In [145]:

```
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_im
age
from datashader.colors import colormap_select, Greys9, viridis,
inferno
import copy


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 shape
ly.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/site/planning/data-maps/open-data/dwn-pluto-mappluto.page)](https://www1.nyc.gov/site/planning/data-maps/open-data/dwn-pluto-mappluto.page). 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 [146]:

```
# Code to read in v17, column names have been updated (without u
pper 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_20v1_csv/pluto_20v1.csv')


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

```
/Users/bobo/opt/anaconda3/lib/python3.7/site-package
s/IPython/core/interactiveshell.py:3058: DtypeWarnin
g:

Columns (17,18,20,22) have mixed types. Specify dtyp
e option on import or set low_memory=False.
```

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 [147]:

```
# 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 +datum=NAD83 +to_meter=0.3048006096012192 +no_defs")
# 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 [148]:

```
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.FigureWidget(data = [trace], layout = layout)

fig
```

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)](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)](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)](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)](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)](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)](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)](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 [136]:

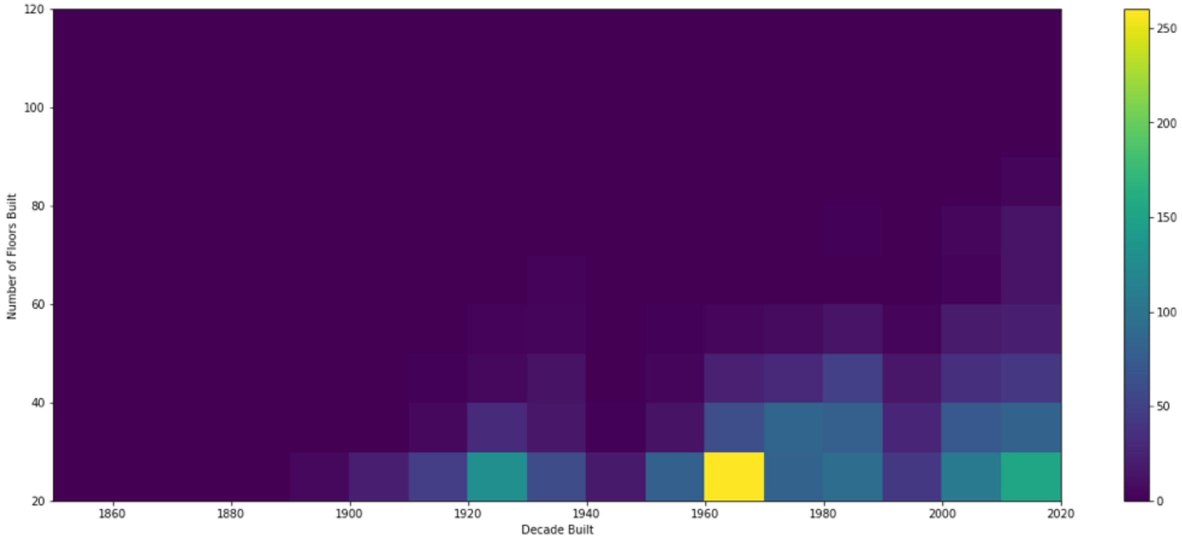
```
ny.groupby('yearbuilt')
```

Out[136]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object
at 0x8c7a328d0>
```

In [149]:

