In [12]:
```
import datashader as ds
import datashader.transfer_functions as tf
import datashader.glyphs
from datashader import reductions
from datashader.core import bypixel
from datashader.utils import lnglat_to_meters as webm, export_image
from datashader.colors import colormap_select, Greys9, viridis, inferno
import copy
import math as math


from pyproj import Proj, transform
import numpy as np
import pandas as pd
import urllib
import json
import datetime
import colorlover as cl

import plotly.offline as py
import plotly.graph_objs as go
from plotly import tools

from shapely.geometry import Point, Polygon, shape
# In order to get shapley, you'll need to run [pip install shapely.geometry] from your terminal

from functools import partial

from IPython.display import GeoJSON

py.init_notebook_mode()
```

For module 2 we'll be looking at techniques for dealing with big data. In particular binning strategies and the datashader library (which possibly proves we'll never need to bin large data for visualization ever again.)

To demonstrate these concepts we'll be looking at the PLUTO dataset put out by New York City's department of city planning. PLUTO contains data about every tax lot in New York City.

PLUTO data can be downloaded from here (https://www1.nyc.gov/assets/planning/download/zip/data-maps/open-data/nyc_pluto_17v1_1.zip). Unzip them to the same directory as this notebook, and you should be able to read them in using this (or very similar) code. Also take note of the data dictionary, it'll come in handy for this assignment.

In [2]:
```
# Code to read in v17, column names have been updated (without upper case letters) for v18

# bk = pd.read_csv('PLUTO17v1.1/BK2017V11.csv')
# bx = pd.read_csv('PLUTO17v1.1/BX2017V11.csv')
# mn = pd.read_csv('PLUTO17v1.1/MN2017V11.csv')
# qn = pd.read_csv('PLUTO17v1.1/QN2017V11.csv')
# si = pd.read_csv('PLUTO17v1.1/SI2017V11.csv')

# ny = pd.concat([bk, bx, mn, qn, si], ignore_index=True)

ny = pd.read_csv('nyc_pluto_18v2_csv/pluto_18v2.csv', low_memory=False)

# Getting rid of some outliers
ny = ny[(ny['yearbuilt'] > 1850) & (ny['yearbuilt'] < 2020) & (ny['numfloors'] != 0)]
```

I'll also do some prep for the geographic component of this data, which we'll be relying on for datashader.

You're not required to know how I'm retrieving the lattitude and longitude here, but for those interested: this dataset uses a flat x-y projection (assuming for a small enough area that the world is flat for easier calculations), and this needs to be projected back to traditional lattitude and longitude.

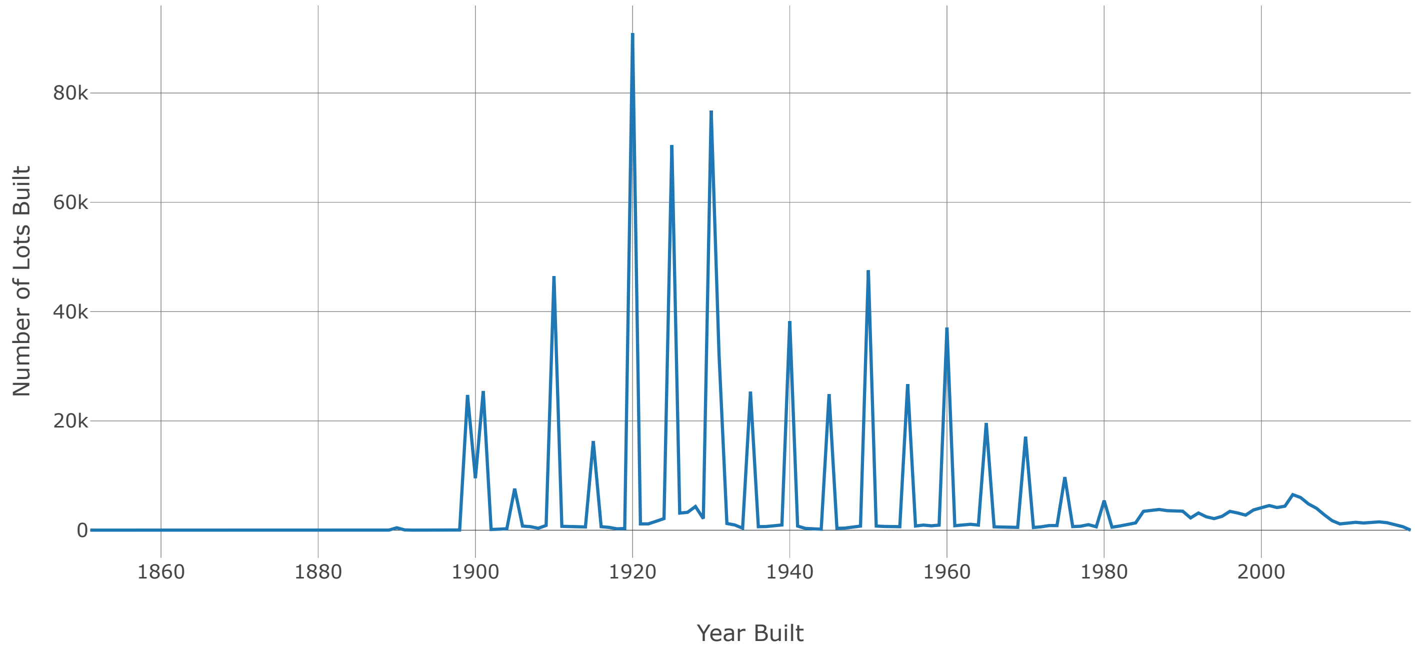
In [3]:
```
wgs84 = Proj("+proj=longlat +ellps=GRS80 +datum=NAD83 +no_defs")
nyli = Proj("+proj=lcc +lat_1=40.66666666666666 +lat_2=41.03333333333333 +lat_0=40.16666666666666 +lon_0=-74 +x_0=300000 +y_0=0 +ellps=GRS80 +dat
ny['xcoord'] = 0.3048*ny['xcoord']
ny['ycoord'] = 0.3048*ny['ycoord']
ny['lon'], ny['lat'] = transform(nyli, wgs84, ny['xcoord'].values, ny['ycoord'].values)

ny = ny[(ny['lon'] < -60) & (ny['lon'] > -100) & (ny['lat'] < 60) & (ny['lat'] > 20)]

#Defining some helper functions for DataShader
background = "black"
export = partial(export_image, background = background, export_path="export")
cm = partial(colormap_select, reverse=(background!="black"))
```

