EMERITUS
INSTITUTE OF MANAGEMENT

WEEK 13

DATA ANALYSIS AND VISUALIZATION — USING PYTHON'S PANDAS FOR DATA WRANGLING

COLUMBIA | ENGINEERING
EXECUTIVE EDUCATION

Data Cleaning with Pandas - A

Visualize and analyze data using Panda

```
In [ ]: datafile = "nyc_311_data_subset.csv"
```

```
In [ ]: import pandas as pd
        import numpy as np
```

read_csv: A pandas function that reads a comma separated file

read_csv will try to format the data so that it is the correct type and will report any typing problems

It will also look for a header row.

http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

```
In [ ]: data = pd.read_csv(datafile)
        data
```

Let's examine our data

```
In [ ]: data.info()
```

Looks like Unique Key really is a unique key and can serve as an index

```
In [ ]: data = pd.read_csv(datafile,index_col='Unique Key')
```

```
In [ ]: data.iloc[1:10]
```

Data Cleaning with Pandas - B

Data contains several columns, also known as Panda data frame-

```
In [3]: data = pd.read_csv(datafile)
        data

/Users/cvn-mm-pbs-001/anaconda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2717: DtypeWarning: Colum
ns (4) have mixed types. Specify dtype option on import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)

Out[3]:
```

	Unique Key	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude
0	33136109	10/11/2016 11:53:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
1	33137323	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
2	33139057	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
3	33140865	10/11/2016 12:39:00 PM	10/11/2016 12:39:00 PM	DSNY	NaN	QUEENS	NaN	NaN
4	33141225	10/11/2016 12:18:00 PM	10/11/2016 12:18:00 PM	DSNY	NaN	QUEENS	NaN	NaN
5	33141715	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
6	33141787	10/11/2016 12:39:00 PM	10/11/2016 12:39:00 PM	DSNY	NaN	QUEENS	NaN	NaN
7	33141934	10/11/2016 11:44:00 AM	NaN	DSNY	NaN	QUEENS	NaN	NaN

The 'data.info()' command indicates the structure of data file/Panda frame.

Let's examine our data

```
In [4]: data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 971063 entries, 0 to 971062
Data columns (total 8 columns):
Unique Key       971063 non-null int64
Created Date     971063 non-null object
Closed Date      882944 non-null object
Agency           971063 non-null object
Incident Zip     911140 non-null object
Borough          971063 non-null object
Latitude         887284 non-null float64
Longitude        887284 non-null float64
dtypes: float64(2), int64(1), object(5)
memory usage: 59.3+ MB
```

There are 971,000 records of 'Unique Key'

Data Cleaning with Pandas - C

Use first ten records

```
In [6]: data.iloc[1:10]
```

Out[6]:

	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude
Unique Key							
33137323	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33139057	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33140865	10/11/2016 12:39:00 PM	10/11/2016 12:39:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33141225	10/11/2016 12:18:00 PM	10/11/2016 12:18:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33141715	10/11/2016 11:36:00 AM	10/11/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33141787	10/11/2016 12:39:00 PM	10/11/2016 12:39:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33141934	10/11/2016 11:44:00 AM	NaN	DSNY	NaN	QUEENS	NaN	NaN
33142524	10/11/2016 12:35:00 PM	10/11/2016 12:35:00 PM	DSNY	NaN	QUEENS	NaN	NaN
33142733	10/11/2016 11:26:00 AM	05/27/2016 12:00:00 PM	DSNY	NaN	QUEENS	NaN	NaN

Note -

```
In [5]: data = pd.read_csv(datafile,index_col='Unique Key')

/Users/cvn-mm-pbs-001/anaconda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2717: DtypeWarning: Colum
ns (4) have mixed types. Specify dtype option on import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

If we want to do analysis, we want to use an NP array kind of structure, i.e. data of same kind.

Data Cleaning with Pandas - D

A function called 'unique' can be used to pull out unique values in a column.

Columns 4 has mixed types

Column 4 is incident zip

Let's examine it

The unique() function returns unique values in a column

```
In [7]: data['Incident Zip'].unique()

Out[7]: array([nan, '10001', '11691', '11211', '10027', '10452', '11428', '11101',
               '10075', '11215', '11210', '11231', '11217', '10457', '10033',
               '11209', '11201', '11367', '10029', '10021', '10028', '10034',
               '10032', '10039', '11414', '10461', '11229', '10462', '11223',
               '10023', '10453', '11225', '11219', '10451', '11234', '10014',
               '11354', '11361', '10468', '11233', '10466', '11204', '11413',
               '11224', '11375', '11040', '11232', '11203', '11205', '11434',
               '10011', '10003', '10025', '10013', '10036', '11237', '11355',
               '11368', '10454', '10456', '10463', '11222', '11228', '11216',
               '10128', '11435', '11419', '11358', '11421', '10019', '11238',
               '11213', '11235', '11420', '10038', '11226', '10472', '10016',
               '11221', '11236', '11436', '11214', '11377', '11385', '11365',
               '10312', '11426', '11373', '11218', '10005', '11230', '10026',
               '10473', '10280', '10301', '10309', '10310', '10009', '10002',
               '11433', '10020', '11357', '10030', '11378', '11249', '11432',
               '11212', '10024', '10035', '11429', '11206', '11372', '10471',
               '10119', '10307', '11364', '11103', '10017', '10012', '11105',
               '10458', '10018', '11374', '10459', '10314', '10037', '10302',
```

Some issues

- Sometimes zip is a float, other times it is a str
- Zipcodes that are represented as floats and start with 0 are missing the first digit
- Some zipcodes have the 4 digit extension added. Comparison becomes tough
- What the heck is zip 0?
- What about the missing (nan) values? The ? (question mark)? "UNKNOWN"?

Data Cleaning with Pandas: Step One

The first step in data cleaning is to:

Decide what to do with "bad" data ("JFK", "UNKNOWN", etc.). Convert to Nan or delete the record.

Make sure all data in a column is in the correct format (convert floats to strings, get rid of the 4 digit extension)

Decide what to do with missing values (NaNs)

for "Incident Zip"

we'll drop rows with NaN or bad data

get rid of the 4 digit extension

remove zips less than 10000 and greater than 19999

Let's write a function that fixes zips

Data Cleaning with Pandas: Step One

Use a function called 'fix_zip’ to take the zip code in whatever format.