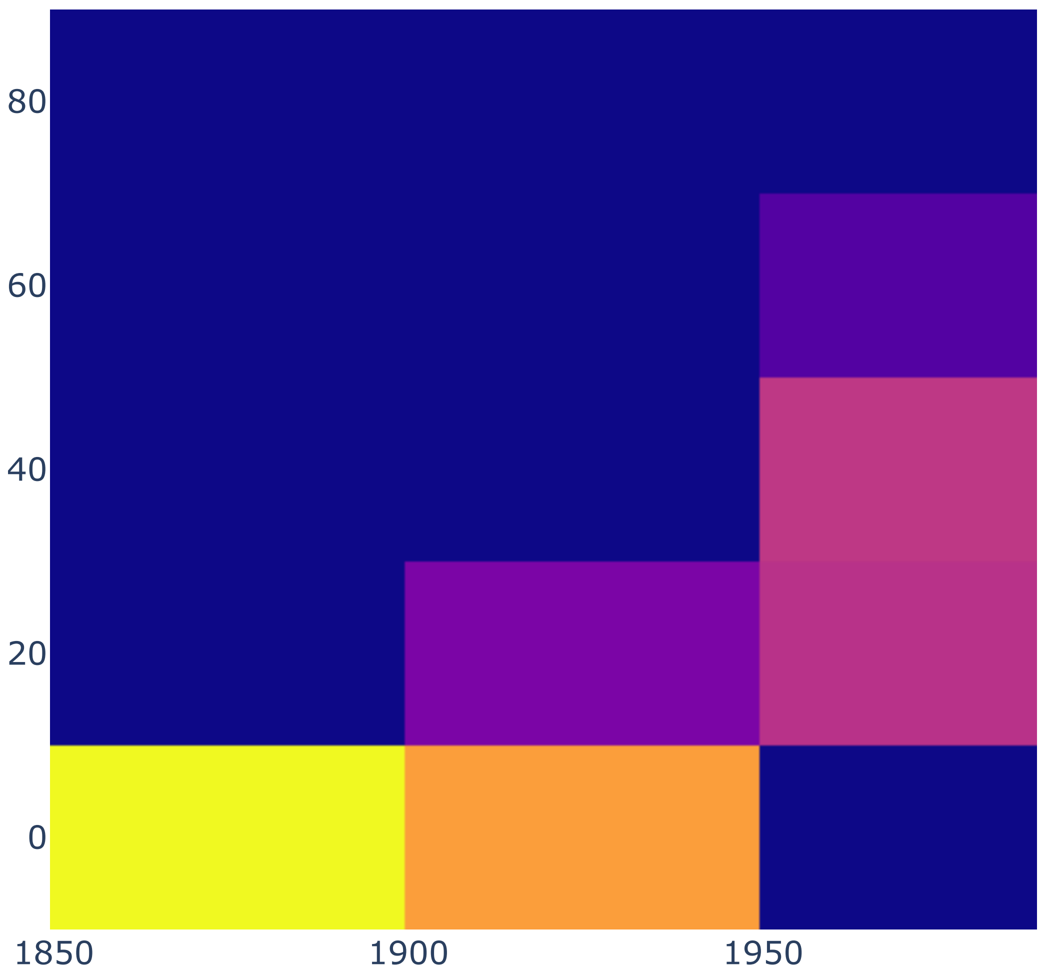
```
# Start your answer here, inserting more cells as you go along

import matplotlib.pyplot as plt
plt.figure(figsize=(20,8))
plt.hist2d(x = ny.yearbuilt,
        y = ny.numfloors,
        bins = [(2020-1850)/10,(120-20)/10],
        range = [[1850,2020],[20,120]])
plt.xlabel('Decade Built')
plt.ylabel('Number of Floors Built')
plt.colorbar()
plt.show()
```



From the 2D Histogram, we see that most buildings that are taller than 20 floors were built during 1960-1970, followed by the decade between 1920 and 1930. The first unusual tall building were built in the 1890's. However, we can't really tell the different counts between 0 and 50 for buildings above 50 stories. Thus, we would create two more 2D histogram focusing on the dark blue areas in the older time of history.