Part 1: Binning and Aggregation

Binning is a common strategy for visualizing large datasets. Binning is inherent to a few types of visualizations, such as histograms and [2D histograms (https://plot.ly/python/2D-Histogram/)](https://plot.ly/python/2D-Histogram/) (also check out their close relatives: [2D density plots (https://plot.ly/python/2d-density-plots/)](https://plot.ly/python/2d-density-plots/) and the more general form: [heatmaps (https://plot.ly/python/heatmaps/)](https://plot.ly/python/heatmaps/).

While these visualization types explicitly include binning, any type of visualization used with aggregated data can be looked at in the same way. For example, lets say we wanted to look at building construction over time. This would be best viewed as a line graph, but we can still think of our results as being binned by year:

In [4]:
```
trace = go.Scatter(
    # I'm choosing BBL here because I know it's a unique key.
    x = ny.groupby('yearbuilt').count()['bbl'].index,
    y = ny.groupby('yearbuilt').count()['bbl']
)

layout = go.Layout(
    xaxis = dict(title = 'Year Built'),
    yaxis = dict(title = 'Number of Lots Built')
)

fig = go.Figure(data = [trace], layout = layout)

py.iplot(fig)
```



Export to plot.ly »

Something looks off... You're going to have to deal with this imperfect data to answer this first question.

But first: some notes on pandas. Pandas dataframes are a different beast than R dataframes, here are some tips to help you get up to speed:

Hello all, here are some pandas tips to help you guys through this homework:

Indexing and Selecting (https://pandas.pydata.org/pandas-docs/stable/indexing.html): .loc and .iloc are the analogs for base R subsetting, or filter() in dplyr

Group By (https://pandas.pydata.org/pandas-docs/stable/groupby.html): This is the pandas analog to group_by() and the appended function the analog to summarize(). Try out a few examples of this, and display the results in Jupyter. Take note of what's happening to the indexes, you'll notice that they'll become hierarchical. I personally find this more of a burden than a help, and this sort of hierarchical indexing leads to a fundamentally different experience compared to R dataframes. Once you perform an aggregation, try running the resulting hierarchical datafrome through a reset_index() (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reset_index.html).

Reset_index (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.reset_index.html): I personally find the hierarchical indexes more of a burden than a help, and this sort of hierarchical indexing leads to a fundamentally different experience compared to R dataframes. reset_index() is a way of restoring a dataframe to a flatter index style. Grouping is where you'll notice it the most, but it's also useful when you filter data, and in a few other split-apply-combine workflows. With pandas indexes are more meaningful, so use this if you start getting unexpected results.

Indexes are more important in Pandas than in R. If you delve deeper into the using python for data science, you'll begin to see the benefits in many places (despite the personal gripes I highlighted above.) One place these indexes come in handy is with time series data. The pandas docs have a huge section (http://pandas.pydata.org/pandas-docs/stable/timeseries.html) on datetime indexing. In particular, check out resample (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.resample.html), which provides time series specific aggregation.

Merging, joining, and concatenation (https://pandas.pydata.org/pandas-docs/stable/merging.html): There's some overlap between these different types of merges, so use this as your guide. Concat is a single function that replaces cbind and rbind in R, and the results are driven by the indexes. Read through these examples to get a feel on how these are performed, but you will have to manage your indexes when you're using these functions. Merges are fairly similar to merges in R, similarly mapping to SQL joins.

Apply: This is explained in the "group by" section linked above. These are your analogs to the plyr library in R. Take note of the lambda syntax used here, these are anonymous functions in python. Rather than predefining a custom function, you can just define it inline using lambda.

Browse through the other sections for some other specifics, in particular reshaping and categorical data (pandas' answer to factors.) Pandas can take a while to get used to, but it is a pretty strong framework that makes more advanced functions easier once you get used to it. Rolling functions for example follow logically from the apply workflow (and led to the best google results ever when I first tried to find this out and googled "pandas rolling")

Google Wes Mckinney's book "Python for Data Analysis," which is a cookbook style intro to pandas. It's an O'Reilly book that should be pretty available out there.

Question

After a few building collapses, the City of New York is going to begin investigating older buildings for safety. The city is particularly worried about buildings that were unusually tall when they were built, since best-practices for safety hadn't yet been determined. Create a graph that shows how many buildings of a certain number of floors were built in each year (note: you may want to use a log scale for the number of buildings). Find a strategy to bin buildings (It should be clear 20-29-story buildings, 30-39-story buildings, and 40-49-story buildings were first built in large numbers, but does it make sense to continue in this way as you get taller?)