```
In [ ]:

In [ ]: def fix_zip(input_zip):
            try:
                input_zip = int(float(input_zip))
            except:
                try:
                    input_zip = int(input_zip.split('-')[0])
                except:
                    return np.NaN
            if input_zip < 10000 or input_zip > 19999:
                return np.NaN
            return str(input_zip)
```

And test it

```
In [9]: fix_zip('11211.00')
Out[9]: '11211'
```

And test it

```
In [11]: fix_zip('UNKNOWN')
Out[11]: nan
```

Data Cleaning with Pandas: Step Two

Next, we'll apply this function to every element in input zip to get a revised column

The pandas function "apply" applies a function to a dataframe column
- **fix_zip will be applied to each element of the Incident Zip column and we replace the existing column with the modified one**

```
In [12]: data['Incident Zip']

Out[12]: Unique Key
         33136109         NaN
         33137323         NaN
         33139057         NaN
         33140865         NaN
         33141225         NaN
         33141715         NaN
         33141787         NaN
         33141934         NaN
         33142524         NaN
         33142733         NaN
         34215673       10001
         34219052       11691
         34219145       11211
         34219385       10027
         34219399       10452
         34219470       11691
         34219513       11428
         34219516       11101
         34219534       10075
         34219623       11215
         34219638       11101
         34219639       11210
         34219640       11231
         34219643       11210
```

```
In [12]: data['Incident Zip'].apply(fix_zip)

Out[12]: Unique Key
         33136109         NaN
         33137323         NaN
         33139057         NaN
         33140865         NaN
         33141225         NaN
         33141715         NaN
         33141787         NaN
         33141934         NaN
         33142524         NaN
         33142733         NaN
         34215673       10001
         34219052       11691
         34219145       11211
         34219385       10027
```

Data Cleaning with Pandas: Step Two

```
In [14]: data['Incident Zip'] = data['Incident Zip'].apply(fix_zip)

In [15]: data['Incident Zip'].unique()

Out[15]: array([nan, '10001', '11691', '11211', '10027', '10452', '11428', '11101',
                '10075', '11215', '11210', '11231', '11217', '10457', '10033',
                '11209', '11201', '11367', '10029', '10021', '10028', '10034',
                '10032', '10039', '11414', '10461', '11229', '10462', '11223',
                '10023', '10453', '11225', '11219', '10451', '11234', '10014',
                '11354', '11361', '10468', '11233', '10466', '11204', '11413',
                '11224', '11375', '11040', '11232', '11203', '11205', '11434',
                '10011', '10003', '10025', '10013', '10036', '11237', '11355',
                '11368', '10454', '10456', '10463', '11222', '11228', '11216',
                '10128', '11435', '11419', '11358', '11421', '10019', '11238',
                '11213', '11235', '11420', '10038', '11226', '10472', '10016',
                '11221', '11236', '11436', '11214', '11377', '11385', '11365',
                '10312', '11426', '11373', '11218', '10005', '11230', '10026',
                '10473', '10280', '10301', '10309', '10310', '10009', '10002',
                '11433', '10020', '11357', '10030', '11378', '11249', '11432',
                '11212', '10024', '10035', '11429', '11206', '11372', '10471',
                '10119', '10307', '11364', '11103', '10017', '10012', '11105',
                '10458', '10018', '11374', '10459', '10314', '10037', '10302',
                '10040', '11411', '11692', '10303', '11418', '10031', '11220',
                '11427', '10465', '10306', '10010', '10460', '10305', '11207',
                '11208', '10474', '11417', '10475', '10455', '11416', '10065',
                '11363', '11693', '10308', '11356', '10469', '11369', '10470',
                '10467', '10007', '10304', '11366', '11694', '11102', '11423',
                '11422', '19044', '11412', '10022', '11379', '11251', '11004',
                '11104', '10004', '11362', '11360', '11109', '11590', '11001',
                '11430', '11106', '10464', '11370', '10271', '11239', '11415',
```

Data Cleaning with Pandas: Final Step

Finally, we'll get rid of all rows that have zip == Nan

- We don't have to, that's just a choice we're making

```
In [16]: data = data[data['Incident Zip'].notnull()]
```

```
In [17]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 910907 entries, 34215673 to 34368845
Data columns (total 7 columns):
Created Date     910907 non-null object
Closed Date      829453 non-null object
Agency           910907 non-null object
Incident Zip     910907 non-null object
Borough          910907 non-null object
Latitude         887168 non-null float64
Longitude        887168 non-null float64
dtypes: float64(2), object(5)
memory usage: 55.6+ MB
```

Compare Table After Data Clean-Up

Let's examine our data

In [4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 971063 entries, 0 to 971062
Data columns (total 8 columns):
Unique Key       971063 non-null int64
Created Date     971063 non-null object
Closed Date      882944 non-null object
Agency           971063 non-null object
Incident Zip     911140 non-null object
Borough          971063 non-null object
Latitude         887284 non-null float64
Longitude        887284 non-null float64
dtypes: float64(2), int64(1), object(5)
memory usage: 59.3+ MB
```

In [17]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 910907 entries, 34215673 to 34368845
Data columns (total 7 columns):
Created Date     910907 non-null object
Closed Date      829453 non-null object
Agency           910907 non-null object
Incident Zip     910907 non-null object
Borough          910907 non-null object
Latitude         887168 non-null float64
Longitude        887168 non-null float64
dtypes: float64(2), object(5)
memory usage: 55.6+ MB
```

Closed Data, Latitude, and Longitude

Let's get rid of them

```
In [19]: data = data[(data['Latitude'].notnull()) & (data['Longitude'].notnull())  & (data['Closed Date'].notnull())]
```

```
In [20]: data.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 806561 entries, 34215673 to 34368845
Data columns (total 7 columns):
Created Date    806561 non-null object
Closed Date     806561 non-null object
Agency          806561 non-null object
Incident Zip    806561 non-null object
Borough         806561 non-null object
Latitude        806561 non-null float64
Longitude       806561 non-null float64
dtypes: float64(2), object(5)
memory usage: 49.2+ MB
```

Let's take a look at Borough data

```
In [21]: data['Borough'].unique()

Out[21]: array(['MANHATTAN', 'QUEENS', 'BROOKLYN', 'BRONX', 'STATEN ISLAND',
                'Unspecified'], dtype=object)
```

Closed Data, Latitude, and Longitude

Let's look at 'Unspecified'

```
In [22]: data[data['Borough']=='Unspecified'][['Agency','Incident
```

Out[22]:

	Agency	Incident Zip
Unique Key		
35281310	NYPD	10312
35287553	NYPD	11368
35288108	NYPD	11422
35288996	NYPD	10454
35280674	NYPD	11209
35280721	NYPD	11226
35281352	NYPD	11225
35281873	NYPD	11373
35282130	NYPD	10467

Closer Look

```
In [23]: data[data['Borough']=='Unspecified'].groupby('Agency').count()
```

Out[23]:

	Created Date	Closed Date	Incident Zip	Borough	Latitude	Longitude
Agency						
3-1-1	1	1	1	1	1	1
DHS	67	67	67	67	67	67
DOE	1	1	1	1	1	1
DOF	3	3	3	3	3	3
DOT	13	13	13	13	13	13
DPR	2	2	2	2	2	2
FDNY	1	1	1	1	1	1
NYPD	725	725	725	725	725	725
TLC	6	6	6	6	6	6

We found lot of NYPD entries

Closed Data, Latitude, and Longitude

Unspecified appears to have a systematic bias toward NYPD

Though only a small proportion of NYPD complaints (see below)

We have to decide whether to keep them or lose them!