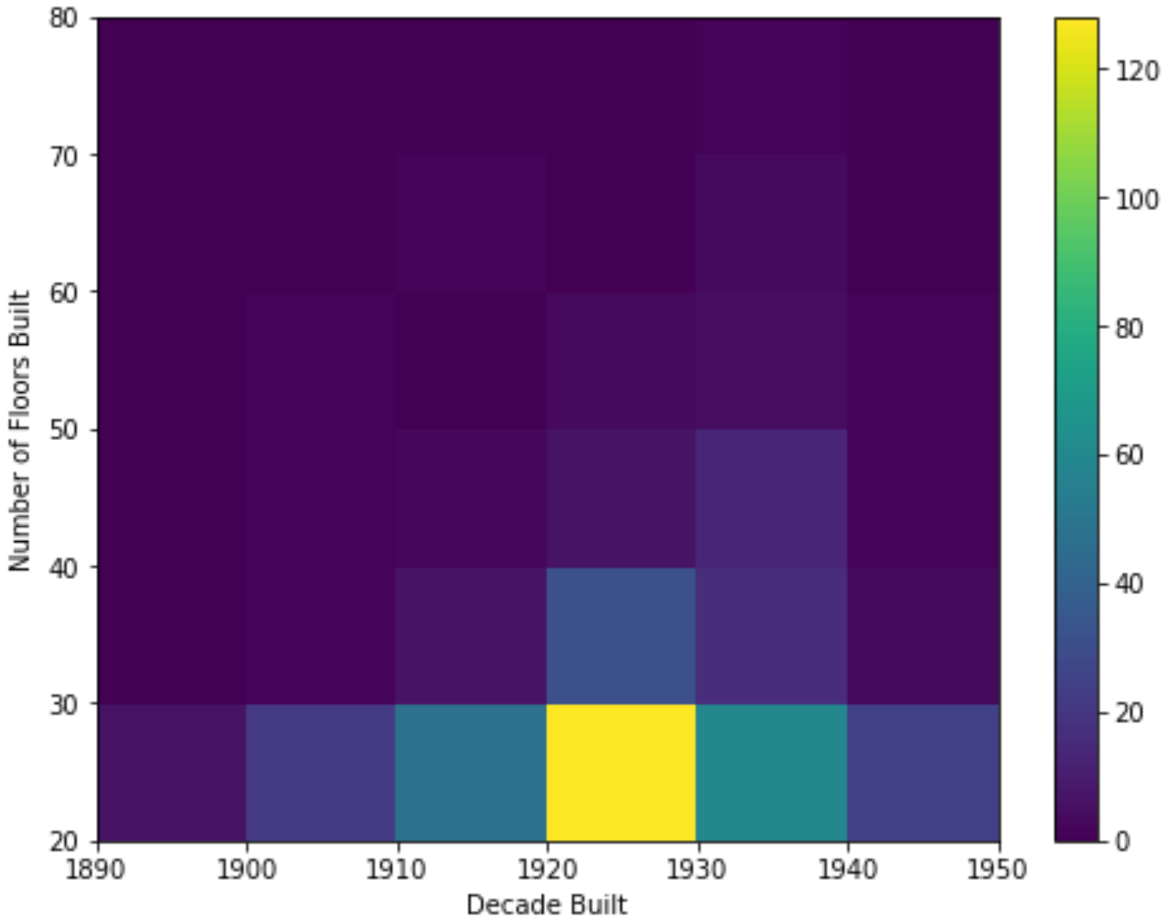
In [21]:

```
# Start your answer here, inserting more cells as you go along

fig = go.Figure(go.Histogram2d(
        x = ny.groupby('yearbuilt').count()['bbl'].index,
        y = ny.groupby('numfloors').count()['bbl'].index
    ))
fig.show()
```



In [150]:

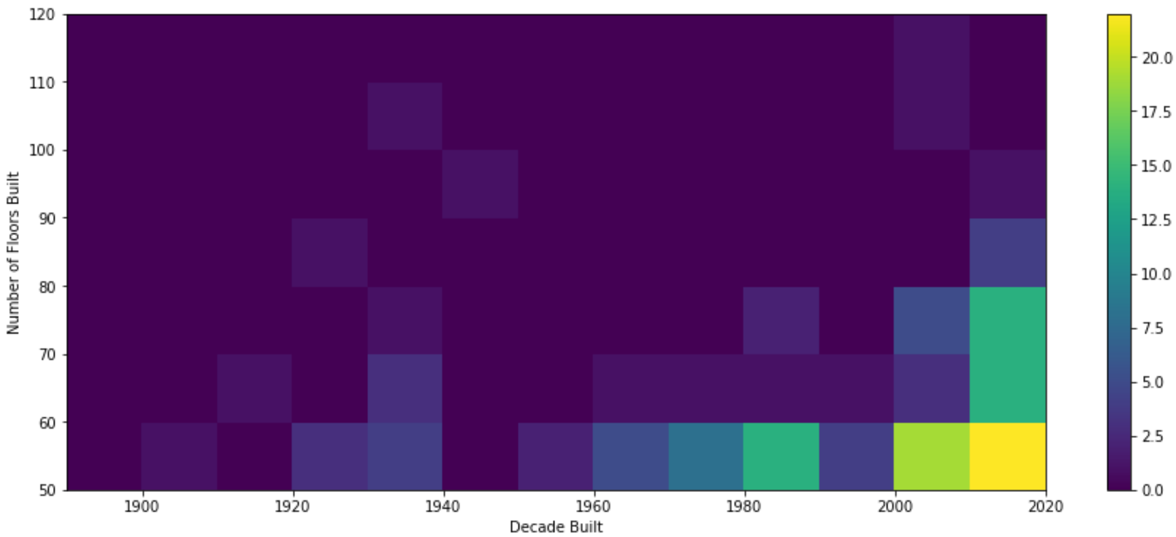
```
# The oldest tall buildings built before 1950

plt.figure(figsize=(8,6))
plt.hist2d(x = ny.yearbuilt,
        y = ny.numfloors,
        bins = [(1950-1890)/10,(80-20)/10],
        range = [[1890,1950],[20,80]])
plt.xlabel('Decade Built')
plt.ylabel('Number of Floors Built')
plt.colorbar()
plt.show()
```