In [5]:
```python
# Start your answer here, inserting more cells as you go along

# Create a subset for question 1 that only has 'yearbuilt' and 'numfloors' columns.
ny_subset = ny[['yearbuilt','numfloors']]

#suppress warning when data type of 'yearbuilt' is changed: https://github.com/pandas-dev/pandas/pull/5390
pd.set_option('chained',None)

#change data type of 'yearbuilt' from float to int
ny_subset['yearbuilt'] = ny_subset.yearbuilt.astype(int)

#show head of ny_subset
print(ny_subset.head())
```

```
   yearbuilt  numfloors
0       1930        2.0
1       1935        2.5
2       1977        2.0
3       1920        2.0
4       1925        2.5
```

In [6]:
```python
#DataFrame for descriptive statistics on number of floors for each year
year_numfloor_stats = ny_subset['numfloors'].groupby(ny_subset['yearbuilt']).describe()
```

Below you will see a preview of some descriptive statistics of the number of floors for each year.

In [7]:
```python
year_numfloor_stats.head(n=10)
```

Out[7]:

yearbuilt	count	mean	std	min	25%	50%	75%	max
1851	8.0	3.750000	0.707107	3.0	3.0000	4.00	4.000	5.0
1852	13.0	3.307692	0.947331	2.0	3.0000	3.00	4.000	5.0
1853	12.0	4.125000	2.317179	2.0	3.0000	3.50	4.125	11.0
1854	6.0	3.791667	0.400520	3.0	3.8125	4.00	4.000	4.0
1855	14.0	3.071429	1.124160	1.0	3.0000	3.00	3.875	5.0
1856	11.0	4.045455	0.789131	2.0	4.0000	4.00	4.250	5.0
1857	8.0	4.500000	1.388730	3.0	3.5000	4.00	5.250	7.0
1858	8.0	3.312500	0.961305	2.0	2.8750	3.00	4.000	5.0
1859	10.0	3.350000	1.131616	2.0	2.5000	3.25	4.000	5.0
1860	37.0	3.864865	1.631498	1.0	3.0000	3.50	4.000	10.0

In [8]:
```
year_numfloor_stats.tail(n=10)
```

Out[8]:

yearbuilt	count	mean	std	min	25%	50%	75%	max
2010	1146.0	3.497452	4.206691	1.0	2.00	2.00	3.50	58.0
2011	1327.0	3.106307	3.294795	1.0	2.00	2.00	3.00	50.0
2012	1441.0	3.707821	5.079404	1.0	2.00	2.00	4.00	90.0
2013	1311.0	4.271571	6.482046	1.0	2.00	3.00	4.00	88.0
2014	1499.0	3.644376	4.218667	1.0	2.00	2.50	4.00	57.0
2015	1523.0	4.649028	7.807752	1.0	2.00	3.00	4.00	88.0
2016	1363.0	4.548166	6.062021	1.0	2.00	3.00	4.00	71.0
2017	1094.0	4.079671	4.640522	1.0	2.00	3.00	4.31	63.0
2018	639.0	3.868545	3.965952	1.0	2.00	2.25	4.00	43.0
2019	4.0	5.500000	3.109126	1.0	4.75	6.50	7.25	8.0

Below you will get an idea of the min-max range of the min, median, 75% pecentile, and max number of floors across all years.

- Across all years, the range for the min number of floors is from 0.5 to 3.0 floors.
- Across all years, the range for the median number of floors is from 2.0 to 7.0 floors.
- Across all years, the range for the 75th percential number of floors is from 2.0 to 8.0 floors.
- Across all years, the range for the max number of floors is from 4.0 to 205.0 floors.

This tells us that most building built from 1851 to 2019 are not very tall buildings.

In [9]:
```
# print(year_numfloor_stats.columns)
print('Min of min number of floors across all years: ' + str(year_numfloor_stats['min'].min()))
print('Max of min number of floors across all years: ' + str(year_numfloor_stats['min'].max()))
print('Min of median number of floors across all years: ' + str(year_numfloor_stats['50%'].min()))
print('Max of median number of floors across all years: ' + str(year_numfloor_stats['50%'].max()))
print('Min of 75th percentile number of floors across all years: ' + str(year_numfloor_stats['75%'].min()))
print('Max of 75th percentile number of floors across all years: ' + str(year_numfloor_stats['75%'].max()))
print('Min of max number of floors across all years: ' + str(year_numfloor_stats['max'].min()))
print('Max of max number of floors across all years: ' + str(year_numfloor_stats['max'].max()))
```

```
Min of min number of floors across all years: 0.5
Max of min number of floors across all years: 3.0
Min of median number of floors across all years: 2.0
Max of median number of floors across all years: 7.0
Min of 75th percentile number of floors across all years: 2.0
Max of 75th percentile number of floors across all years: 8.0
Min of max number of floors across all years: 4.0
Max of max number of floors across all years: 205.0
```

I'm going to take a look at the distribution of the number of floors across all years.