```
In [24]: nypd_complaints_total = data[data['Agency']=='NYPD']['Borough'].count()
         #nypd_unspecified = data[(data['Borough']=='Unspecified') & (data['Agency']=="NYPD")]['Borough'].count()
         #percentage = nypd_unspecified/nypd_complaints_total*100
         #print("%1.2f"%percentage)

In [25]: nypd_complaints_total

Out[25]: 274408
```

We have to decide whether to keep them or lose them!

```
In [26]: nypd_complaints_total = data[data['Agency']=='NYPD']['Borough'].count()
         nypd_unspecified = data[(data['Borough']=='Unspecified') & (data['Agency']=="NYPD")]['Borough'].count()
         #percentage = nypd_unspecified/nypd_complaints_total*100
         #print("%1.2f"%percentage)

In [27]: nypd_unspecified

Out[27]: 725

In [28]: nypd_complaints_total = data[data['Agency']=='NYPD']['Borough'].count()
         nypd_unspecified = data[(data['Borough']=='Unspecified') & (data['Agency']=="NYPD")]['Borough'].count()
         percentage = nypd_unspecified/nypd_complaints_total*100
         print("%1.2f"%percentage)

         0.26
```

Closed Data, Latitude, and Longitude

For now, we'll get rid of them. Unspecified will be hard to explain!

```
In [29]: data = data[data['Borough'] != 'Unspecified']
```

```
In [30]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 805742 entries, 34215673 to 34368845
Data columns (total 7 columns):
Created Date    805742 non-null object
Closed Date     805742 non-null object
Agency          805742 non-null object
Incident Zip    805742 non-null object
Borough         805742 non-null object
Latitude        805742 non-null float64
Longitude       805742 non-null float64
dtypes: float64(2), object(5)
memory usage: 49.2+ MB
```

Closed Data, Latitude, and Longitude

Dealing with time

- Dates and times are best converted to datetime
- That way they will be useful for analysis because we can compute timedelta objects

The aim is to convert all the strings into a datetime object.

```
In [ ]: import datetime
        data['Created Date'] = data['Created Date'].apply(lambda x:datetime.datetime.strptime(x,'%m/%d/%Y %I:%M:%S %p'))
```

```
Out[34]: Unique Key
         34215673    2016-09-01 00:33:42
         34219052    2016-09-01 20:16:24
         34219145    2016-09-01 12:17:00
         34219385    2016-09-01 12:10:22
         34219399    2016-09-01 12:32:32
         34219470    2016-09-01 20:16:24
         34219513    2016-09-01 08:35:00
         34219516    2016-09-01 13:19:42
         34219534    2016-09-01 11:00:00
         34219623    2016-09-01 11:45:00
         34219638    2016-09-01 10:11:45
         34219639    2016-09-01 08:22:53
         34219640    2016-09-01 17:31:04
         34219643    2016-09-01 08:50:41
         34219644    2016-09-01 14:19:21
         34219646    2016-09-01 12:46:35
         34219681    2016-09-01 13:33:58
         34219813    2016-09-01 13:51:10
```

Complaints Closure – Turnaround Time

Out[37]:

	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude
Unique Key							
34215673	2016-09-01 00:33:42	2016-09-16 01:06:56	DCA	10001	MANHATTAN	40.744790	-73.988834
34219052	2016-09-01 20:16:24	2016-09-10 18:08:25	HPD	11691	QUEENS	40.600554	-73.750704
34219145	2016-09-01 12:17:00	2016-09-07 12:00:00	DSNY	11211	BROOKLYN	40.704925	-73.962007
34219385	2016-09-01 12:10:22	2016-09-10 14:23:44	HPD	10027	MANHATTAN	40.812322	-73.955338
34219399	2016-09-01 12:32:32	2016-09-11 02:03:37	HPD	10452	BRONX	40.839529	-73.922534
34219470	2016-09-01 20:16:24	2016-09-10 18:08:24	HPD	11691	QUEENS	40.600554	-73.750704
34219513	2016-09-01 08:35:00	2016-09-07 12:00:00	DSNY	11428	QUEENS	40.721866	-73.745982
34219516	2016-09-01 13:19:42	2016-09-16 14:32:35	DOT	11101	QUEENS	40.746875	-73.952711
34219534	2016-09-01 11:00:00	2016-09-08 12:00:00	DSNY	10075	MANHATTAN	40.773336	-73.955054

In [38]:

```
data['processing_time'] =  data['Closed Date'] - data['Created Date']
```

In [39]:

```
#And look at summary statistics
data['processing_time'].describe()
```

Out[39]:

```
count                         805742
mean       5 days 00:05:11.538976
std       12 days 06:08:17.201098
min           -134 days +00:00:00
25%               0 days 02:34:46
50%        0 days 21:10:44.500000
75%        4 days 14:29:59.750000
max             148 days 13:10:54
Name: processing_time, dtype: object
```

Complaints Closure – Turnaround Time

There is some odd stuff here

- Negative processing time?
- Since our data is for two months, a max of 148 days worth checking out

Let's examine the negative processing time data

```
In [40]: data[data['processing_time']<datetime.timedelta(0,0,0)]
```

34339796	2016-09-16 14:24:00	2016-09-15 14:23:00	DOT	10314	STATEN ISLAND	40.597868	-74.140537
34367448	2016-09-20 14:03:00	2016-09-16 14:03:00	DOT	11220	BROOKLYN	40.630682	-74.010970
34580456	2016-10-20 11:24:00	2016-10-19 01:24:00	DOT	11412	QUEENS	40.696186	-73.751966
34580514	2016-10-20 16:40:00	2016-10-19 16:39:00	DOT	10306	STATEN ISLAND	40.580343	-74.103262
34580724	2016-10-20 12:19:00	2016-10-19 12:18:00	DOT	11209	BROOKLYN	40.634865	-74.026381
34582178	2016-10-20 12:05:00	2016-10-19 02:05:00	DOT	11208	BROOKLYN	40.681095	-73.873586
34612455	2016-10-24 10:37:00	2016-10-21 10:37:00	DOT	11691	QUEENS	40.608713	-73.747670
34669594	2016-10-31 10:26:00	2016-10-28 10:26:00	DOT	11417	QUEENS	40.676871	-73.840344
34671873	2016-10-31 10:46:00	2016-10-27 10:46:00	DOT	11362	QUEENS	40.765202	-73.738088
34360609	2016-09-20 11:49:00	2016-09-16 11:49:00	DOT	11432	QUEENS	40.703220	-73.802559
34360615	2016-09-20 14:16:00	2016-09-16 14:16:00	DOT	11238	BROOKLYN	40.680797	-73.958397

And the large processing times as well

```
In [41]: data[data['processing_time']>datetime.timedelta(148,0,0)]
```

Out[41]:

	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude	processing_time
Unique Key								
34220964	2016-09-01 10:49:06	2017-01-28 00:00:00	DOB	11691	QUEENS	40.597741	-73.775975	148 days 13:10:54
34222594	2016-09-01 09:04:14	2017-01-27 14:12:22	DOT	11357	QUEENS	40.791344	-73.827361	148 days 05:08:08

Function Incorporating All the Changes

Though it sounds trivial, incorporating all the changes is very important step, the data will be used several times.