Looking at the 2D histogram above, we see that most of the unusal tall buildings were built

In [151]:

```
# Buildings that have 50 or more floors

plt.figure(figsize=(15,6))
plt.hist2d(x = ny.yearbuilt,
        y = ny.numfloors,
        bins = [(2020-1890)/10,(120-50)/10],
        range = [[1890,2020],[50,120]])
plt.xlabel('Decade Built')
plt.ylabel('Number of Floors Built')
plt.colorbar()
plt.show()
```



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 [152]:

```
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['yearbu
ilt'].max(), yearbins))
ylabels = np.floor(np.logspace(1, np.log(ny['numfloors'].max()),
floorbins))

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

fig
```

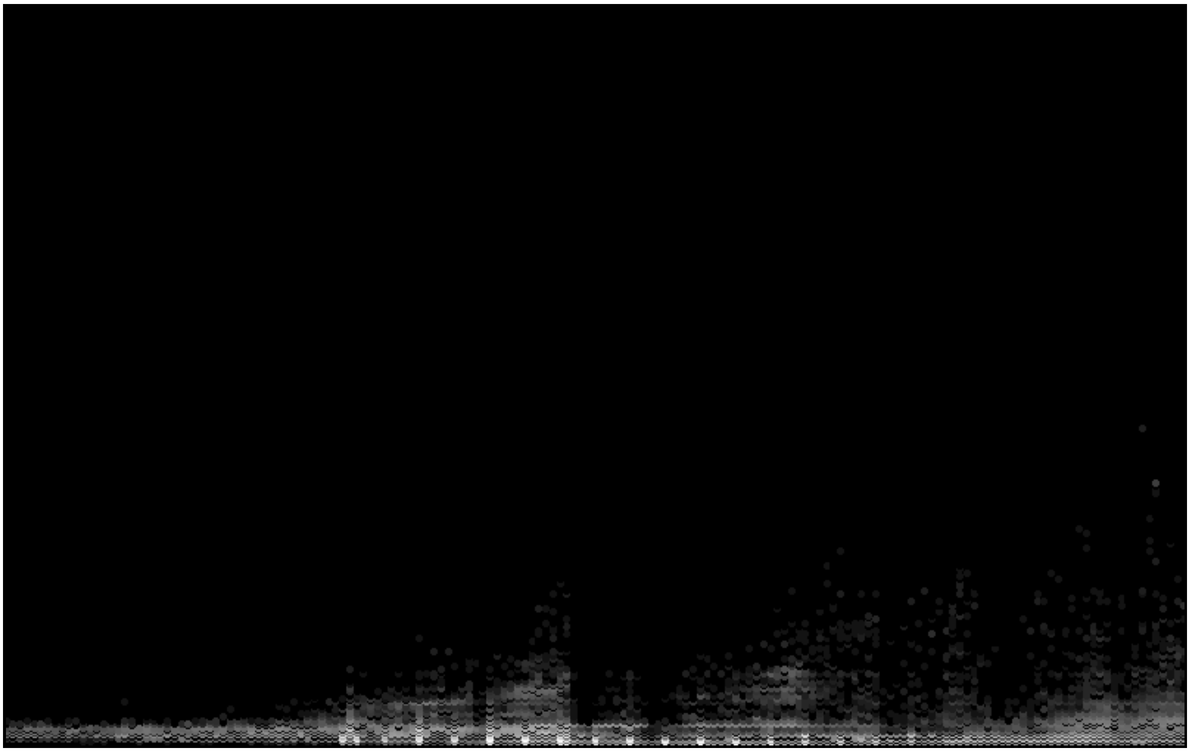
This shows us the distribution, but it's subject to some biases discussed in the Anaconda notebook [Plotting Perils (https://anaconda.org/jbednar/plotting_pitfalls/notebook)](https://anaconda.org/jbednar/plotting_pitfalls/notebook).