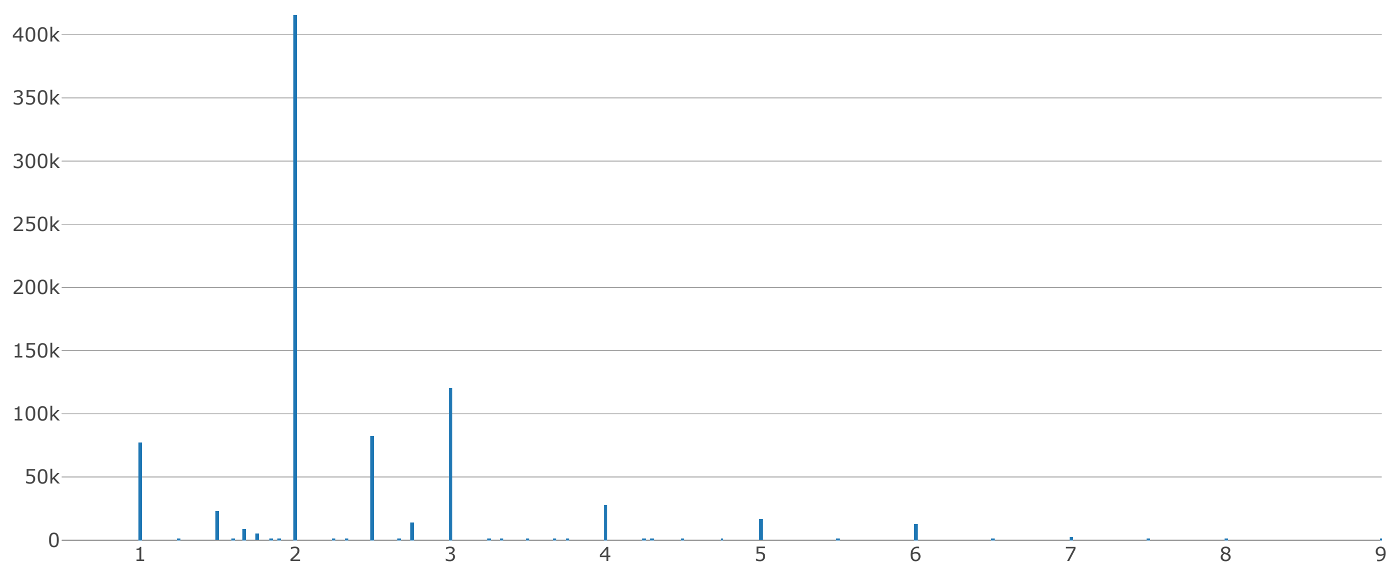
We know that the number of floors inl the entire data set is from 0.5 to 205 floors. From the analysis above, we know that the 75th percentile of the buildings fall within 2 to 8 floors.

This is the distribution of the buildings with less than 10 floors.

To make the binning easier, I decided to round up the floors that are not whole numbers. For example, if a building has 2.75 floors, this will be rounded up to 3 floors.

In [13]:
```
#create a new column 'numfloors_roundedup'
ny_subset['numfloors_roundedup'] = ny_subset['numfloors']

#apply the math.ceill function to the column 'numfloors_rounded up'
ny_subset['numfloors_roundedup'] = ny_subset['numfloors_roundedup'].apply(math.ceil)
```

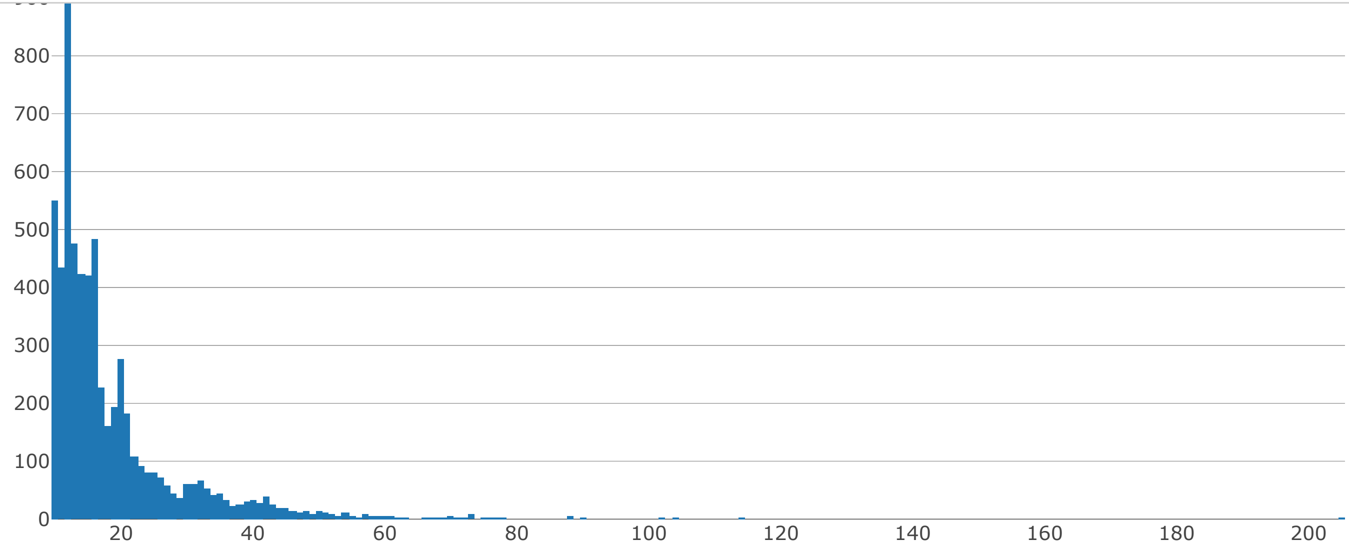
In [14]:
```
x = ny_subset[ny_subset['numfloors_roundedup'] < 10]
x = x['numfloors']
data = [go.Histogram(x=x)]
py.iplot(data)
```



Export to plot.ly »

Below is distribution of buildings that fall above the 75th percentile. These are buildings with 10 floors or higher.

In [15]:
```
x = ny_subset[ny_subset['numfloors'] >= 10]
x = x['numfloors']
data = [go.Histogram(x=x)]
py.iplot(data, filename='basic histogram')
```



Here are the bins by number of floors that I am going to use:

- 1 to 5
- 6 to 10
- 11 to 20
- 21 to 50
- 51 to 80
- 81+

In [16]:
```
def assign_category(num_of_floors):
    if num_of_floors <= 5:
        return '1 - 5'
    if num_of_floors > 5 and num_of_floors <= 10:
        return '6 - 10'
    if num_of_floors > 10 and num_of_floors <= 20:
        return '11 - 20'
    if num_of_floors > 20 and num_of_floors <= 50:
        return '21 - 50'
    if num_of_floors > 50 and num_of_floors <= 80:
        return '51 - 80'
    return '81+'
```