```
In [ ]: def read_311_data(datafile):
            import pandas as pd
            import numpy as np
            #Add the fix_zip function
            def fix_zip(input_zip):
                try:
                    input_zip = int(float(input_zip))
                except:
                    try:
                        input_zip = int(input_zip.split('-')[0])
                    except:
                        return np.NaN
                if input_zip < 10000 or input_zip > 19999:
                    return np.NaN
                return str(input_zip)

            #Read the file
            df = pd.read_csv(datafile,index_col='Unique Key')

            #fix the zip
            df['Incident Zip'] = df['Incident Zip'].apply(fix_zip)

            #drop all rows that have any nans in them (note the easier syntax!)

            df = df.dropna(how='any')

            #get rid of unspecified boroughs
            df = df[df['Borough'] != 'Unspecified']

            #Convert times to datetime and create a processing time column

            import datetime
```

```
import datetime
df['Created Date'] = df['Created Date'].apply(lambda x:datetime.datetime.strptime(x,'%m/%d/%Y %I:%M:%S %p'))
df['Closed Date'] = df['Closed Date'].apply(lambda x:datetime.datetime.strptime(x,'%m/%d/%Y %I:%M:%S %p')
df['processing_time'] =  df['Closed Date'] - df['Created Date']
```

Function Incorporating All the Changes

```
In [45]: df = read_311_data('nyc_311_data_subset.csv')
         df.info()

/Users/cvn-mm-pbs-001/anaconda/lib/python3.6/site-packages/IPython/core/interactiveshell.py:2821: DtypeWarning: Colum
ns (4) have mixed types. Specify dtype option on import or set low_memory=False.
  if self.run_code(code, result):

<class 'pandas.core.frame.DataFrame'>
Int64Index: 799323 entries, 34215673 to 34368845
Data columns (total 8 columns):
Created Date        799323 non-null datetime64[ns]
Closed Date         799323 non-null datetime64[ns]
Agency              799323 non-null object
Incident Zip        799323 non-null object
Borough             799323 non-null object
Latitude            799323 non-null float64
Longitude           799323 non-null float64
processing_time     799323 non-null timedelta64[ns]
dtypes: datetime64[ns](2), float64(2), object(3), timedelta64[ns](1)
memory usage: 54.9+ MB
```

The idea there is that we take data, look at it, examine it in as much detail as possible, and that really means getting down and looking at the actual values, looking at what the data is telling us in terms of what it contains, and then trying to figure out what kinds of anomalies. We're finding in the data anything that looks problematic and then removing it but is not going to bias our results; that's our goal with the data cleaning process.

Visualizing Data: Reading the Data Files

The idea is to look at the actual values in data and analyze anomalies in it. That's our goal though the data cleaning process removes problematic data without biasing results.

```
def read_311_data(datafile):
    import pandas as pd
    import numpy as np

    #Add the fix_zip function
    def fix_zip(input_zip):
        try:
            input_zip = int(float(input_zip))
        except:
            try:
                input_zip = int(input_zip.split('-')[0])
            except:
                return np.NaN
        if input_zip < 10000 or input_zip > 19999:
            return np.NaN
        return str(input_zip)

    #Read the file
    df = pd.read_csv(datafile,index_col='Unique Key')

    #fix the zip
    df['Incident Zip'] = df['Incident Zip'].apply(fix_zip)

    #drop all rows that have any nans in them (note the easier syntax!)

    df = df.dropna(how='any')

    #get rid of unspecified boroughs
    df = df[df['Borough'] != 'Unspecified']

    #Convert times to datetime and create a processing time column

    import datetime
    df['Created Date'] = df['Created Date'].apply(lambda x:datetime.datetime.strptime(x,'%m/%d/%Y %I:%M:%S %p'))
    df['Closed Date'] = df['Closed Date'].apply(lambda x:datetime.datetime.strptime(x,'%m/%d/%Y %I:%M:%S %p'))
    df['processing_time'] =  df['Closed Date'] - df['Created Date']
```

Plotting Data on Google Maps

```
In [3]: data
```

Out[3]:

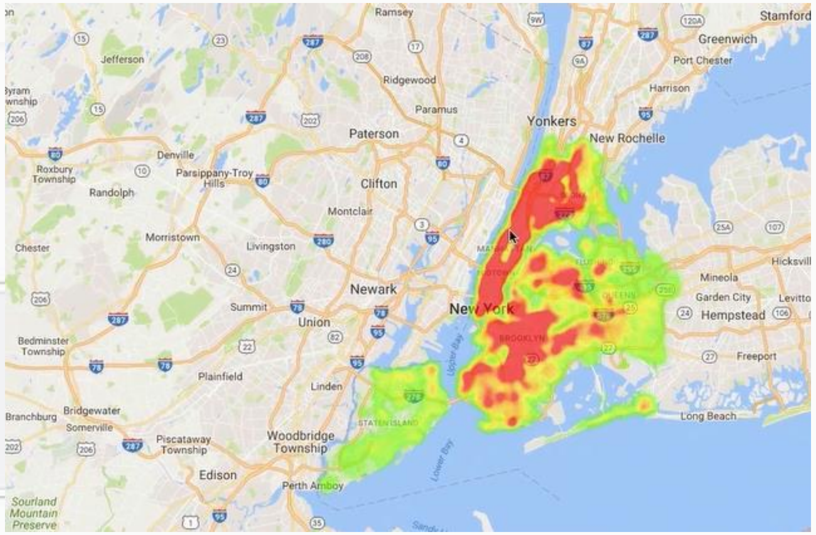
	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude	processing_time
Unique Key								
34215673	2016-09-01 00:33:42	2016-09-16 01:06:56	DCA	10001	MANHATTAN	40.744790	-73.988834	15 days 00:33:14
34219052	2016-09-01 20:16:24	2016-09-10 18:08:25	HPD	11691	QUEENS	40.600554	-73.750704	8 days 21:52:01
34219145	2016-09-01 12:17:00	2016-09-07 12:00:00	DSNY	11211	BROOKLYN	40.704925	-73.962007	5 days 23:43:00
34219385	2016-09-01 12:10:22	2016-09-10 14:23:44	HPD	10027	MANHATTAN	40.812322	-73.955338	9 days 02:13:22
34219399	2016-09-01 12:32:32	2016-09-11 02:03:37	HPD	10452	BRONX	40.839529	-73.922534	9 days 13:31:05
34219470	2016-09-01 20:16:24	2016-09-10 18:08:24	HPD	11691	QUEENS	40.600554	-73.750704	8 days 21:52:00
34219513	2016-09-01 08:35:00	2016-09-07 12:00:00	DSNY	11428	QUEENS	40.721866	-73.745982	6 days 03:25:00
34219516	2016-09-01 13:19:42	2016-09-16 14:32:35	DOT	11101	QUEENS	40.746875	-73.952711	15 days 01:12:53
34219534	2016-09-01 11:00:00	2016-09-08 12:00:00	DSNY	10075	MANHATTAN	40.773336	-73.955054	7 days 01:00:00

Install library called – Gmplot library: https://github.com/vgm64/gmplot

