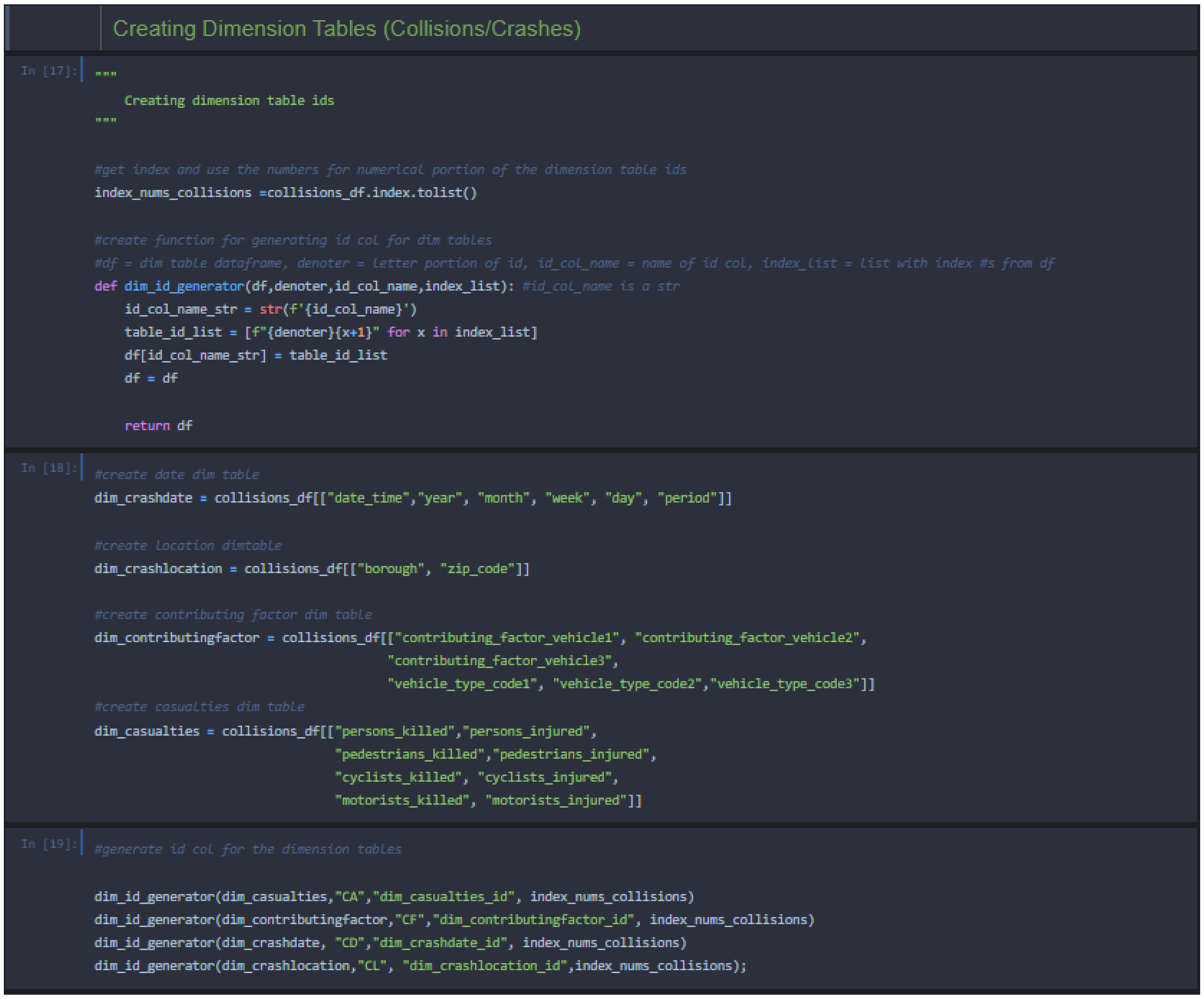
After this we then proceeded to rename the columns to be more inline with how we want the column headers to look like in our respective dimension tables. Also, for organization and legibility purposes, we decided to rearrange the columns. We didn't want to have to jump from one column from the left-most side of the dataframe and then have to scroll the way to the right-most side if we wanted to take a look at certain columns (i.e collision _id and crash_date/crash_time).



```
In [14]: #dictionary containing current col names and what they will be changed to
         col_rename = {"number_of_persons_injured": "persons_injured",
                      "number_of_persons_killed": "persons_killed",
                      "number_of_pedestrians_injured": "pedestrians_injured",
                      "number_of_pedestrians_killed": "pedestrians_killed",
                      "number_of_cyclist_injured": "cyclists_injured",
                      "number_of_cyclist_killed": "cyclists_killed",
                      "number_of_motorist_injured": "motorists_injured",
                      "number_of_motorist_killed": "motorists_killed",
                      "vehicle_type_code_3": "vehicle_type_code3",
                      "contributing_factor_vehicle_1":"contributing_factor_vehicle1",
                      "contributing_factor_vehicle_2":"contributing_factor_vehicle2",
                      "contributing_factor_vehicle_3":"contributing_factor_vehicle3"}

         #rename columns in the df
         collisions_df.rename(columns = col_rename, inplace = True)
```

```
In [15]: #rearrange columns
         collisions_df = collisions_df.reindex(columns=["collision_id",
                                   "crash_date", "crash_time", "date_time",
                                   "year","month","week", "day", "period",
                                   "borough", "zip_code",
                                   "contributing_factor_vehicle1", "contributing_factor_vehicle2", "contributing_factor_vehicle3",
                                   "vehicle_type_code1", "vehicle_type_code2","vehicle_type_code3",
                                    "persons_injured", "persons_killed",
                                    "pedestrians_injured","pedestrians_killed",
                                    "cyclists_injured","cyclists_killed",
                                    "motorists_injured", "motorists_killed"
                                   ])
```

We then created the dimension tables for the collisions dataset. After partitioning the dataset into the dimension tables, we then proceeded to create a dimension table ID column via a self-defined function called dim_id_generator. The dimension table ID column's values will consist of a 2 letter abbreviation of the dimension table's name (in uppercase) and the row index number of the pandas dataframe that the record is assigned. So for example, the first record in the casualties dimension table would be "CA01".



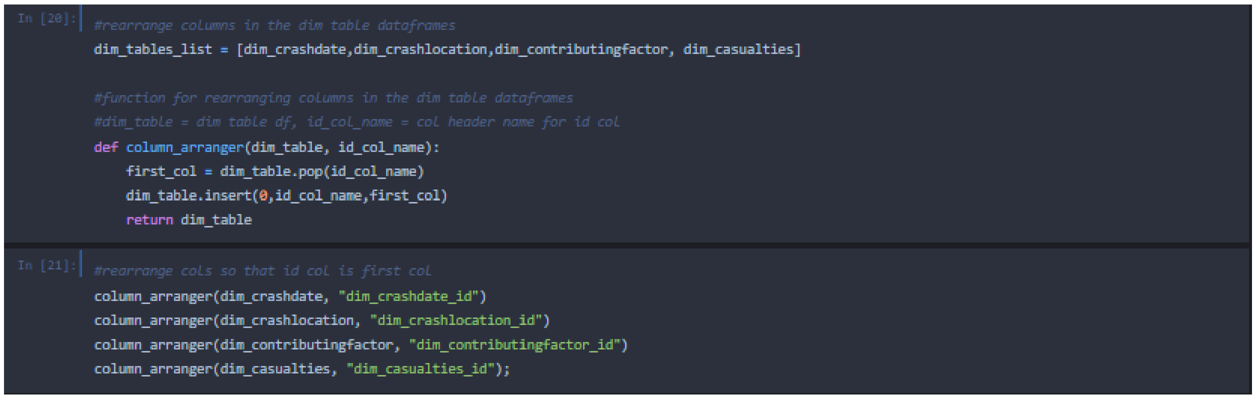
Creating Dimension Tables (Collisions/Crashes)

