Name: Chinedu Onyeka

Data608 - Module2 Assignment

In [1]:
```
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
from pyproj import Proj, transform
import numpy as np
import pandas as pd
import urllib
import json
import datetime
import colorlover as cl

from shapely.geometry import Point, Polygon, shape

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

# 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()
```

```
C:\Users\exper\Anaconda3\lib\site-packages\dask\dataframe\utils.py:369: FutureWarning: pandas.Int64Index is deprecated and will be removed from pandas in a future version. Use pandas.Index with the appropriate dtype instead.
  _numeric_index_types = (pd.Int64Index, pd.Float64Index, pd.UInt64Index)
C:\Users\exper\Anaconda3\lib\site-packages\dask\dataframe\utils.py:369: FutureWarning: pandas.Float64Index is deprecated and will be removed from pandas in a future version. Use pandas.Index with the appropriate dtype instead.
  _numeric_index_types = (pd.Int64Index, pd.Float64Index, pd.UInt64Index)
C:\Users\exper\Anaconda3\lib\site-packages\dask\dataframe\utils.py:369: FutureWarning: pandas.UInt64Index is deprecated and will be removed from pandas in a future version. Use pandas.Index with the appropriate dtype instead.
  _numeric_index_types = (pd.Int64Index, pd.Float64Index, pd.UInt64Index)
```

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. 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_22v2_csv/pluto_22v2.csv')


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

```
<ipython-input-2-6cb414ace903>:11: DtypeWarning:

Columns (21,22,24,26,28) have mixed types. Specify dtype option on import or set low_memory=False.
```

In [3]:
```
ny.columns
```

Out[3]:
```
Index(['borough', 'block', 'lot', 'cd', 'bct2020', 'bctcb2020', 'ct2010',
       'cb2010', 'schooldist', 'council', 'zipcode', 'firecomp', 'policeprct',
       'healthcenterdistrict', 'healtharea', 'sanitboro', 'sanitdistrict',
       'sanitsub', 'address', 'zonedist1', 'zonedist2', 'zonedist3',
       'zonedist4', 'overlay1', 'overlay2', 'spdist1', 'spdist2', 'spdist3',
       'ltdheight', 'splitzone', 'bldgclass', 'landuse', 'easements',
       'ownertype', 'ownername', 'lotarea', 'bldgarea', 'comarea', 'resarea',
       'officearea', 'retailarea', 'garagearea', 'strgearea', 'factryarea',
       'otherarea', 'areasource', 'numbldgs', 'numfloors', 'unitsres',
       'unitstotal', 'lotfront', 'lotdepth', 'bldgfront', 'bldgdepth', 'ext',
       'proxcode', 'irrlotcode', 'lottype', 'bsmtcode', 'assessland',
       'assesstot', 'exempttot', 'yearbuilt', 'yearalter1', 'yearalter2',
       'histdist', 'landmark', 'builtfar', 'residfar', 'commfar', 'facilfar',
       'borocode', 'bbl', 'condono', 'tract2010', 'xcoord', 'ycoord',
       'zonemap', 'zmcode', 'sanborn', 'taxmap', 'edesignum', 'appbbl',
       'appdate', 'plutomapid', 'firm07_flag', 'pfirm15_flag', 'version',
```

```
       'dcpedited', 'latitude', 'longitude', 'notes'],
      dtype='object')
```

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 [4]:
```
# 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(transform, background = "black", export_path="export")
# cm = partial(colormap_select, reverse=(background!="black"))
```

In [5]:
```
from pyproj import Transformer
transformer = Transformer.from_crs("epsg:4326", "epsg:3857")
transformer.transform(12, 12)
```

Out[5]:
```
(1335833.8895192828, 1345708.4084091089)
```

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 (also check out their close relatives: 2D density plots and the more general form: 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 [6]:
```
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: .loc and .iloc are the analogs for base R subsetting, or filter() in dplyr

Group By: 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().

Reset_index: 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 on datetime indexing. In particular, check out resample, which provides time series specific aggregation.

Merging, joining, and concatenation: 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 [7]:

```
# check the head for the ny data
ny.head()
```

Out[7]:

	borough	block	lot	cd	bct2020	bctcb2020	ct2010	cb2010	schooldist	council	...	appbbl	appdate	plutomapid	firm07_flag	pfirm15_flag	version	dcpedited	latitude	longitude	notes
0	SI	1597	125	502.0	5029104.0	5.029104e+10	291.04	3007.0	31.0	50.0	...	NaN	NaN	1	NaN	NaN	22v2	NaN	40.611140	-74.164376	NaN
2	BK	4794	1	309.0	3080600.0	3.080600e+10	806.00	2000.0	17.0	41.0	...	3.047940e+09	08/12/2005	1	NaN	NaN	22v2	NaN	40.661794	-73.942532	NaN
3	BK	1488	105	303.0	3037500.0	3.037500e+10	375.00	1001.0	16.0	41.0	...	3.014880e+09	11/04/1992	1	NaN	NaN	22v2	NaN	40.686484	-73.920169	NaN
4	BK	4794	17	309.0	3080600.0	3.080600e+10	806.00	2000.0	17.0	41.0	...	NaN	NaN	1	NaN	NaN	22v2	NaN	40.661859	-73.941991	NaN
5	BK	4794	78	309.0	3080600.0	3.080600e+10	806.00	2000.0	17.0	41.0	...	3.047940e+09	04/11/2006	1	NaN	NaN	22v2	NaN	40.661517	-73.942539	NaN

5 rows × 92 columns

In [8]:

```
# describe the number of floors
ny['numfloors'].describe()
```

Out[8]:

```
count    809793.000000
mean          2.459627
std           1.947155
min           1.000000
25%           2.000000
50%           2.000000
75%           2.750000
max         104.000000
Name: numfloors, dtype: float64
```

In [9]:

```
# describe the year built
ny['yearbuilt'].describe()
```

Out[9]:

```
count    812064.000000
mean       1941.167871
std          30.580862
min        1851.000000
25%        1920.000000
50%        1931.000000
75%        1960.000000
max        2019.000000
Name: yearbuilt, dtype: float64
```

In [10]:

```
# binning the data
bins = [0, 9, 19, 29, 39, 49, 110]
floor_labels = ['1-9','10-19','20-29','30-39','40-49','50-110']


ny['floor_bins'] = pd.cut(ny['numfloors'], bins=bins, labels=floor_labels)
decade_labels = pd.Series(np.arange(1850,2020,10))
ny['decade'] = pd.cut(ny['yearbuilt'], bins=pd.Series(np.arange(1849,2029,10)), labels=decade_labels)
ny[['yearbuilt','decade','floor_bins']].head()
```

Out[10]:

	yearbuilt	decade	floor_bins
0	1965.0	1960	1-9
2	1899.0	1890	1-9
3	1992.0	1990	1-9
4	1991.0	1990	1-9
5	2005.0	2000	1-9

In [11]:

```
# group into decades
ny_group = ny.groupby(['decade', 'floor_bins']).size().reset_index(name='count')

ny_group.head()
```

Out[11]:

	decade	floor_bins	count
0	1850	1-9	1566
1	1850	10-19	2
2	1850	20-29	0
3	1850	30-39	0
4	1850	40-49	0

In [12]:

```
# Drop rows with zero count
ny_group.drop(ny_group.loc[ny_group['count']==0].index, inplace=True)

ny_group['count_base10'] = np.log10(ny_group['count'])