```
In [4]: !pip install gmplot --upgrade
```

A dataframe contains latitudes and longitudes for each complaint. You can draw a heatmap that might help us see the relative concentration using latitudes and longitudes.

Drawing a Heatmap



Set up the map

GoogleMapPlotter constructor

- GoogleMapPlotter(center_lat, center_lng, zoom)
- from_geocode(location_string,zoom)

```
In [5]:
import gmplot
#gmap = gmplot.GoogleMapPlotter(40.7128, -74.0059, 8)


gmap = gmplot.GoogleMapPlotter.from_geocode("New York",10)
```

Then generate the heatmap passing the two data series (latitude and longitude) to the function

```
In [ ]:
#Then generate a heatmap using the latitudes and longitudes
gmap.heatmap(data['Latitude'], data['Longitude'])
```

Save the heatmap to an html file

The html file can be viewed, printed, or included in another html page

```
In [16]: gmap.draw('incidents.html')
```

Grouping Operations

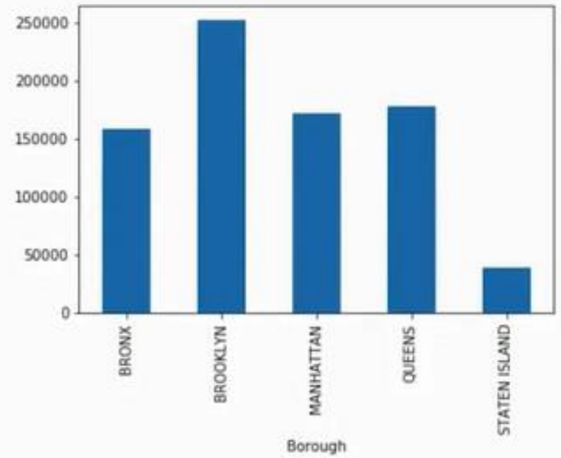
Incidents by Borough

In [20]:
```
%matplotlib inline
```

Group data by borough and plot a bar chart of the incident count

In [19]:
```
borough_group = data.groupby('Borough')
borough_group.size().plot(kind='bar')
#kind can be 'hist', 'scatter'
```

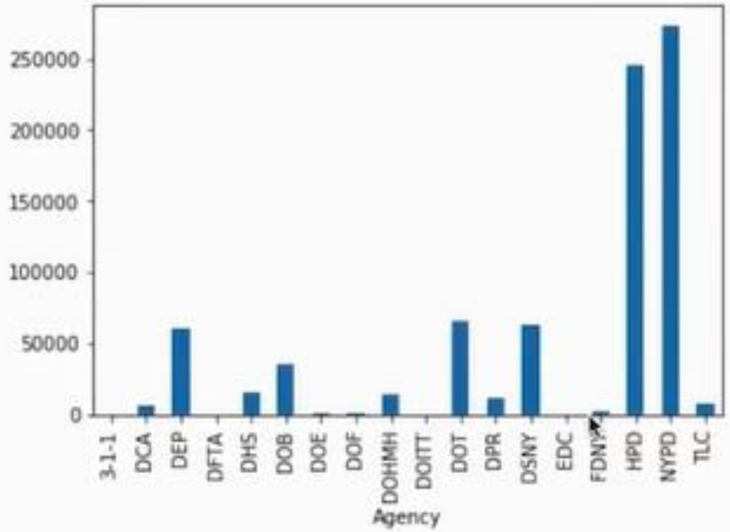
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x113b50ef0>



Incidents by Agency

In [22]:
```
agency_group = data.groupby('Agency')
agency_group.size().plot(kind='bar')
```

Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x113755400>

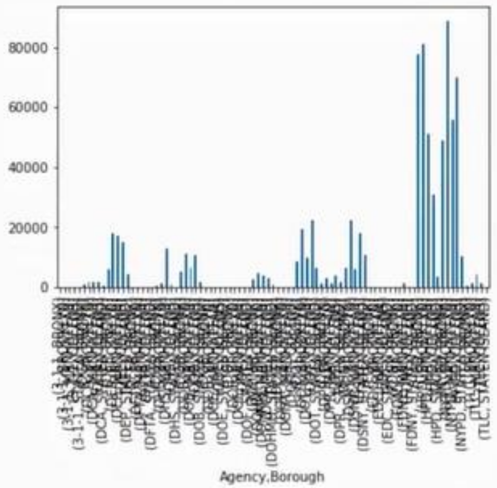


Grouping Operations

Let's combine the two in a single graph

```
In [23]:
agency_borough = data.groupby(['Agency','Borough'])
agency_borough.size().plot(kind='bar')
```

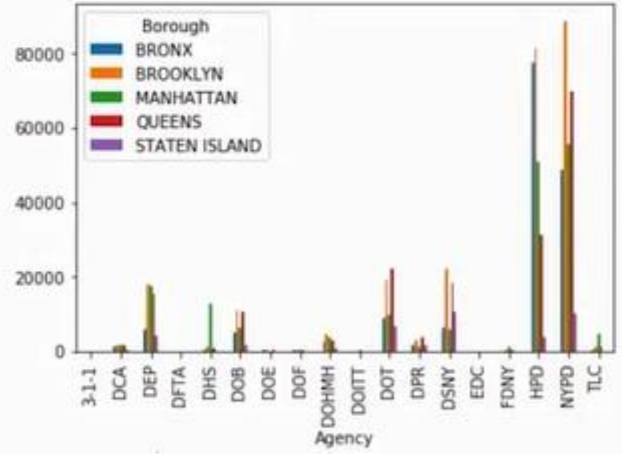
```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x113692e80>
```



This is unreadable and pointless; unstacking can make it clearer

We can unstack the groups so that we get borough by agency