In [17]:
```
"""
    Creating dimension table ids
"""

#get index and use the numbers for numerical portion of the dimension table ids
index_nums_collisions =collisions_df.index.tolist()

#create function for generating id col for dim tables
#df = dim table dataframe, denoter = letter portion of id, id_col_name = name of id col, index_list = list with index #s from df
def dim_id_generator(df,denoter,id_col_name,index_list): #id_col_name is a str
    id_col_name_str = str(f'{id_col_name}')
    table_id_list = [f"{denoter}{x+1}" for x in index_list]
    df[id_col_name_str] = table_id_list
    df = df

    return df
```

In [18]:
```
#create date dim table
dim_crashdate = collisions_df[["date_time","year", "month", "week", "day", "period"]]

#create location dimtable
dim_crashlocation = collisions_df[["borough", "zip_code"]]

#create contributing factor dim table
dim_contributingfactor = collisions_df[["contributing_factor_vehicle1", "contributing_factor_vehicle2",
                                       "contributing_factor_vehicle3",
                                       "vehicle_type_code1", "vehicle_type_code2","vehicle_type_code3"]]
#create casualties dim table
dim_casualties = collisions_df[["persons_killed","persons_injured",
                                "pedestrians_killed","pedestrians_injured",
                                "cyclists_killed", "cyclists_injured",
                                "motorists_killed", "motorists_injured"]]
```

In [19]:
```
#generate id col for the dimension tables

dim_id_generator(dim_casualties,"CA","dim_casualties_id", index_nums_collisions)
dim_id_generator(dim_contributingfactor,"CF","dim_contributingfactor_id", index_nums_collisions)
dim_id_generator(dim_crashdate, "CD","dim_crashdate_id", index_nums_collisions)
dim_id_generator(dim_crashlocation,"CL", "dim_crashlocation_id",index_nums_collisions);
```

We then rearranged the columns a little so that the said ID column was the left-most column for all of the dimension tables that we just created. This was done using the self-defined function called column_arranger.

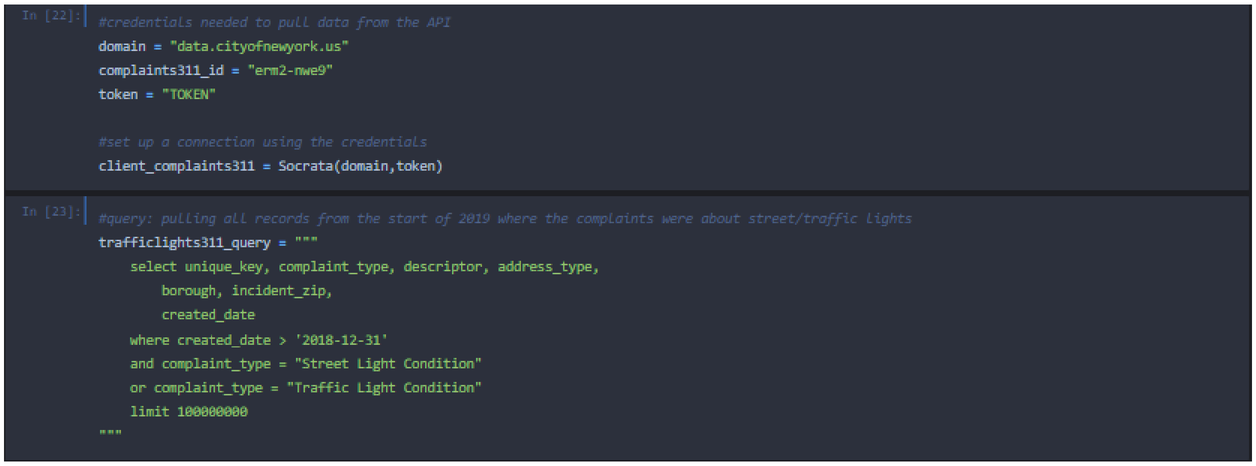


```
In [20]: #rearrange columns in the dim table dataframes
         dim_tables_list = [dim_crashdate,dim_crashlocation,dim_contributingfactor, dim_casualties]

         #function for rearranging columns in the dim table dataframes
         #dim_table = dim table df, id_col_name = col header name for id col
         def column_arranger(dim_table, id_col_name):
             first_col = dim_table.pop(id_col_name)
             dim_table.insert(0,id_col_name,first_col)
             return dim_table
```

```
In [21]: #rearrange cols so that id col is first col
         column_arranger(dim_crashdate, "dim_crashdate_id")
         column_arranger(dim_crashlocation, "dim_crashlocation_id")
         column_arranger(dim_contributingfactor, "dim_contributingfactor_id")
         column_arranger(dim_casualties, "dim_casualties_id");
```

- **Extract & Transform for 311 Traffic Lights Complaints:**
  - **Extracting - 311 Traffic Lights Complaints Data**

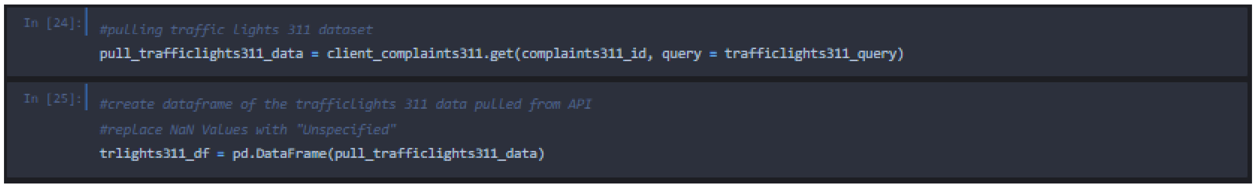
We started off by setting up the credentials needed to connect to the Socrata API and extract the 311 data set. This was followed by us defining a query that we will pass to the API that would filter the exact columns that we wanted to work with for creating our dimension tables.



```
In [22]: #credentials needed to pull data from the API
         domain = "data.cityofnewyork.us"
         complaints311_id = "erm2-nwe9"
         token = "TOKEN"

         #set up a connection using the credentials
         client_complaints311 = Socrata(domain,token)
```

```
In [23]: #query: pulling all records from the start of 2019 where the complaints were about street/traffic lights
         trafficlights311_query = """
             select unique_key, complaint_type, descriptor, address_type,
                 borough, incident_zip,
                 created_date
             where created_date > '2018-12-31'
             and complaint_type = "Street Light Condition"
             or complaint_type = "Traffic Light Condition"
             limit 100000000
         """
```

We then passed the query that we defined and extracted the dataset from the API, which was followed by us loading the data into a pandas dataframe.