In [17]:
```
#Assign category labels for the bins
ny_subset['floor_category'] = ny_subset['numfloors_roundedup'].apply(assign_category)

#Drop columns not needed
floor_category_year = ny_subset.drop(['numfloors', 'numfloors_roundedup'], axis=1)

#preview data
floor_category_year.head(n=10)
```

Out[17]:

	yearbuilt	floor_category
0	1930	1 - 5
1	1935	1 - 5
2	1977	1 - 5
3	1920	1 - 5
4	1925	1 - 5
5	1930	1 - 5
7	2006	6 - 10
8	2004	1 - 5
9	1981	1 - 5
11	1900	1 - 5

Histogram for each floor category

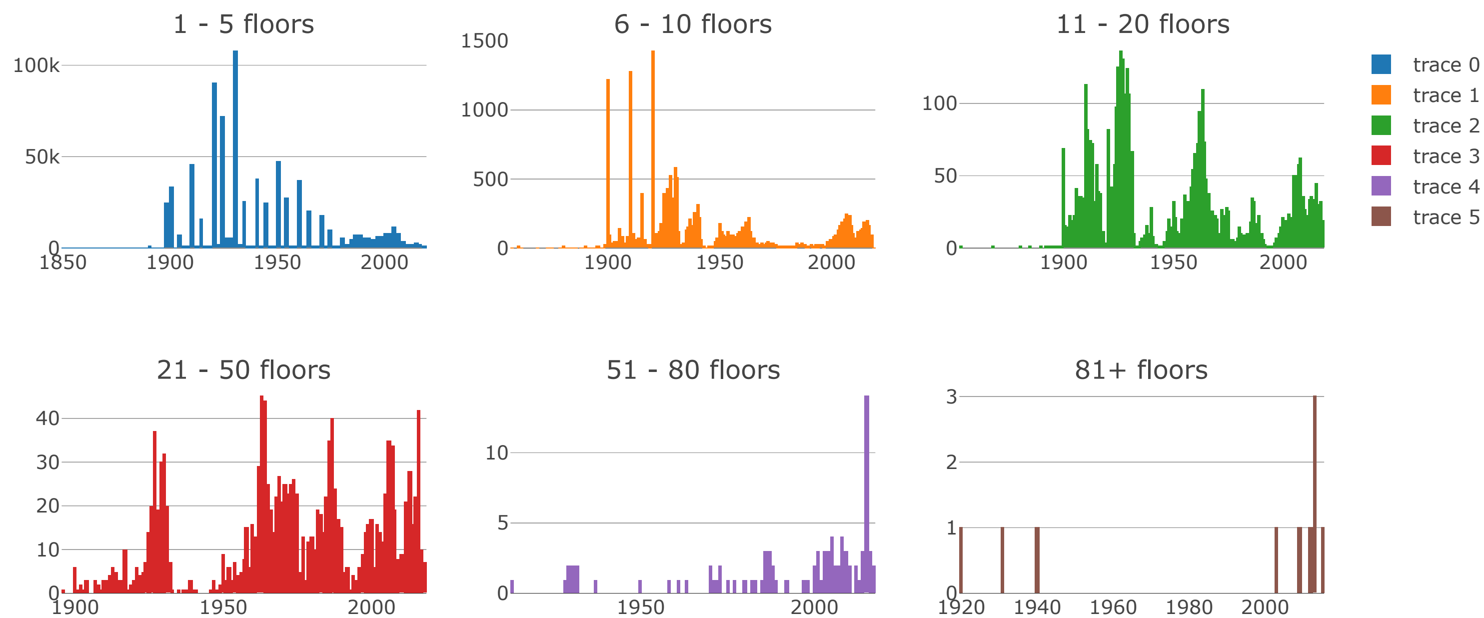
I tried to use the code 'fig['layout']['yaxis1'].update(type='log')' thinking that this would make the histogram log scaled. The plots did not look right to me. I commented out the code.

In [18]:

```python
x0 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='1 - 5']
x1 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='6 - 10']
x2 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='11 - 20']
x3 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='21 - 50']
x4 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='51 - 80']
x5 = floor_category_year.yearbuilt[floor_category_year['floor_category']=='81+']

trace0 = go.Histogram(
    x=x0,
    nbinsx = 168
  )
trace1 = go.Histogram(
    x=x1,
    nbinsx = 168,
  )
trace2 = go.Histogram(
    x=x2,
    nbinsx = 168,
  )
trace3 = go.Histogram(
    x=x3,
    nbinsx = 168,
  )
trace4 = go.Histogram(
    x=x4,
    nbinsx = 168,
  )
trace5 = go.Histogram(
    x=x5,
    nbinsx = 168,
  )


fig = tools.make_subplots(rows=2, cols=3, subplot_titles=('1 - 5 floors', '6 - 10 floors', '11 - 20 floors',
                                                          '21 - 50 floors','51 - 80 floors', '81+ floors'))
fig.append_trace(trace0, 1, 1)
fig.append_trace(trace1, 1, 2)
fig.append_trace(trace2, 1, 3)
fig.append_trace(trace3, 2, 1)
fig.append_trace(trace4, 2, 2)
fig.append_trace(trace5, 2, 3)

#plot without y axis log scaled
py.iplot(fig)

#NOTE: I couldn't figure out how to plot the histograms using log scale. The code below does not look right.
#So I commented them out.
#fig['layout']['yaxis1'].update(type='log')
#fig['layout']['yaxis2'].update(type='log')
#fig['layout']['yaxis3'].update(type='log')
#fig['layout']['yaxis4'].update(type='log')
#fig['layout']['yaxis5'].update(type='log')
#fig['layout']['yaxis6'].update(type='log')
```

```
#plot with y axis log scaled
#py.iplot(fig)
```

```
This is the format of your plot grid:
[ (1,1) x1,y1 ]  [ (1,2) x2,y2 ]  [ (1,3) x3,y3 ]
[ (2,1) x4,y4 ]  [ (2,2) x5,y5 ]  [ (2,3) x6,y6 ]
```