```
In [24]: agency_borough.size().unstack().plot(kind='bar')
Out[24]: <matplotlib.axes._subplots.AxesSubplot at 0x1125571d0>
```



Increasing size of an image can make it readable and clearer.

Increase the size of the image and add a title

```
In [ ]:
agency_borough = data.groupby(['Agency','Borough'])
agency_borough.size().unstack().plot(kind='bar',title="Incidents in each Agency by Borough",figsize=(15,15))
```

Digression: The groupby Function

You can use functions to group data

In [27]:
```
import pandas as pd
writers = pd.DataFrame({'Author':['George Orwell','John Steinbeck',
                                  'Pearl Buck','Agatha Christie'],
                        'Country':['UK','USA','USA','UK'],
                        'Gender':['M','M','F','F'],
                        'Age':[46,66,80,85]})
```

In [28]:
```
writers
```

Out[28]:

	Age	Author	Country	Gender
0	46	George Orwell	UK	M
1	66	John Steinbeck	USA	M
2	80	Pearl Buck	USA	F
3	85	Agatha Christie	UK	F

Digression: The groupby Function

Group by country

```
In [29]: grouped = writers.groupby('Country')
         #grouped.first()
         #grouped.last()
         #grouped.sum()
         #grouped.mean()
         #grouped.apply(sum)

In [30]: grouped.groups

Out[30]: {'UK': Int64Index([0, 3], dtype='int64'),
          'USA': Int64Index([1, 2], dtype='int64')}
```

Group by multiple columns

```
In [40]: grouped = writers.groupby(['Country','Gender'])
         grouped.groups

Out[40]: {('UK', 'F'): Int64Index([3], dtype='int64'),
          ('UK', 'M'): Int64Index([0], dtype='int64'),
          ('USA', 'F'): Int64Index([2], dtype='int64'),
          ('USA', 'M'): Int64Index([1], dtype='int64')}
```

Perform basic analysis

```
In [33]: grouped = writers.groupby('Country')
         grouped.first()
         #grouped.last()
         #grouped.sum()
         #grouped.mean()
         #grouped.apply(sum)
```

Out[33]:

	Age	Author	Gender
Country			
UK	46	George Orwell	M

Group by age groups

```
In [41]: writers
```

Out[41]:

	Age	Author	Country	Gender
0	46	George Orwell	UK	M
1	66	John Steinbeck	USA	M
2	80	Pearl Buck	USA	F
3	85	Agatha Christie	UK	F

```
In [42]: def age_groups(df,index,col):
             print(index,col)
             if df[col].iloc[index] < 30:
                 return 'Young'
             if df[col].iloc[index] < 60:
                 return 'Middle'
             else:
                 return 'Old'

In [ ]: writers['Age'].iloc[0]

In [43]: grouped = writers.groupby(lambda x: age_groups(writers,x,'Age'))
         grouped.groups

         0 Age
         1 Age
         2 Age
         3 Age

Out[43]: {'Middle': Int64Index([0], dtype='int64'),
          'Old': Int64Index([1, 2, 3], dtype='int64')}
```

Digression: The groupby Function

Grouping by the values in a column

For example, grouping the data by values in a column that are greater than or less than zero

In [46]:
```
import numpy as np
import pandas as pd
people = pd.DataFrame(np.random.randn(5, 5), columns=['a', 'b', 'c', 'd', 'e'], index=['Joe', 'Steve', 'Wes', 'Jim',
people
```

Out[46]:

	a	b	c	d	e
Joe	1.147479	0.619510	-1.056473	0.315374	-1.106932
Steve	0.790722	-0.641755	1.709861	0.078417	-0.050602
Wes	-0.449524	-1.060829	-1.175451	-0.435145	0.509904
Jim	1.539456	0.325200	-0.679341	0.596718	1.764196
Travis	-1.493185	-0.550559	-1.025666	0.330545	0.760488

Write a function that takes three arguments - a dataframe, an index, and a column name and returns the grouping for that row

In [ ]:
```
def GroupColFunc(df, ind, col):
    if df[col].loc[ind] > 0:
        return 'Group1'
    else:
        return 'Group2'
```

In [ ]:
```
people.groupby(lambda x: GroupColFunc(people, x, 'a')).groups
```

Now we can compute stats on these groups

In [49]:
```
print(people.groupby(lambda x: GroupColFunc(people, x, 'a')).mean())
print(people.groupby(lambda x: GroupColFunc(people, x, 'a')).std())
```

```
               a         b         c        d         e
Group1  1.159219  0.100985 -0.008651  0.33017  0.202221
Group2 -0.971354 -0.805694 -1.100558 -0.05230  0.635196
               a         b         c         d         e
Group1  0.374505  0.659849  1.500173  0.259467  1.452165
Group2  0.737980  0.360816  0.105914  0.541425  0.177190
```

- Grouping is a versatile function to use in dataframes
- You can find sum, mean, standard deviations, or apply any functions to groups

Incidents by Time

We know the creation date of each incident so we can build a bar graph of number of incidents by month

Not particularly useful with a few months data but if we had all data from 2010, we could use this sort of analysis to eyeball trends and seasonality

We're going to need to do some data manipulation for this

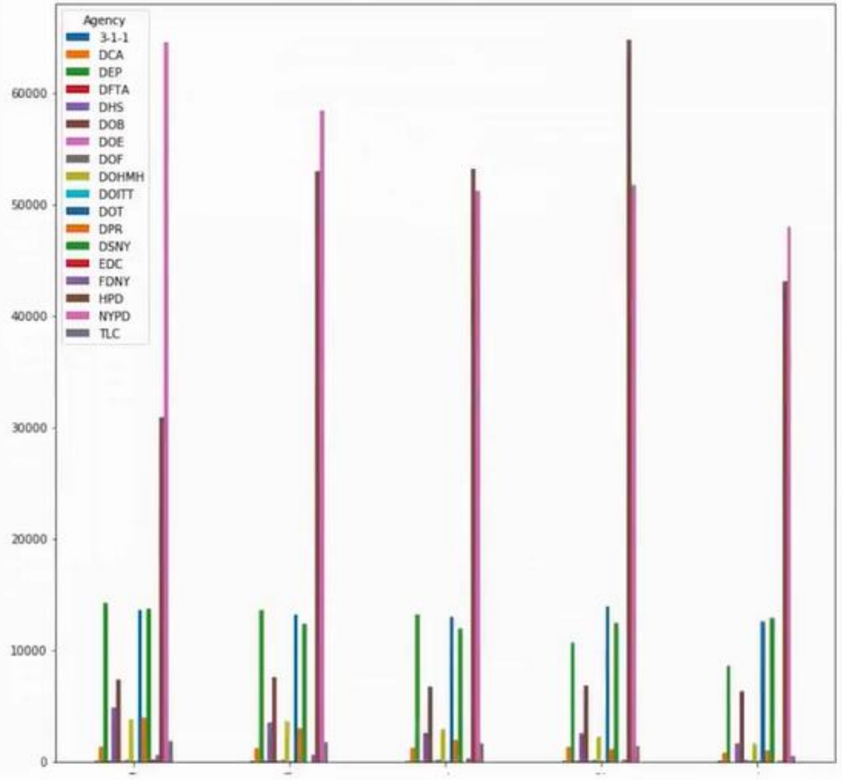
```
In [51]: import datetime
         data['yyyymm'] = data['Created Date'].apply(lambda x:datetime.datetime.strftim
```