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

In [153]:

```
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[153]:



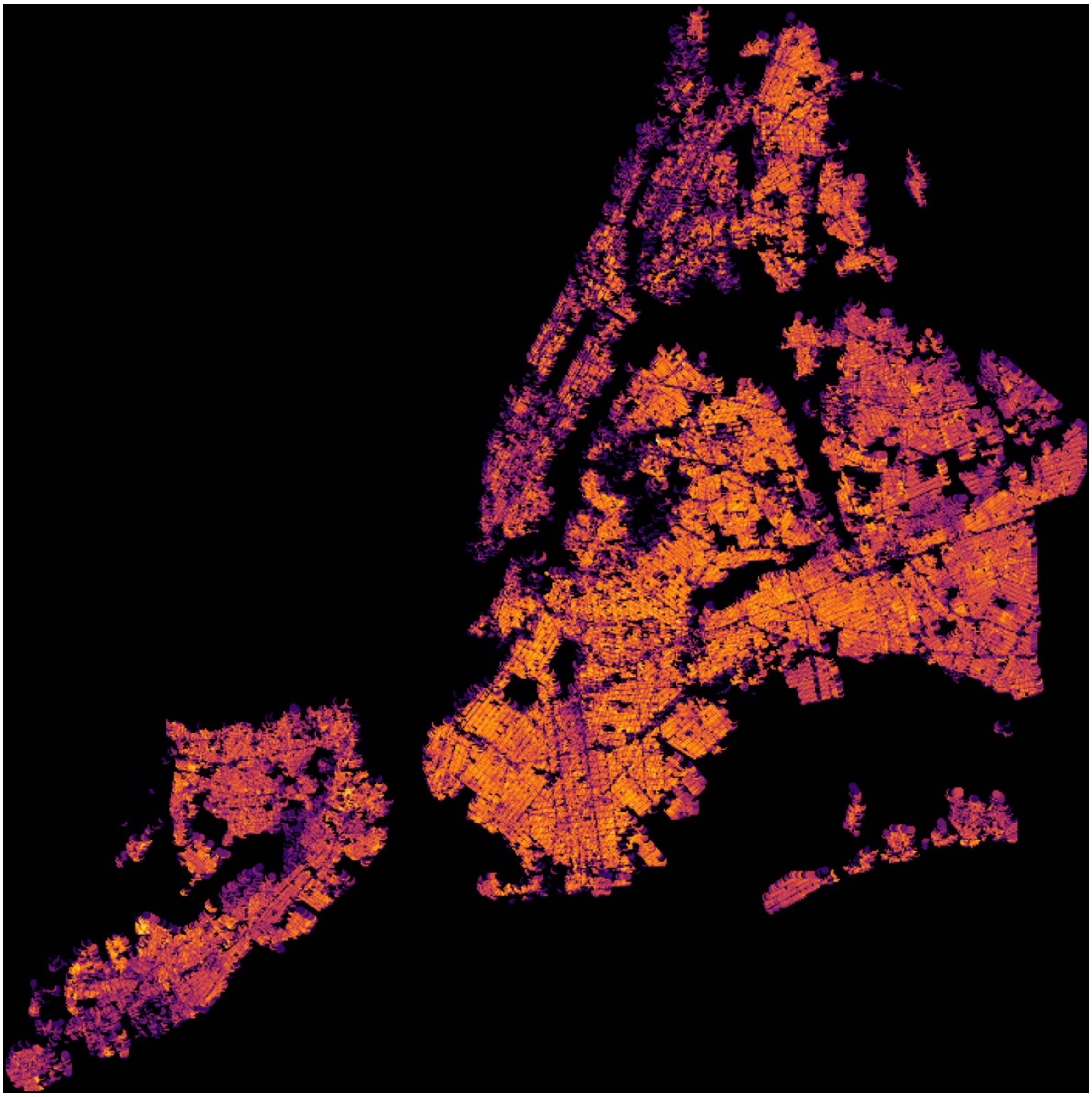
That's technically just a scatterplot, but the points are smartly placed and colored to mimic what one gets in a heatmap. Based on the pixel size, it will either display individual points, or will color the points of denser regions.

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 [154]:

```
NewYorkCity   = (( 913164.0,  1067279.0), (120966.0, 272275.0))
cvs = ds.Canvas(700, 700, *NewYorkCity)
agg = cvs.points(ny, 'xcoord', 'ycoord')
view = tf.shade(agg, cmap = cm(inferno), how='log')
export(tf.spread(view, px=2), 'firery')
```

Out[154]:



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). I would focus on the visualization pipeline (https://anaconda.org/jbednar/pipeline/notebook) and the US Census (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), 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/), 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.

In [161]:

```
ny.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 811684 entries, 0 to 859171
Data columns (total 99 columns):
borough                 811684 non-null object
block                   811684 non-null int64
lot                     811684 non-null int64
cd                      811384 non-null float64
ct2010                  811384 non-null float64
cb2010                  811384 non-null float64
schooldist              811341 non-null float64
council                 811384 non-null float64
zipcode                 811328 non-null float64
firecomp                811339 non-null object
policeprct              811341 non-null float64
healtharea              811341 non-null float64
sanitboro               811321 non-null float64
sanitsub                811252 non-null object
address                 811684 non-null object
zonedist1               811188 non-null object
zonedist2               17421 non-null object
zonedist3               119 non-null object
zonedist4               4 non-null object
overlay1                69789 non-null object
overlay2                152 non-null object
spdist1                 93003 non-null object
spdist2                 64 non-null object
spdist3                 0 non-null float64
ltdheight               2217 non-null object
splitzone               811188 non-null object
bldgclass               811684 non-null object
landuse                 811027 non-null float64
easements               811684 non-null float64
ownertype               16613 non-null object
ownername               811663 non-null object
lotarea                 811684 non-null float64
bldgarea                811668 non-null float64
comarea                 805300 non-null float64
resarea                 805300 non-null float64
officearea              805300 non-null float64
retailarea              805300 non-null float64
garagearea              805300 non-null float64
strgearea               805300 non-null float64
factryarea              805300 non-null float64
otherarea               805300 non-null float64
areasource              811684 non-null float64
```

```
numbldgs              811684 non-null float64
numfloors             811684 non-null float64
unitsres              811684 non-null float64
unitstotal            811684 non-null float64
lotfront              811684 non-null float64
lotdepth              811684 non-null float64
bldgfront             811684 non-null float64
bldgdepth             811684 non-null float64
ext                   790509 non-null object
proxcode              811684 non-null float64
irrlotcode            811684 non-null object
lottype               811684 non-null float64
bsmtcode              811684 non-null float64
assessland            811684 non-null float64
assesstot             811684 non-null float64
exempttot             811684 non-null float64
yearbuilt             811684 non-null float64
yearalter1            811684 non-null float64
yearalter2            811684 non-null float64
histdist              28556 non-null object
landmark              1205 non-null object
builtfar              811668 non-null float64
residfar              811684 non-null float64
commfar               811684 non-null float64
facilfar              811684 non-null float64
borocode              811684 non-null int64
bbl                   811684 non-null int64
condono               7944 non-null float64
tract2010             811384 non-null float64
xcoord                811380 non-null float64
ycoord                811380 non-null float64
latitude              811380 non-null float64
longitude             811380 non-null float64
zonemap               811192 non-null object
zmcode                14293 non-null object
sanborn               811097 non-null object
taxmap                811097 non-null float64
edesignum             0 non-null float64
appbbl                86573 non-null float64
appdate               86573 non-null object
plutomapid            811684 non-null int64
version               811684 non-null object
sanitdistrict         811321 non-null float64
healthcenterdistrict  811341 non-null float64
firm07_flag           26395 non-null float64
```

```
pfirm15_flag                55779 non-null float64
rpaddate                    0 non-null float64
dcasdate                    0 non-null float64
zoningdate                  0 non-null float64
landmkdate                  0 non-null float64
basempdate                  0 non-null float64
masdate                     0 non-null float64
polidate                    0 non-null float64
edesigdate                  0 non-null float64
geom                        811192 non-null object
dcpedited                   26815 non-null object
notes                       508 non-null float64
dtypes: float64(66), int64(5), object(28)
memory usage: 619.3+ MB
```

We will use columns "assessland", "assesstot", "longitude" and "latitude" from the ny dataset

In [184]:

```
# Build the dataframe
df = ny[['assessland', 'assesstot', 'longitude', 'latitude']]

# Assessment of Structure is calculated as the difference betwee
n total assessment and land assessment
assessstruct = df['assesstot'] - df['assessland']