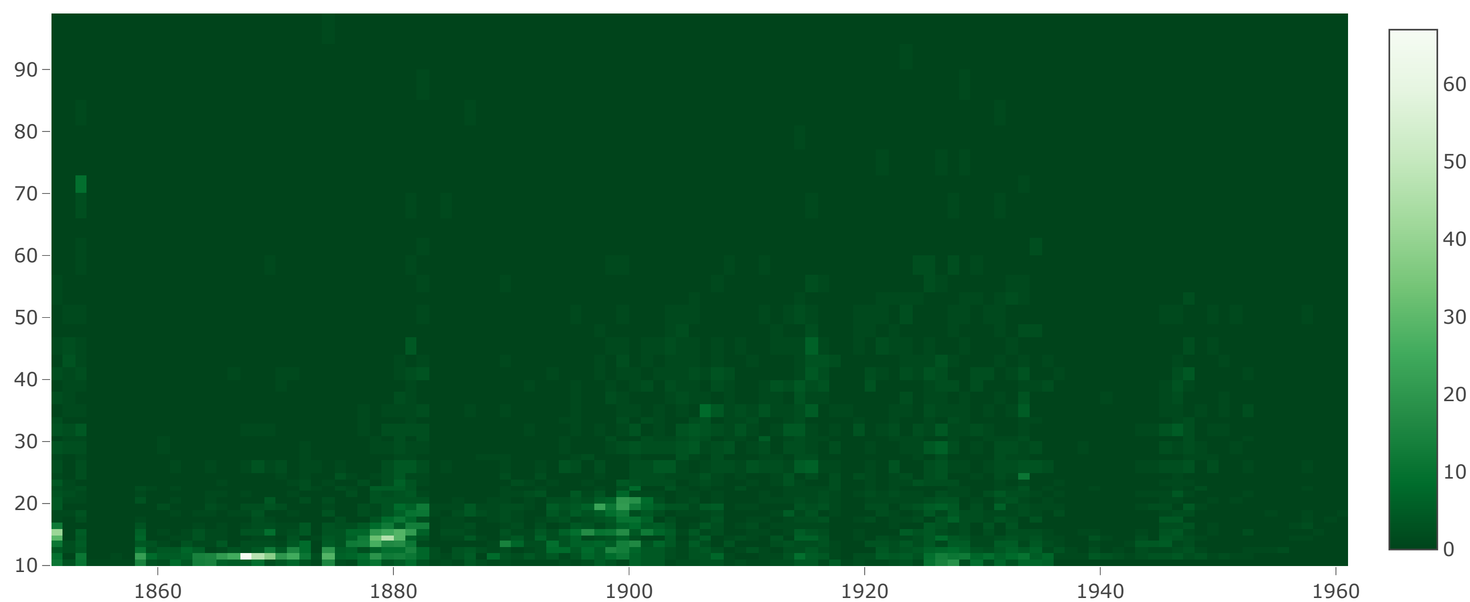
Export to plot.ly »

Part 2: Datashader

Datashader is a library from Anaconda that does away with the need for binning data. It takes in all of your datapoints, and based on the canvas and range returns a pixel-by-pixel calculations to come up with the best representation of the data. In short, this completely eliminates the need for binning your data.

As an example, lets continue with our question above and look at a 2D histogram of YearBuilt vs NumFloors:

In [19]:
```
yearbins = 200
floorbins = 200

yearBuiltCut = pd.cut(ny['yearbuilt'], np.linspace(ny['yearbuilt'].min(), ny['yearbuilt'].max(), yearbins))
numFloorsCut = pd.cut(ny['numfloors'], np.logspace(1, np.log(ny['numfloors'].max()), floorbins))

xlabels = np.floor(np.linspace(ny['yearbuilt'].min(), ny['yearbuilt'].max(), yearbins))
ylabels = np.floor(np.logspace(1, np.log(ny['numfloors'].max()), floorbins))

data = [
    go.Heatmap(z = ny.groupby([numFloorsCut, yearBuiltCut])['bbl'].count().unstack().fillna(0).values,
              colorscale = 'Greens', x = xlabels, y = ylabels)
]

py.iplot(data)
```