```
In [52]: data['yyyymm']
```

```
Out[52]: Unique Key
         34215673    201609
         34219052    201609
         34219145    201609
         34219385    201609
         34219399    201609
         34219470    201609
         34219513    201609
         34219516    201609
         34219534    201609
         34219623    201609
         34219638    201609
         34219639    201609
         34219640    201609
         34219643    201609
         34219644    201609
         34219646    201609
         34219681    201609
         34219813    201609
         34219941    201609
         34220256    201609
         34220375    201609
         34220447    201609
         34220448    201609
         34220449    201609
         34220450    201609
         34220479    201609
         34220488    201609
         34220607    201609
         34220609    201609
         34220627    201609
                       ...
         34364916    201609
         34365064    201609
         34365415    201609
         34365624    201609
```

```
In [ ]: date_agency = data.groupby(['yyyymm','Agency'])
        date_agency.size().unstack().plot(kind='bar',figsize=(12,15))
```

```
Out[53]: <matplotlib.axes._subplots.AxesSubplot at 0x11bde53c8>
```



Examining by Agencies

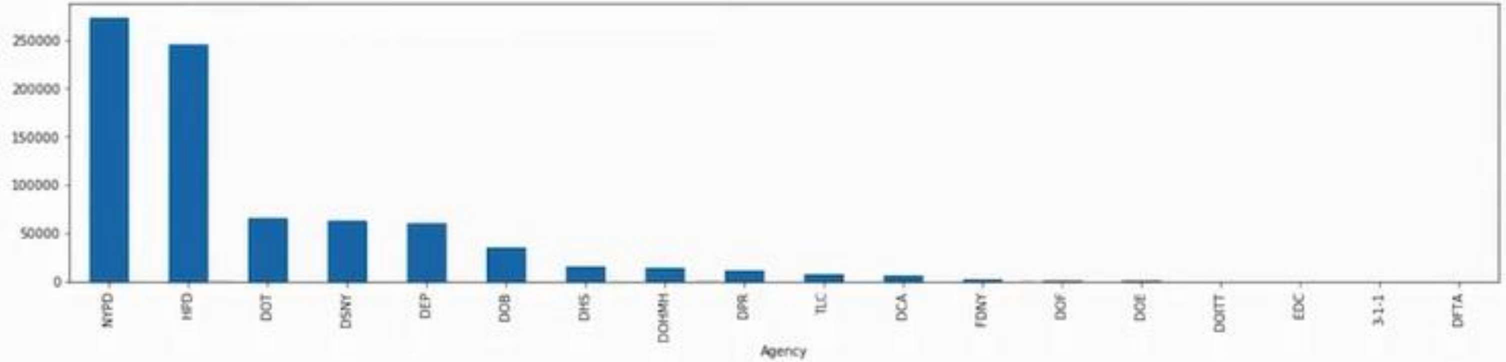
We'll look at the frequency by agency and report the top 5 values

```
In [54]: data.groupby('Agency').size().sort_values(ascending=False)
```

```
Out[54]: Agency
NYPD      273683
HPD       244815
DOT        66178
DSNY       63321
DEP        60346
DOB        34821
DHS        15083
DOHMH      14188
DPR        10830
TLC         7129
DCA         5760
FDNY        1676
DOF          579
DOE          454
DOITT        134
```

```
In [55]: data.groupby('Agency').size().sort_values(ascending=False).plot(kind='bar', figsize=(20,4))
```

```
Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x11bdea668>
```



Drill Down Data

Drilling down into agency complaints by borough

```
In [56]: agency_borough = data.groupby(['Agency', 'Borough']).size().unstack()