# Add Structure Assessment column into the dataframe
df.insert(4, "AssessStruct", assessstruct)

# Show the first few rows
df.head()
```

Out[184]:

	assessland	assesstot	longitude	latitude	AssessStruct
0	146250.0	350550.0	-74.007347	40.637972	204300.0
1	12240.0	78900.0	-73.846003	40.786562	66660.0
2	18120.0	34380.0	-73.926923	40.653216	16260.0
3	7680.0	24600.0	-73.925958	40.623876	16920.0
4	8160.0	29760.0	-73.926030	40.623874	21600.0

In [192]:

```
pd.options.display.float_format = '{:20,.2f}'.format # remove th
e scienfific notation, see https://stackoverflow.com/questions/1
7737300/suppressing-scientific-notation-in-pandas

# Take a look at the statistics of structure assessment
df.AssessStruct.describe()
```

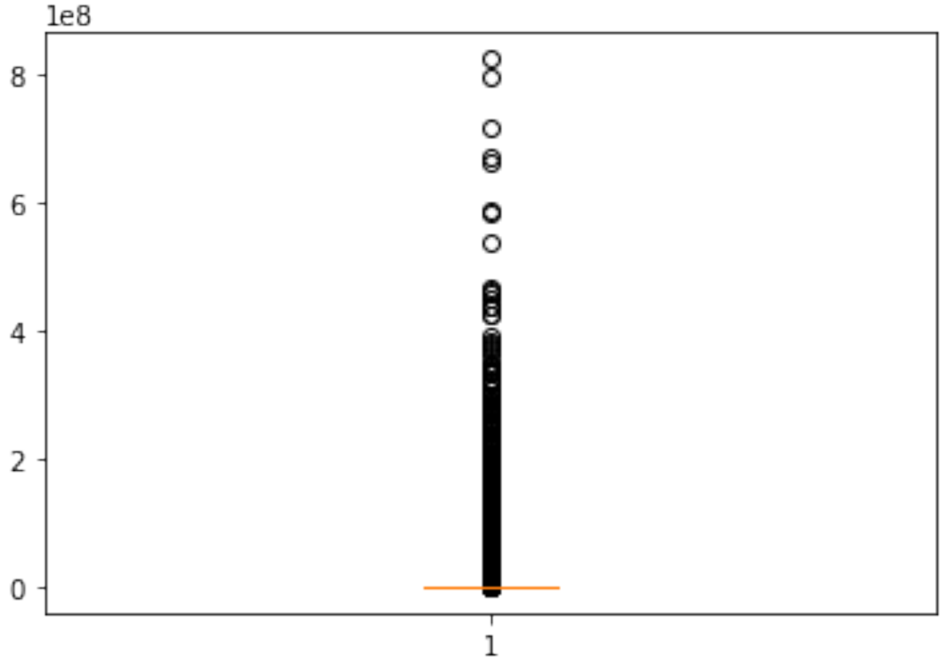
Out[192]:

```
count                811,684.00
mean                 412,635.06
std                7,106,367.80
min                        0.00
25%                   23,820.00
50%                   36,600.00
75%                   68,940.00
max            3,924,464,620.00
Name: AssessStruct, dtype: float64
```

The mean is much higher than the 75th percentile, implying that the data is heavily skewed. We confirm this with the maximum being near 4 billion. We also see minimum being zero, which wrongly suggests that the structure has no value. Compare the two boxplots below - the first one is not normally distributed; but when we take the natural log of the values, the second boxplot looks much better.

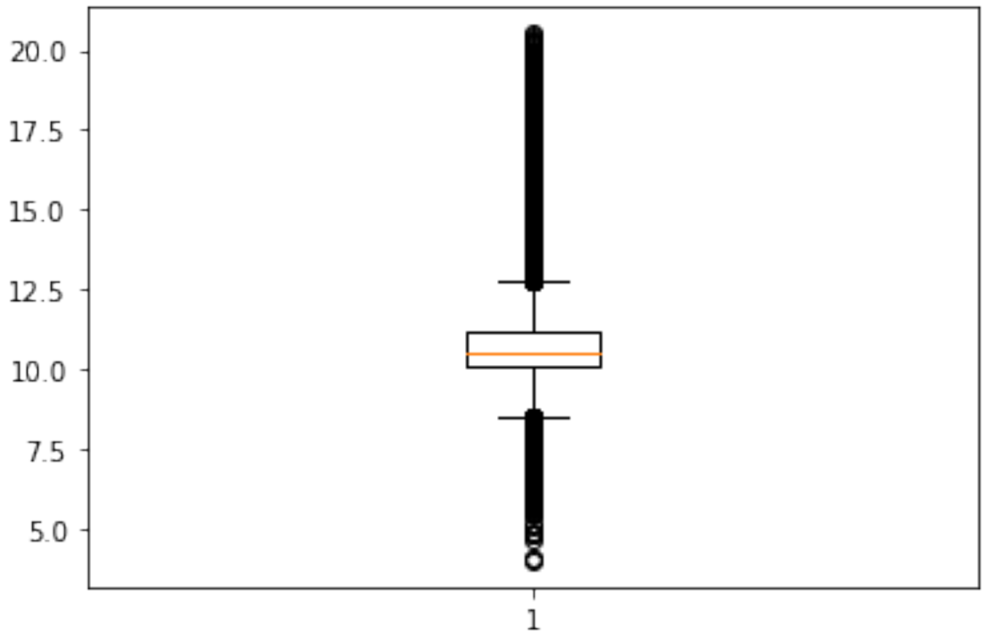
In [242]:

```
plt.boxplot(df.AssessStruct)
plt.show()
```



In [241]:

```
plt.boxplot(np.log(df.AssessStruct))
plt.show()
```



In [243]:

```
# Remove the zero land assessments
df = df.loc[df['assessland'] != 0]

# Take the natural log of land assessment
df['assessland'] = np.log(df['assessland'])

# Break land assessment into 3 categories
# df = df.assign(LogLand = np.log(df['assessland']))
df = df.assign(LandValue = pd.cut(df['assessland'],
                                  np.linspace(start = df['assess
land'].min(), stop = df['assessland'].max(), num = 3+1),
                                  labels=["L","M","H"]))

df.groupby('LandValue').size()
```

Out[243]:

```
LandValue
L    418317
M    385960
H      4898
dtype: int64
```

In [244]:

```
# Remove the zero structure assessments
df = df.loc[df['AssessStruct'] != 0]

# Take the natural log of structure assessment
df['AssessStruct'] = np.log(df['AssessStruct'])

df = df.assign(StructValue = pd.cut(df['AssessStruct'],
                                    np.linspace(start = df['Assess
Struct'].min(), stop = df['AssessStruct'].max(), num = 3+1),
                                    labels=["L","M","H"]))

df.groupby('StructValue').size()
```

Out[244]:

```
StructValue
L      18544
M     777510
H      13121
dtype: int64
```

In [253]:

```
# Combine land and structure categories into a total of 9 catego
ries
df = df.assign(LandStructV = df['LandValue'].astype(str) + df['S
tructValue'].astype(str)) # see https://stackoverflow.com/questi
ons/19377969/combine-two-columns-of-text-in-dataframe-in-pandas-
python

# Remove the NaNs
df = df.loc[df['LandStructV'] != 'Lnan']
df = df.loc[df['LandStructV'] != 'nanL']

df.groupby('LandStructV').size()
```

Out[253]:

```
LandStructV
HH        4427
HM         471
LL       13630
LM      404686
MH        8694
ML        4913
MM      372353
dtype: int64
```

In [254]:

```
# Turn LandStructV into categorical data
df['LandStructV'] = pd.Categorical(df['LandStructV'])

colorKeys = {'HH':'#3b4994', 'HM':'#5698b9', 'HL':'#5ac8c8',
             'MH':'#8c62aa', 'MM':'#a5add3', 'ML':'#ace4e4',
             'LH':'#be64ac', 'LM':'#dfb0d6', 'LL':'#e8e8e8'}

NYC = (( -74.29,  -73.69), (40.49, 40.92))
cvs = ds.Canvas(700, 700, *NYC)
agg = cvs.points(df, 'longitude', 'latitude', ds.count_cat('Land
StructV'))
view = tf.shade(agg, color_key = colorKeys)
export(tf.spread(view, px=1), 'Assessment')
```

Out[254]:

Purple areas represent both land and structure assessments are high, we can see mostly are in Manhattan.

Underbuilt - blue areas represent high land value and low structure value, mostly distributed among Brooklyn and Queens.

Overbuilt - pink areas represent low land value and high structure value, mostly distributed among Staten Island and upper Bronx.