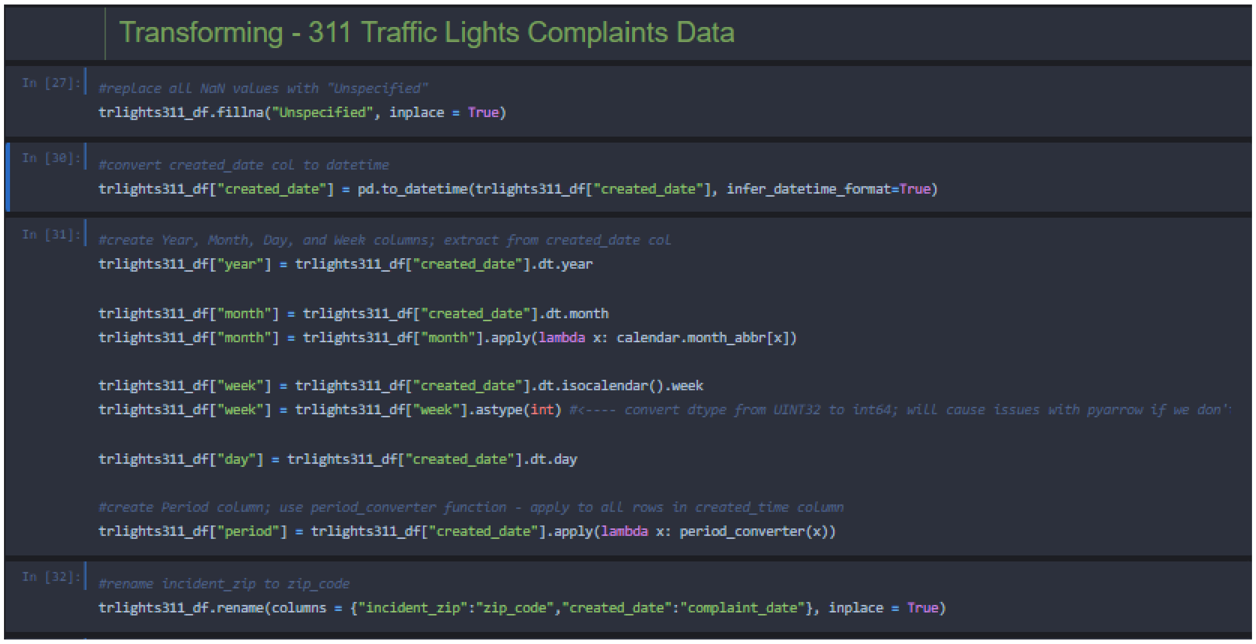


```
In [24]: #pulling traffic lights 311 dataset
         pull_trafficlights311_data = client_complaints311.get(complaints311_id, query = trafficlights311_query)
```

```
In [25]: #create dataframe of the trafficlights 311 data pulled from API
         #replace NaN Values with "Unspecified"
         trlights311_df = pd.DataFrame(pull_trafficlights311_data)
```

- **Transforming - 311 Traffic Lights Complaints Data**

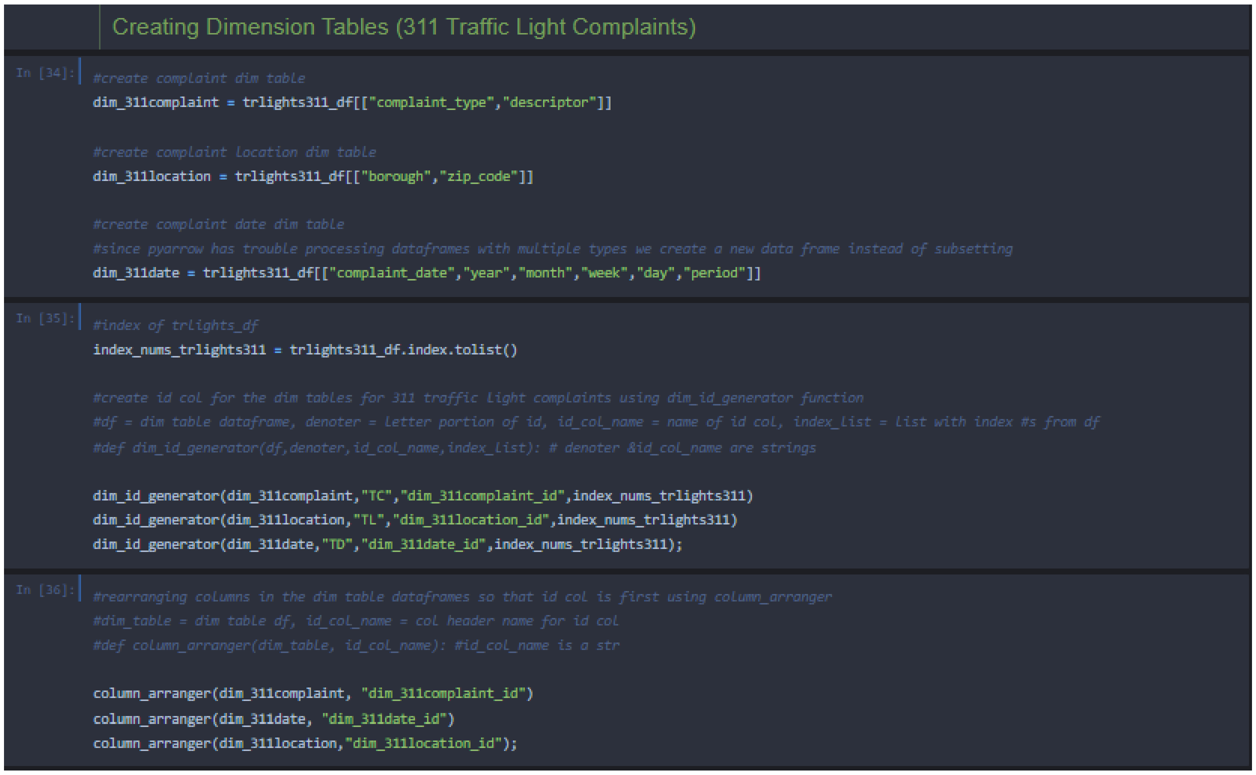
We started off the transformation by substituting missing values (NaN) with "Unspecified". We then converted the created_date column from a string to a datetime object. This was followed by us extracting the year, month, week, day, and period from the created_date column, much like how we extracted the same set of information from the date_time column with the collisions dataset. We then renamed some columns so that they would be more in line with how we wanted our dimension tables to look like.



Transforming - 311 Traffic Lights Complaints Data

```
In [27]: #replace all NaN values with "Unspecified"
         trlights311_df.fillna("Unspecified", inplace = True)

In [30]: #convert created_date col to datetime
         trlights311_df["created_date"] = pd.to_datetime(trlights311_df["created_date"], infer_datetime_format=True)

In [31]: #create Year, Month, Day, and Week columns; extract from created_date col
         trlights311_df["year"] = trlights311_df["created_date"].dt.year

         trlights311_df["month"] = trlights311_df["created_date"].dt.month
         trlights311_df["month"] = trlights311_df["month"].apply(lambda x: calendar.month_abbr[x])

         trlights311_df["week"] = trlights311_df["created_date"].dt.isocalendar().week
         trlights311_df["week"] = trlights311_df["week"].astype(int) #<---- convert dtype from UINT32 to int64; will cause issues with pyarrow if we don'

         trlights311_df["day"] = trlights311_df["created_date"].dt.day

         #create Period column; use period_converter function - apply to all rows in created_time column
         trlights311_df["period"] = trlights311_df["created_date"].apply(lambda x: period_converter(x))

In [32]: #rename incident_zip to zip_code
         trlights311_df.rename(columns = {"incident_zip":"zip_code","created_date":"complaint_date"}, inplace = True)
```

We then created the dimension tables for the 311 dataset, and generated dimension table ID columns using the same method that we used for creating the dimension tables for the collisions dataset -- by using the self-defined function dim_id_generator. This was followed by us rearranging the columns so that the dimension table ID column would be the left-most column across all the dimension tables. This was done by using the self-defined function called column_arranger.