In [57]: agency_borough
Out[57]:
```

Borough	BRONX	BROOKLYN	MANHATTAN	QUEENS	STATEN ISLAND
Agency					
3-1-1	17.0	28.0	23.0	28.0	6.0
DCA	958.0	1532.0	1529.0	1547.0	194.0
DEP	5837.0	17917.0	17315.0	15216.0	4061.0
DFTA	21.0	33.0	24.0	21.0	2.0
DHS	397.0	1130.0	12767.0	734.0	55.0
DOB	5160.0	10993.0	6507.0	10567.0	1594.0
DOE	129.0	127.0	49.0	136.0	13.0
DOF	143.0	161.0	153.0	112.0	10.0
DOHMH	2406.0	4481.0	3759.0	2814.0	728.0
DOITT	7.0	18.0	91.0	18.0	NaN
DOT	8682.0	19176.0	9673.0	22096.0	6551.0
DPR	1416.0	2929.0	1103.0	3897.0	1485.0
DSNY	6406.0	22208.0	6079.0	18125.0	10503.0
EDC	1.0	62.0	41.0	15.0	4.0
FDNY	39.0	127.0	1344.0	158.0	8.0
HPD	77774.0	81382.0	51017.0	31080.0	3562.0
NYPD	48837.0	88973.0	55841.0	69931.0	10101.0
TLC	318.0	1238.0	4393.0	1146.0	34.0

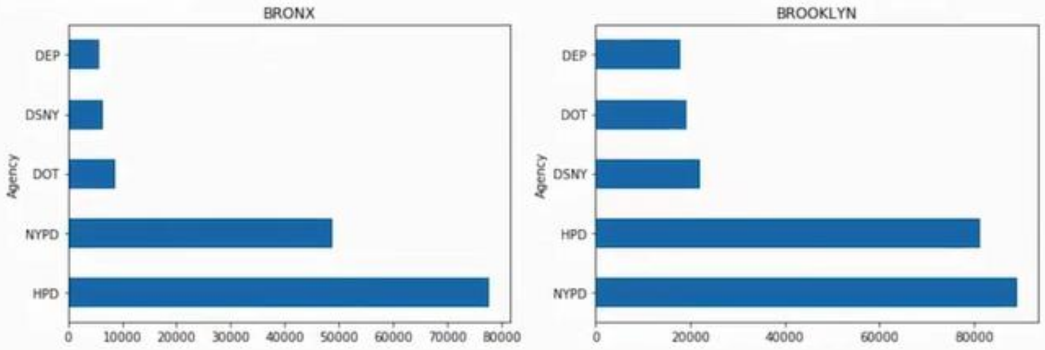
Examining by Agencies

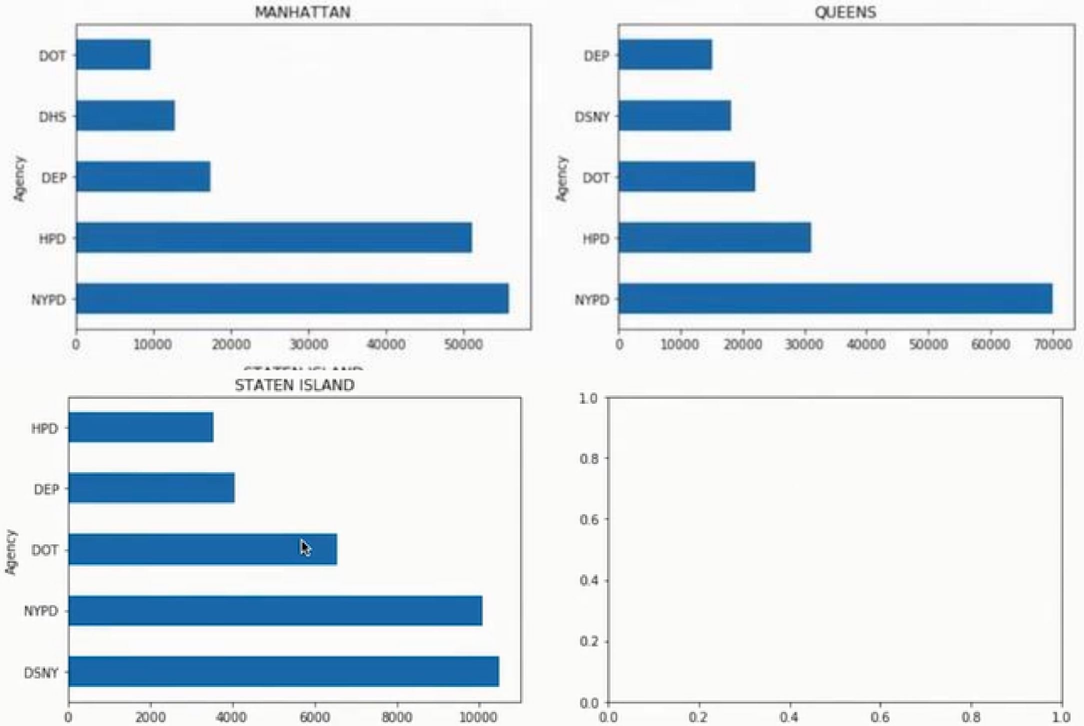
"Top Five Agency" subplot for each borough

```
In [58]: #We'll arrange the subplots in two rows and three columns.
         #Since we have only 5 boroughs, one plot will be blank
         COL_NUM = 2
         ROW_NUM = 3
         import matplotlib.pyplot as plt
         fig, axes = plt.subplots(ROW_NUM, COL_NUM, figsize=(12,12))

         for i, (label, col) in enumerate(agency_borough.iteritems()):
             ax = axes[int(i/COL_NUM), i%COL_NUM]
             col = col.sort_values(ascending=False)[:5]
             col.plot(kind='barh', ax=ax)
             ax.set_title(label)

         plt.tight_layout()
```





Processing Time

It is easier to convert the timedelta processing_time into floats for calculation purposes.

```
In [61]: grouped = data[['processing_time','Borough']].groupby('Borough'
```

```
In [62]: grouped.describe()
```

Out[62]:

		processing_time
Borough		
BRONX	count	158548
	mean	5 days 11:22:39.529133
	std	10 days 19:29:45.763262
	min	0 days 00:00:00
	25%	0 days 05:48:38.250000
	50%	1 days 21:27:00
	75%	5 days 19:48:12.750000
	max	145 days 00:23:57
BROOKLYN	count	252515
	mean	5 days 01:22:08.762913
	std	11 days 20:44:39.914032
	min	0 days 00:00:00
	25%	0 days 02:33:20.500000
	50%	0 days 20:19:00
	75%	4 days 05:20:01
	max	146 days 17:26:50
	count	171708
	mean	5 days 07:43:58.957480
	std	12 days 01:57:03.858305
	min	0 days 00:00:00

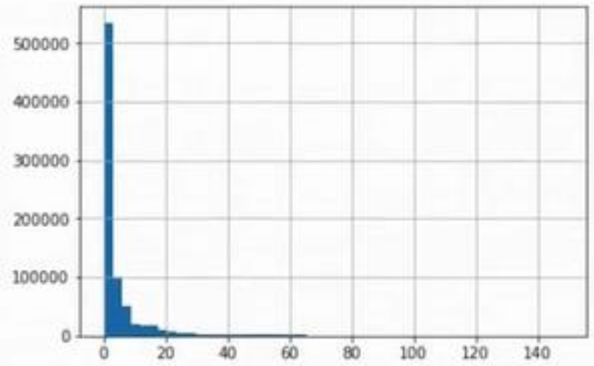
```
In [63]: import numpy as np
         #The time it takes to process. Cleaned up
         data['float_time'] =data['processing_time'].apply(lambda x:x/np.timedelta64(1, 'D'))
```

```
In [64]: data
```

Out[64]:

	Created Date	Closed Date	Agency	Incident Zip	Borough	Latitude	Longitude	processing_time	yyyymm	float_time
Unique Key										
34215673	2016-09-01 00:33:42	2016-09-16 01:06:56	DCA	10001	MANHATTAN	40.744790	-73.988834	15 days 00:33:14	201609	15.023079
34219052	2016-09-01 20:16:24	2016-09-10 18:08:25	HPD	11691	QUEENS	40.600554	-73.750704	8 days 21:52:01	201609	8.911123
34219145	2016-09-01 12:17:00	2016-09-07 12:00:00	DSNY	11211	BROOKLYN	40.704925	-73.962007	5 days 23:43:00	201609	5.988194
34219385	2016-09-01 12:10:22	2016-09-10 14:23:44	HPD	10027	MANHATTAN	40.812322	-73.955338	9 days 02:13:22	201609	9.092616
34219399	2016-09-01 12:32:32	2016-09-11	HPD	10452	BRONX	40.839529	-73.922534	9 days 13:31:05	201609	9.563252

```
In [66]: data['float_time'].hist(bins=50)
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x1206017b8>
```



Processing Time

Other useful visualization libraries

seaborn	https://seaborn.pydata.org
bokeh	http://bokeh.pydata.org/en/latest

GLOBAL IVY EDUCATION

www.emeritus.org