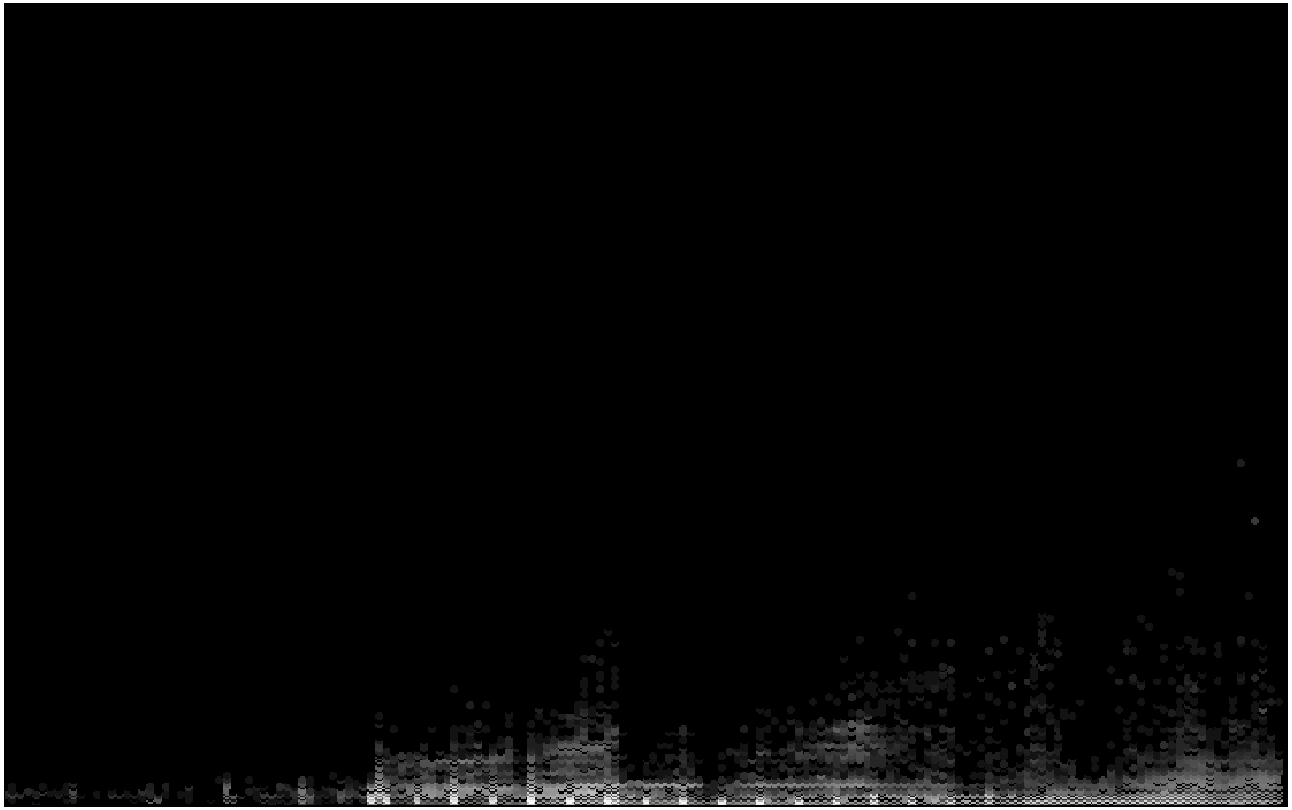
Export to plot.ly »

This shows us the distribution, but it's subject to some biases discussed in the Anaconda notebook [Plotting Perils (https://anaconda.org/jbednar/plotting_pitfalls/notebook)](https://anaconda.org/jbednar/plotting_pitfalls/notebook).

Here is what the same plot would look like in datashader:

In [20]:
```
cvs = ds.Canvas(800, 500, x_range = (ny['yearbuilt'].min(), ny['yearbuilt'].max()),
                                y_range = (ny['numfloors'].min(), ny['numfloors'].max()))
agg = cvs.points(ny, 'yearbuilt', 'numfloors')
view = tf.shade(agg, cmap = cm(Greys9), how='log')
export(tf.spread(view, px=2), 'yearvsnumfloors')
```

Out[20]:



That's technically just a scatterplot, but the points are smartly placed and colored to mimic what one gets in a heatmap. Based on the pixel size, it will either display individual points, or will color the points of denser regions.

Datashader really shines when looking at geographic information. Here are the latitudes and longitudes of our dataset plotted out, giving us a map of the city colored by density of structures:

In [21]:
```
NewYorkCity   = (( -74.29,  -73.69), (40.49, 40.92))
cvs = ds.Canvas(700, 700, *NewYorkCity)
agg = cvs.points(ny, 'lon', 'lat')
view = tf.shade(agg, cmap = cm(inferno), how='log')
export(tf.spread(view, px=2), 'firery')
```

Out[21]:

Interestingly, since we're looking at structures, the large buildings of Manhattan show up as less dense on the map. The densest areas measured by number of lots would be single or multi family townhomes.

Unfortunately, Datashader doesn't have the best documentation. Browse through the examples from their [github repo (https://github.com/bokeh/datashader/tree/master/examples)](https://github.com/bokeh/datashader/tree/master/examples). I would focus on the [visualization pipeline (https://anaconda.org/jbednar/pipeline/notebook)](https://anaconda.org/jbednar/pipeline/notebook) and the [US Census (https://anaconda.org/jbednar/census/notebook)](https://anaconda.org/jbednar/census/notebook) Example for the question below. Feel free to use my samples as templates as well when you work on this problem.

Question

You work for a real estate developer and are researching underbuilt areas of the city. After looking in the [Pluto data dictionary (https://www1.nyc.gov/assets/planning/download/pdf/data-maps/open-data/pluto_datadictionary.pdf?v=17v1_1)](https://www1.nyc.gov/assets/planning/download/pdf/data-maps/open-data/pluto_datadictionary.pdf?v=17v1_1), you've discovered that all tax assessments consist of two parts: The assessment of the land and assessment of the structure. You reason that there should be a correlation between these two values: more valuable land will have more valuable structures on them (more valuable in this case refers not just to a mansion vs a bungalow, but an apartment tower vs a single family home). Deviations from the norm could represent underbuilt or overbuilt areas of the city. You also recently read a really cool blog post about [bivariate choropleth maps (http://www.joshuastevens.net/cartography/make-a-bivariate-choropleth-map/)](http://www.joshuastevens.net/cartography/make-a-bivariate-choropleth-map/), and think the technique could be used for this problem.

Datashader is really cool, but it's not that great at labeling your visualization. Don't worry about providing a legend, but provide a quick explanation as to which areas of the city are overbuilt, which areas are underbuilt, and which areas are built in a way that's properly correlated with their land value.

Answer

I am not sure how to start to approach this question. I looked at the PLUTO dictionary and read the bivariate choropleth maps article. I noticed 2 variables that could possibly be used to answer this question: (1) assessed land value (AssessLand) and (2) building class (BldgClass). I could break down the AssessLand into low, medium, high (3 groups) and there are so many building classes (Appendix C) - categories that go from A through Z. Perhaps I can focus on on a limited number of building classes just to get an insight.

I decided to just focus on condominiums (R), and take a look at the assessland value of these properties across the 5 boroughs.