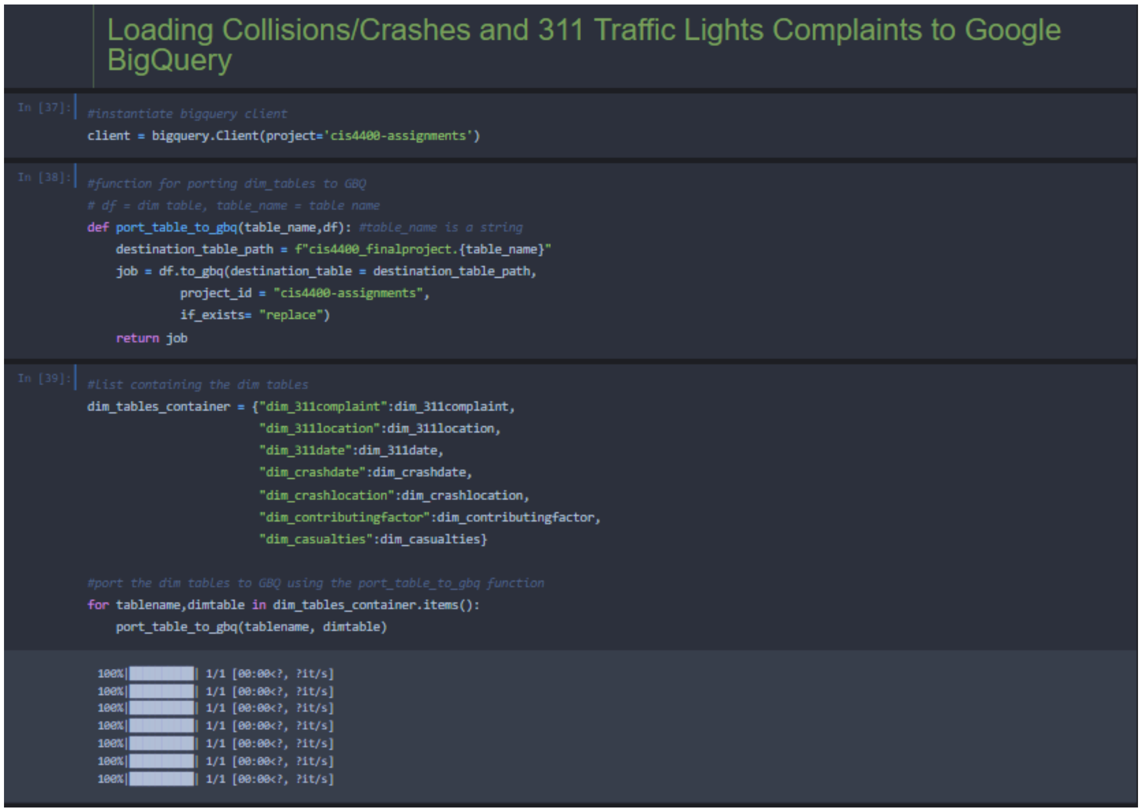
Creating Dimension Tables (311 Traffic Light Complaints)

```
In [34]: #create complaint dim table
         dim_311complaint = trlights311_df[["complaint_type","descriptor"]]

         #create complaint location dim table
         dim_311location = trlights311_df[["borough","zip_code"]]

         #create complaint date dim table
         #since pyarrow has trouble processing dataframes with multiple types we create a new data frame instead of subsetting
         dim_311date = trlights311_df[["complaint_date","year","month","week","day","period"]]
```

```
In [35]: #index of trlights_df
         index_nums_trlights311 = trlights311_df.index.tolist()

         #create id col for the dim tables for 311 traffic light complaints using dim_id_generator function
         #df = dim table dataframe, denoter = letter portion of id, id_col_name = name of id col, index_list = list with index #s from df
         #def dim_id_generator(df,denoter,id_col_name,index_list): # denoter &id_col_name are strings

         dim_id_generator(dim_311complaint,"TC","dim_311complaint_id",index_nums_trlights311)
         dim_id_generator(dim_311location,"TL","dim_311location_id",index_nums_trlights311)
         dim_id_generator(dim_311date,"TD","dim_311date_id",index_nums_trlights311);
```

```
In [36]: #rearranging columns in the dim table dataframes so that id col is first using column_arranger
         #dim_table = dim table df, id_col_name = col header name for id col
         #def column_arranger(dim_table, id_col_name): #id_col_name is a str

         column_arranger(dim_311complaint, "dim_311complaint_id")
         column_arranger(dim_311date, "dim_311date_id")
         column_arranger(dim_311location,"dim_311location_id");
```

After all of this, we then established a connection with the BigQuery API and proceeded to port the dimension tables that we created over to BigQuery.



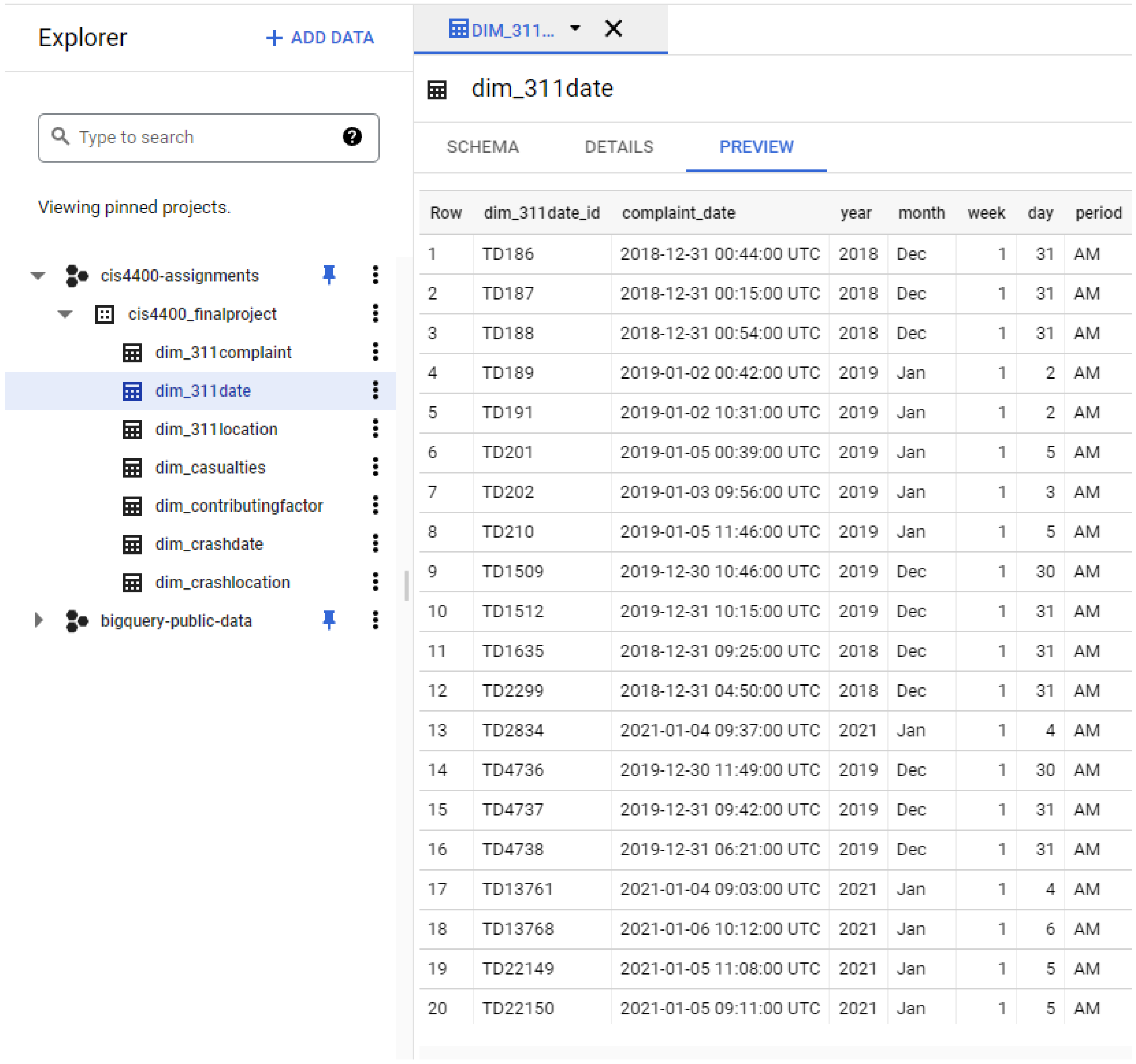
Loading Collisions/Crashes and 311 Traffic Lights Complaints to Google BigQuery

```
In [37]: #instantiate bigquery client
         client = bigquery.Client(project='cis4400-assignments')
```

```
In [38]: #function for porting dim_tables to GBQ
         # df = dim table, table_name = table name
         def port_table_to_gbq(table_name,df): #table_name is a string
             destination_table_path = f"cis4400_finalproject.{table_name}"
             job = df.to_gbq(destination_table = destination_table_path,
                     project_id = "cis4400-assignments",
                     if_exists= "replace")
             return job
```

```
In [39]: #list containing the dim tables
         dim_tables_container = {"dim_311complaint":dim_311complaint,
                                 "dim_311location":dim_311location,
                                 "dim_311date":dim_311date,
                                 "dim_crashdate":dim_crashdate,
                                 "dim_crashlocation":dim_crashlocation,
                                 "dim_contributingfactor":dim_contributingfactor,
                                 "dim_casualties":dim_casualties}

         #port the dim tables to GBQ using the port_table_to_gbq function
         for tablename,dimtable in dim_tables_container.items():
             port_table_to_gbq(tablename, dimtable)

          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
          100%|██████████| 1/1 [00:00<?, ?it/s]
```

Here's a look at the dataset - which is appropriately named as "cis4400_finalproject"- that we created in BigQuery which contains the dimension tables that we ported over via Python.

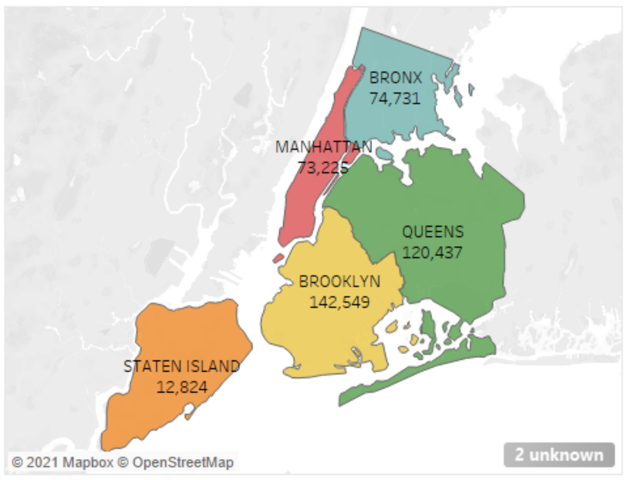


Row	dim_311date_id	complaint_date	year	month	week	day	period
1	TD186	2018-12-31 00:44:00 UTC	2018	Dec	1	31	AM
2	TD187	2018-12-31 00:15:00 UTC	2018	Dec	1	31	AM
3	TD188	2018-12-31 00:54:00 UTC	2018	Dec	1	31	AM
4	TD189	2019-01-02 00:42:00 UTC	2019	Jan	1	2	AM
5	TD191	2019-01-02 10:31:00 UTC	2019	Jan	1	2	AM
6	TD201	2019-01-05 00:39:00 UTC	2019	Jan	1	5	AM
7	TD202	2019-01-03 09:56:00 UTC	2019	Jan	1	3	AM
8	TD210	2019-01-05 11:46:00 UTC	2019	Jan	1	5	AM
9	TD1509	2019-12-30 10:46:00 UTC	2019	Dec	1	30	AM
10	TD1512	2019-12-31 10:15:00 UTC	2019	Dec	1	31	AM
11	TD1635	2018-12-31 09:25:00 UTC	2018	Dec	1	31	AM
12	TD2299	2018-12-31 04:50:00 UTC	2018	Dec	1	31	AM
13	TD2834	2021-01-04 09:37:00 UTC	2021	Jan	1	4	AM
14	TD4736	2019-12-30 11:49:00 UTC	2019	Dec	1	30	AM
15	TD4737	2019-12-31 09:42:00 UTC	2019	Dec	1	31	AM
16	TD4738	2019-12-31 06:21:00 UTC	2019	Dec	1	31	AM
17	TD13761	2021-01-04 09:03:00 UTC	2021	Jan	1	4	AM
18	TD13768	2021-01-06 10:12:00 UTC	2021	Jan	1	6	AM
19	TD22149	2021-01-05 11:08:00 UTC	2021	Jan	1	5	AM
20	TD22150	2021-01-05 09:11:00 UTC	2021	Jan	1	5	AM