Below is a discription of the assessland value of the lots in New York City.

In [22]:
```
ny['assessland'].describe()
```

Out[22]:
```
count    8.124310e+05
mean     9.646607e+04
std      3.984989e+06
min      0.000000e+00
25%      6.985000e+03
50%      9.658000e+03
75%      1.429300e+04
max      3.211276e+09
Name: assessland, dtype: float64
```

In [23]:
```
ny_subset2 = ny[['borough', 'assessland', 'bldgclass']]
ny_subset2.head(n=5)
```

Out[23]:

	borough	assessland	bldgclass
0	QN	6787.0	A5
1	QN	9758.0	B3
2	SI	67764.0	R3
3	BK	12191.0	A1
4	BK	13079.0	A5

Rank the lots/properties by their assessland value

In [24]:
```
ny_subset2['assessland_rank'] = ny_subset2.assessland.rank(pct=True)
```

In [25]:
```
condos = ny_subset2[ny_subset2['bldgclass'].str.contains('R')]
condos.head()
```

Out[25]:

	borough	assessland	bldgclass	assessland_rank
2	SI	67764.0	R3	0.906718
7	BK	56258.0	R4	0.897532
8	BK	3761.0	R1	0.059651
9	BK	43105.0	R1	0.883210
11	MN	68492.0	R1	0.907229

Visualize assessland value of condiminiums across 5 boroughs

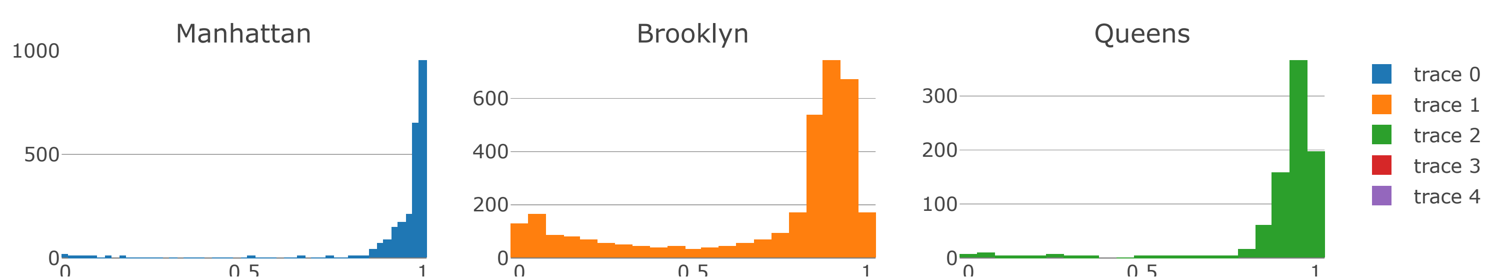
Below is distribution of lots/properties by their assessland rank.

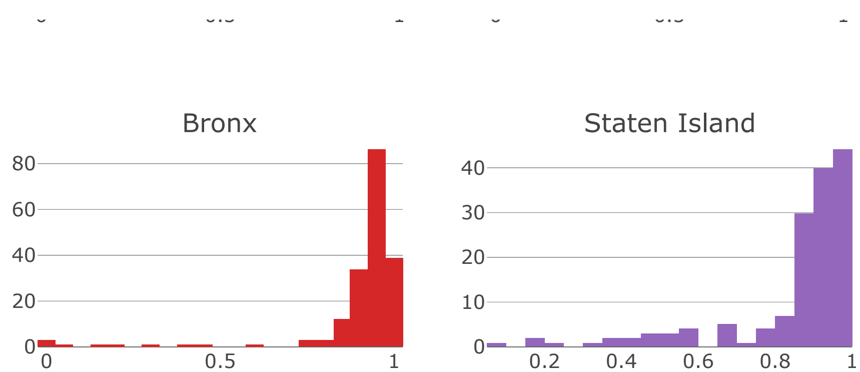
In [26]:
```
condos.borough.unique()
```

Out[26]:
```
array(['SI', 'BK', 'MN', 'QN', 'BX'], dtype=object)
```

In [27]:
```
x0 = condos.assessland_rank[condos['borough']=='MN']
x1 = condos.assessland_rank[condos['borough']=='BK']
x2 = condos.assessland_rank[condos['borough']=='QN']
x3 = condos.assessland_rank[condos['borough']=='BX']
x4 = condos.assessland_rank[condos['borough']=='SI']

trace0 = go.Histogram(
    x=x0
  )
trace1 = go.Histogram(
    x=x1
  )
trace2 = go.Histogram(
    x=x2
  )
trace3 = go.Histogram(
    x=x3
  )
trace4 = go.Histogram(
    x=x4
  )


fig = tools.make_subplots(rows=2, cols=3, subplot_titles=('Manhattan', 'Brooklyn', 'Queens',
                                                          'Bronx','Staten Island'))
fig.append_trace(trace0, 1, 1)
fig.append_trace(trace1, 1, 2)
fig.append_trace(trace2, 1, 3)
fig.append_trace(trace3, 2, 1)
fig.append_trace(trace4, 2, 2)

#plot without y axis log scaled
py.iplot(fig)
```

```
This is the format of your plot grid:
[ (1,1) x1,y1 ]  [ (1,2) x2,y2 ]  [ (1,3) x3,y3 ]
[ (2,1) x4,y4 ]  [ (2,2) x5,y5 ]  [ (2,3) x6,y6 ]
```





Export to plot.ly »

Answer

Based on this very simple analysis, there is a similar pattern in the distribution of condominiums assessland value. Most are concentrated towards the the right. Most condominiums rank towards the top in terms of assessment land value. However, for all boroughs there are some condominiums that are on the lower end of the assessment land value. These are most likely neighborhoods that are developing.

```
In [ ]:
```