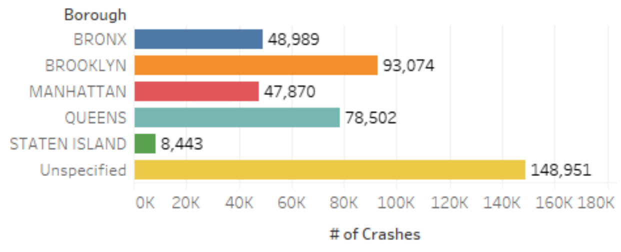
Dashboarding & Analysis

Dashboard

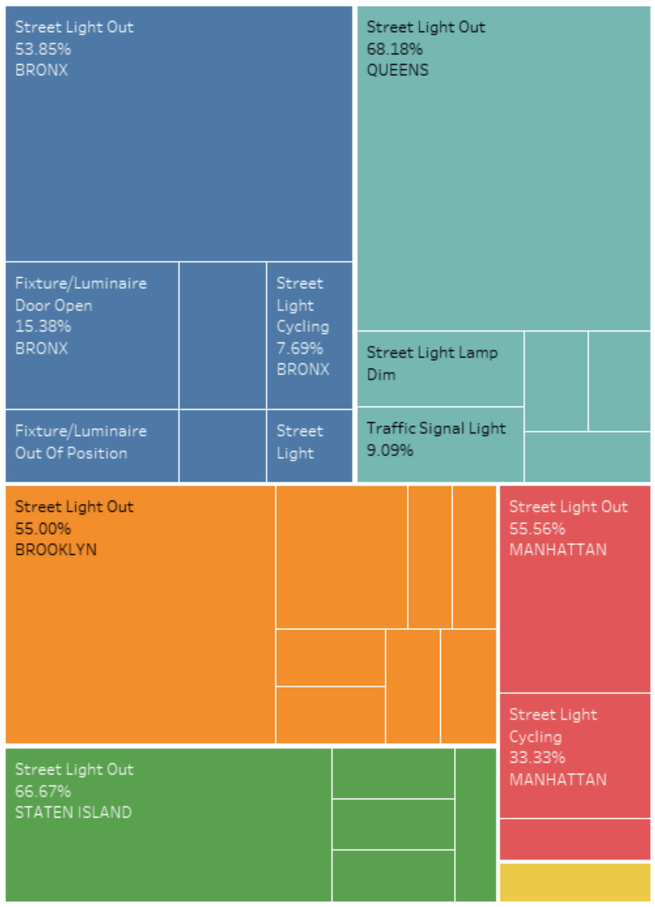
Map of Total Killed by Crashes



No. of Crashes by Borough (2019-Present)



Most Common 311 Complaints by Borough



Summary: Based on the dashboard above, which was created via Tableau, we were able to draw some conclusions:

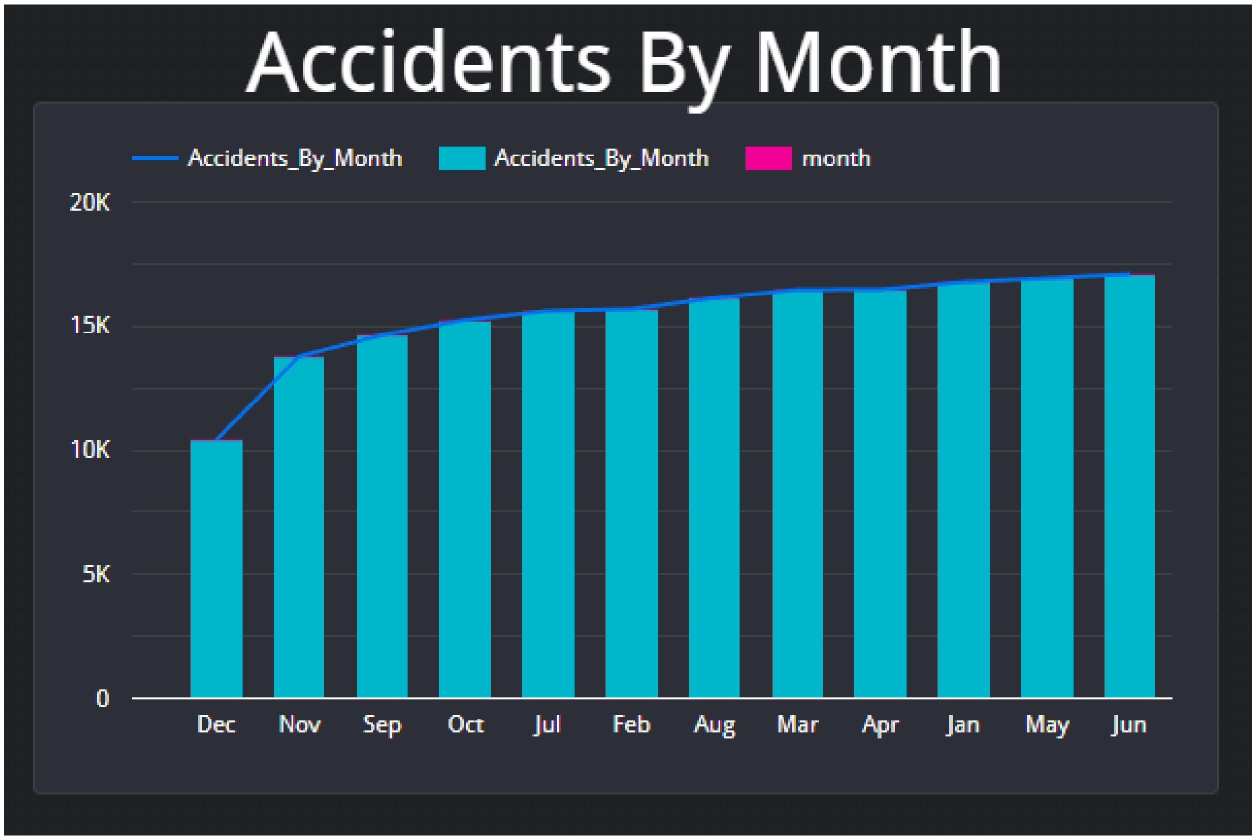
- An overwhelming majority of deaths caused by car crashes occurred in Queens and in Brooklyn. On the other hand, Staten Island had by far the least amount of deaths compared to the rest of the 4 boroughs. (see *Map of Total Killed by Crashes)*
- The number of crashes by borough seemed to be proportional to the number of people killed by car crashes per borough. With that said, Queens and Brooklyn had the most number of crashes by borough, while Staten Island had the least. However, these numbers

need to be taken with some skepticism since there are about 150,000 records where the Borough field in the crash reports was either left blank or had "Unspecified" inputted for the field. (see *No. of Crashes by Borough)*

- The most common 311 complaint was that the street lights at a specific intersection were out of commission - that is to say that the street lights were not working. The street lights being out of commission accounted for more than 50% of the complaints made to NYC311 across all boroughs. It was also interesting to see that street light complaints for cycling lights accounted for a third (⅓) of all NYC311 complaints made in Manhattan despite Brooklyn and Queens having their fare share of large bike boulevards and bike lanes.

Additional Visuals

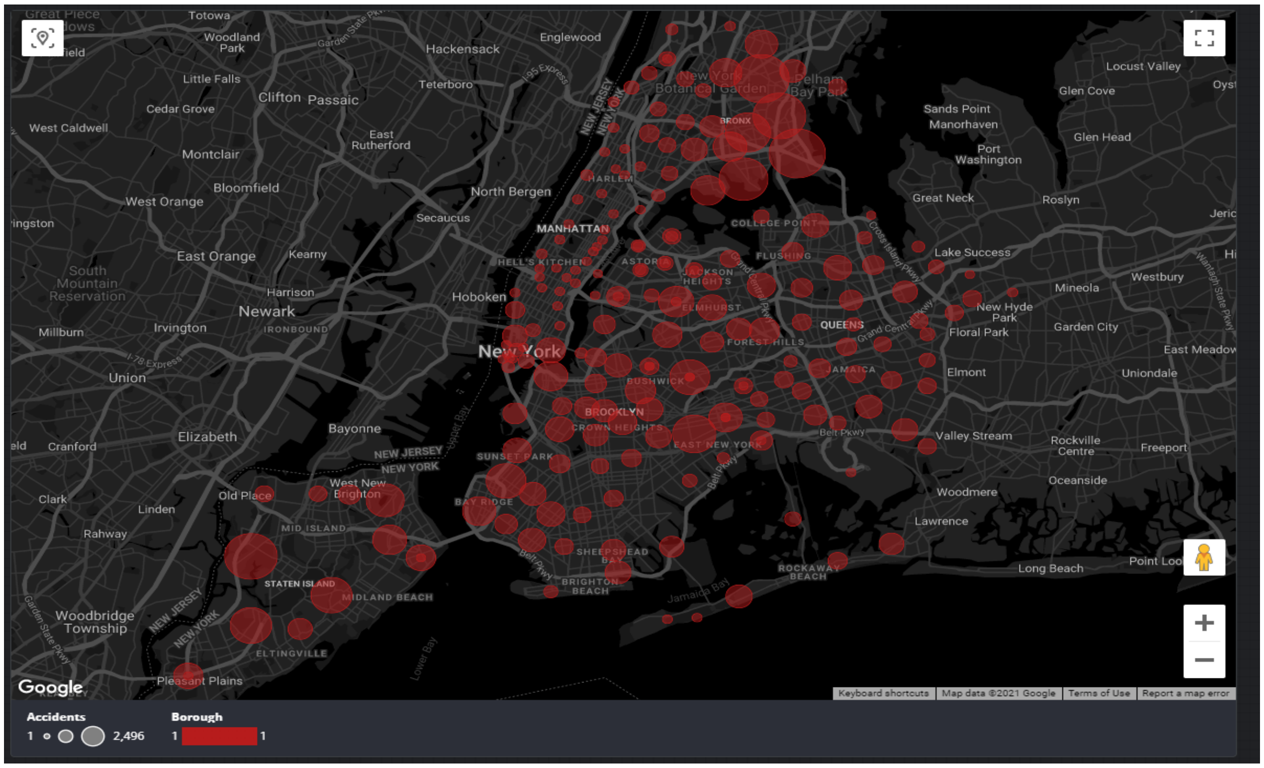
❖ Accidents by Month in NYC by month from 2019 - Present

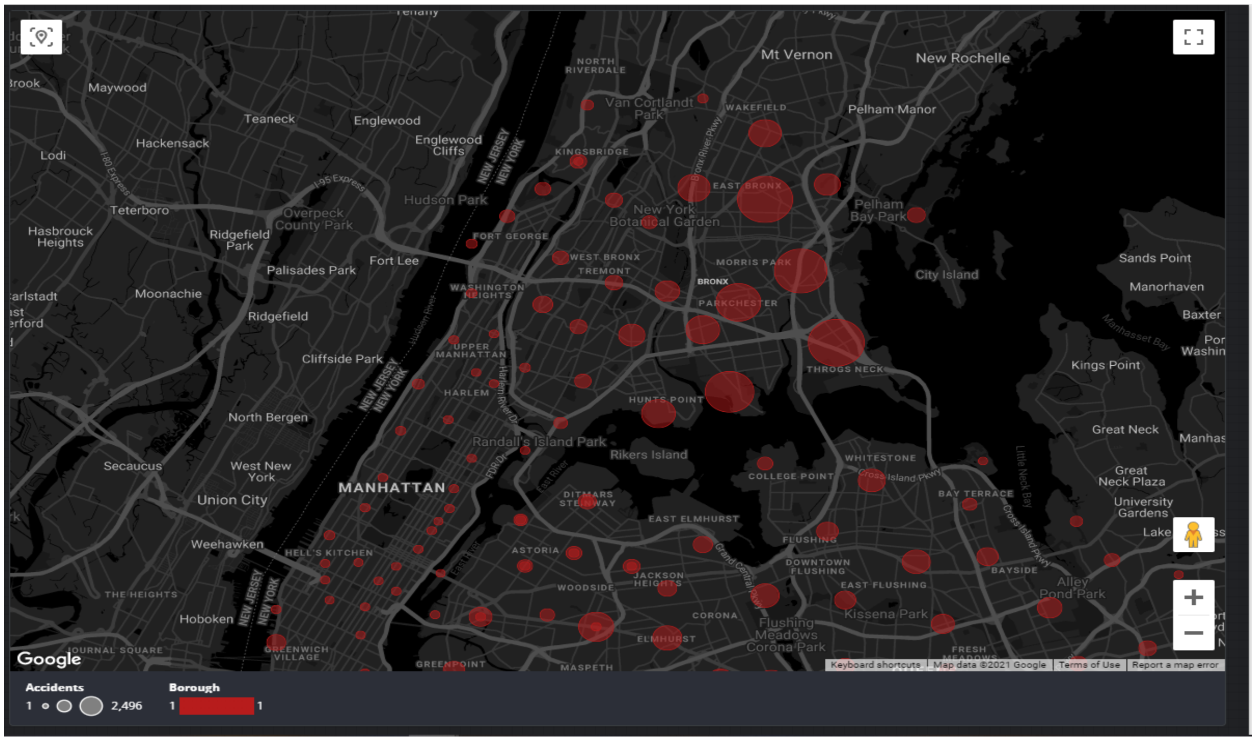


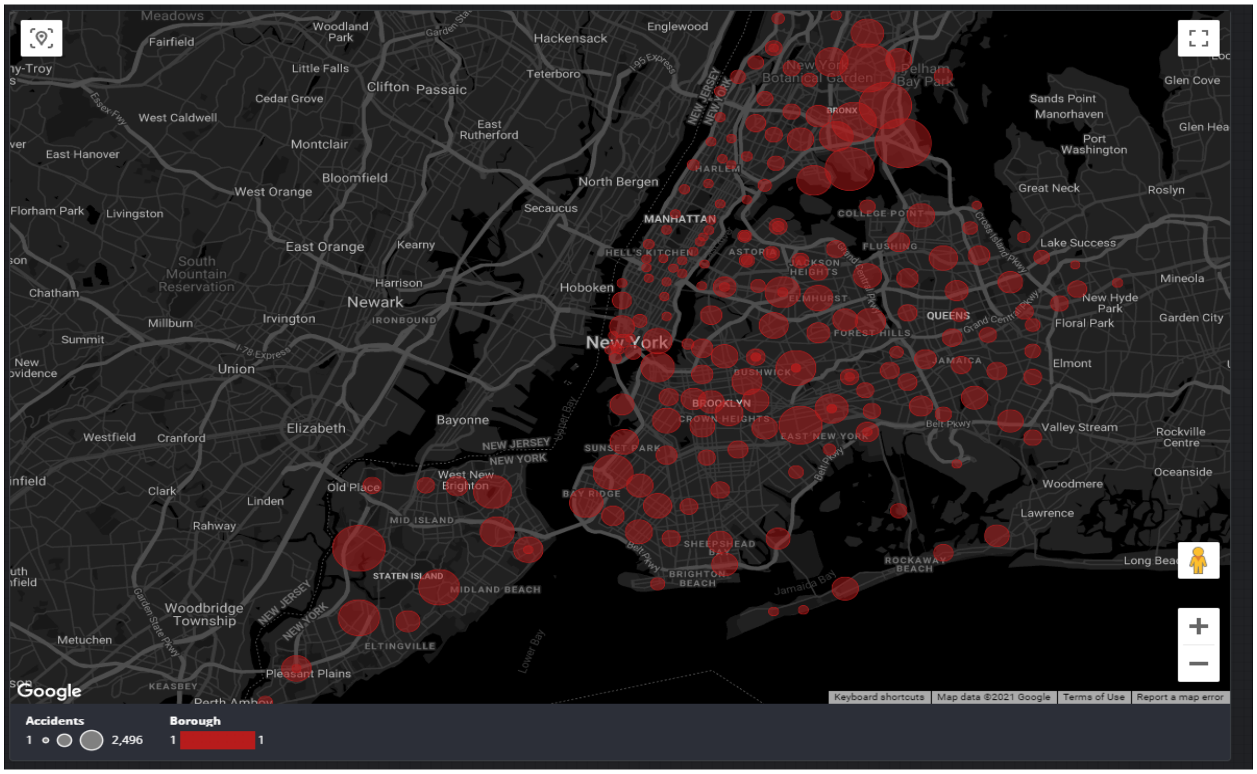
Summary: Here we can see the number of accidents in NYC from 2019 until today aggregated by months. Going by this chart, it appears that the least amount of accidents occurred during the month of December. June, on the other hand, is the month with the most accidents during the given time frame.

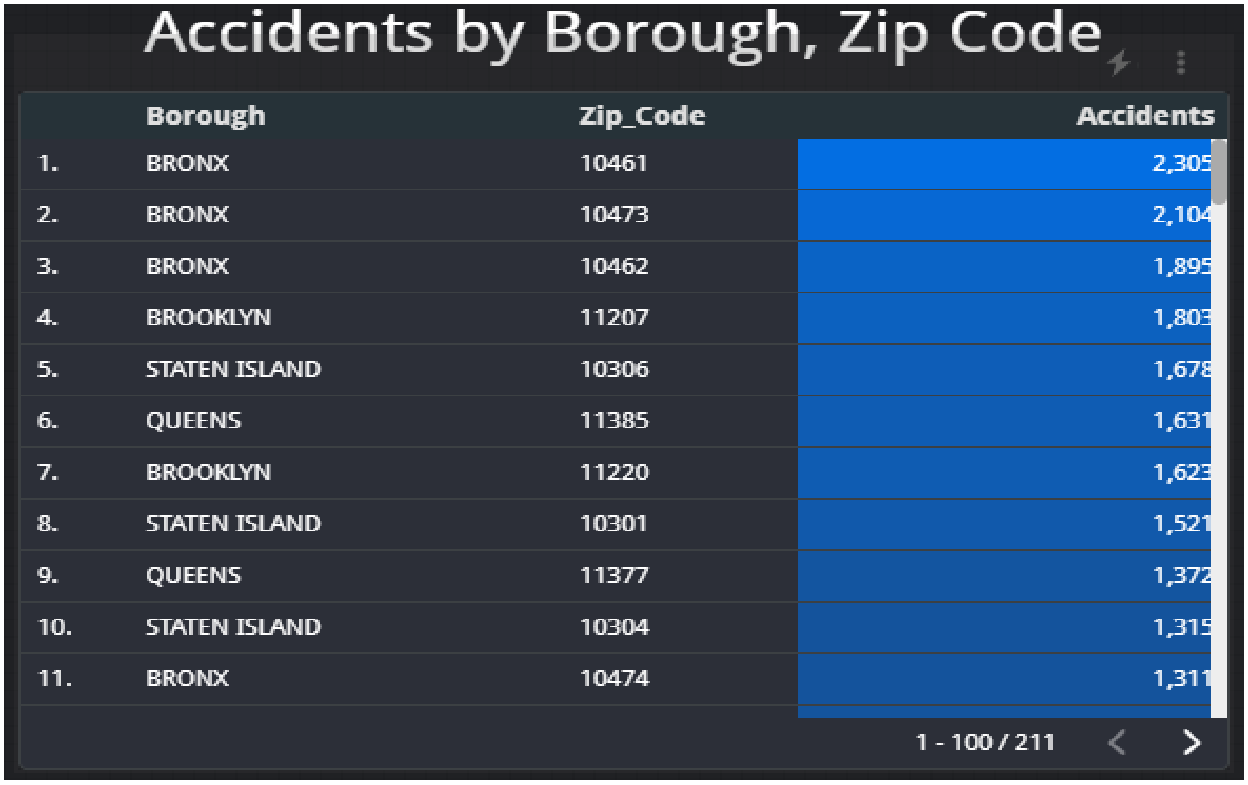
```
SELECT month, week, COUNT(*) as Accidents,
FROM `cis4400-project-334118.DATE.DATE`
GROUP BY month, week
ORDER BY COUNT(*) DESC;
```

❖ Count of Accidents by Zip Code




Accidents
1 ● ● ● 2,496
Borough
1 🟥 1


Accidents
1 ● ● ● 2,496
Borough
1 🟥 1



Accidents by Borough, Zip Code

	Borough	Zip_Code	Accidents
1.	BRONX	10461	2,305
2.	BRONX	10473	2,104
3.	BRONX	10462	1,895
4.	BROOKLYN	11207	1,803
5.	STATEN ISLAND	10306	1,678
6.	QUEENS	11385	1,631
7.	BROOKLYN	11220	1,623
8.	STATEN ISLAND	10301	1,521
9.	QUEENS	11377	1,372
10.	STATEN ISLAND	10304	1,315
11.	BRONX	10474	1,311

1 - 100 / 211

```
SELECT string_field_2 as Borough,string_field_3 as Zip_Code, COUNT(*) as Accidents
FROM `cis4400-project-334118.Accidents.Accidents By Borough`
WHERE string_field_3 != 'Unspecified'
GROUP BY Borough, Zip_Code
ORDER BY COUNT(*) DESC;
```

Summary: Here we are looking to visualize accidents by zip code. Above is the corresponding query that was created to get the distinct count of accidents by first the borough, then the zip code of each record. After creating the query, we transformed it into a table and created a map chart via Data Studio.

After Analyzing the chart via a google map visual, it is clear that there is a significant number of accidents around the eastern part of the Bronx, including: Parkchester, Pelham Bay, Castle Hill and Throgs Neck. Staten Island also appears to have a large number of accidents, especially in the Travis-Chelsea area. Brooklyn appears to have a large number of accidents where it is most heavily populated including areas such as Bushwick and New Lots.

Narrative Conclusion Section

I. Describe the software and database tools the group used to coordinate and manage the project as well as carry out the programming tasks (list of bullet points with software or service and one sentence of what it was used for).

The group used WhatsApp as our primary channel of communication. As for the project itself, we used:

- **LucidChart** for drawing up the dimensional models.
- **Google Drive/Docs** for creating the written deliverables.
- **GitHub** to host and share the code and other materials such as csv backups for the cleaned datasets.
- **Python** and the **Socrata API** for the ETL process.
- **Google BigQuery** for warehousing the data/dimensional tables.
- **Tableau** and **Google Data Studio** for performing ad-hoc analysis and creating dashboards to briefly go over the KPIs that we defined early on in the project.

II. Describe the group's experience with the project (which steps were the most difficult? Which were the easiest? what did you learn that you did not imagine you would have? if you had to do it all over again, what would you have done differently?)

The group had a few difficulties in finding and deciding on a version of the dimensional model to move forward with. Other than the headache of dealing with datetime objects and properly reformatting them via Python during the transformation portion of the ETL process, the rest of the project was fairly straight-forward for us. As a whole, we feel like there was a really valuable lesson learned in how we could establish an ETL pipeline even on a relatively small scale compared to the ones used by companies. A collection of concepts and processes that once seemed like a task similar to reinventing the wheel now just feels like just another project. If we had to do it again, we feel like we should have given other options available for us such as Pentaho and Data Flow.

III. Describe if the proposed benefits can be realized by the new system.

By using ETL tools such as Pentaho, Data Flow, or dbt our group definitely would have had an easier time with the ETL process since those platforms/tools come with the advantage of just having to interact with GUI to get the ETL done as opposed to manually coding everything out. Though, on the bright side, our decision to use Python helped us learn how to code out an entire ETL process using Python.