# order floor bins in numerical order
ny_group.sort_values('floor_bins', inplace=True)
```

```
ny_group.head()
```

Out[12]:

	decade	floor_bins	count	count_base10
0	1850	1-9	1566	3.194792
96	2010	1-9	14763	4.169175
90	2000	1-9	42439	4.627765
84	1990	1-9	28790	4.459242
78	1980	1-9	26804	4.428200

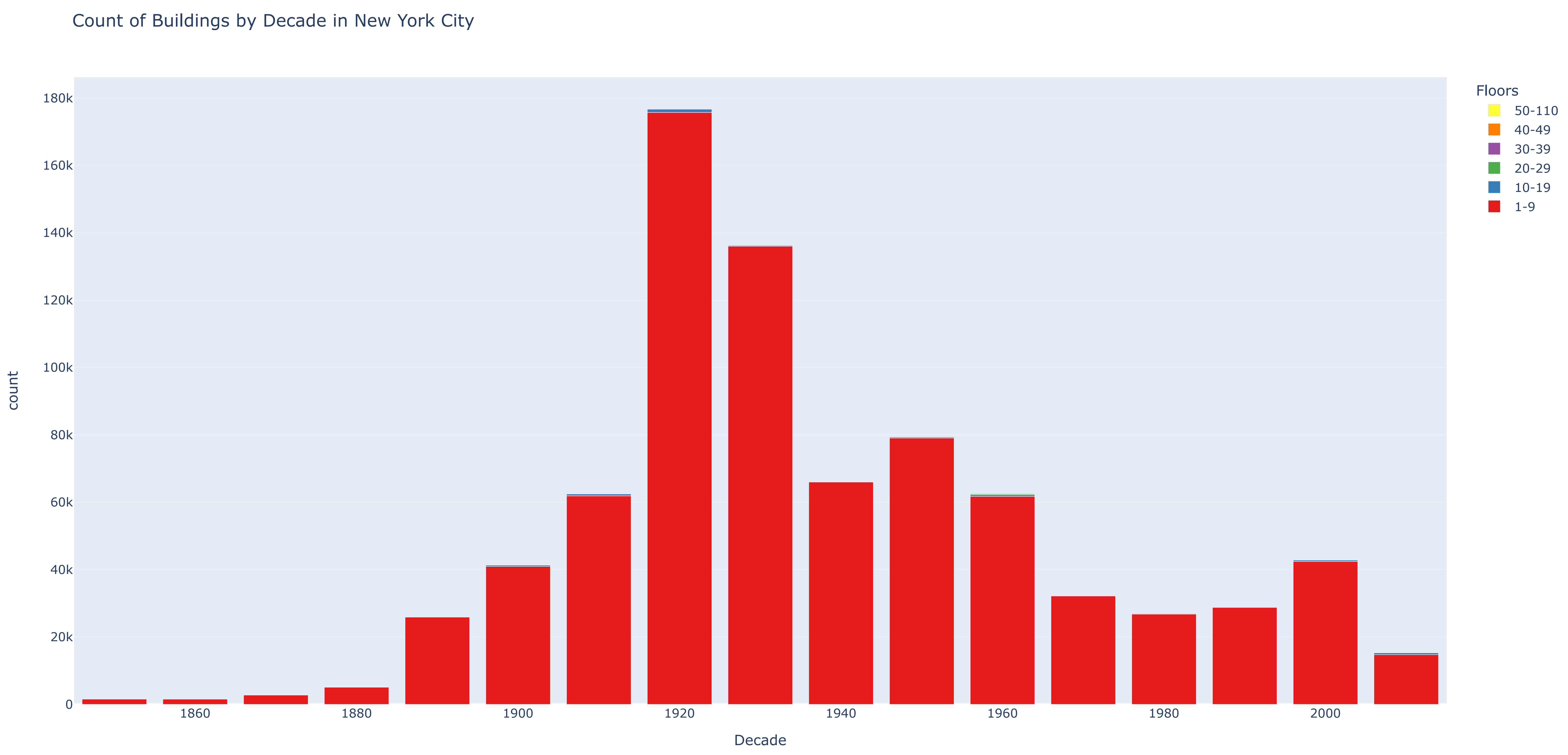
In [13]:

```
# Graph the stacked bar chart without log of counts

# Relabel the axis and legend appropriately

fig_ny = px.bar(ny_group, x="decade", y="count", color="floor_bins",
               color_discrete_sequence=px.colors.qualitative.Set1,
               height=800,
               title="Count of Buildings by Decade in New York City",
               labels={
                  "decade": "Decade",
                  "count_base10": "Count",
                  "floor_bins": "Floors"
               })

# Order the legend to match the chart and order of numbers
fig_ny.update_layout(legend_traceorder="reversed")
fig_ny.show()
```



We can see that the other floors are not clearly visible if we do not take the log of the counts.

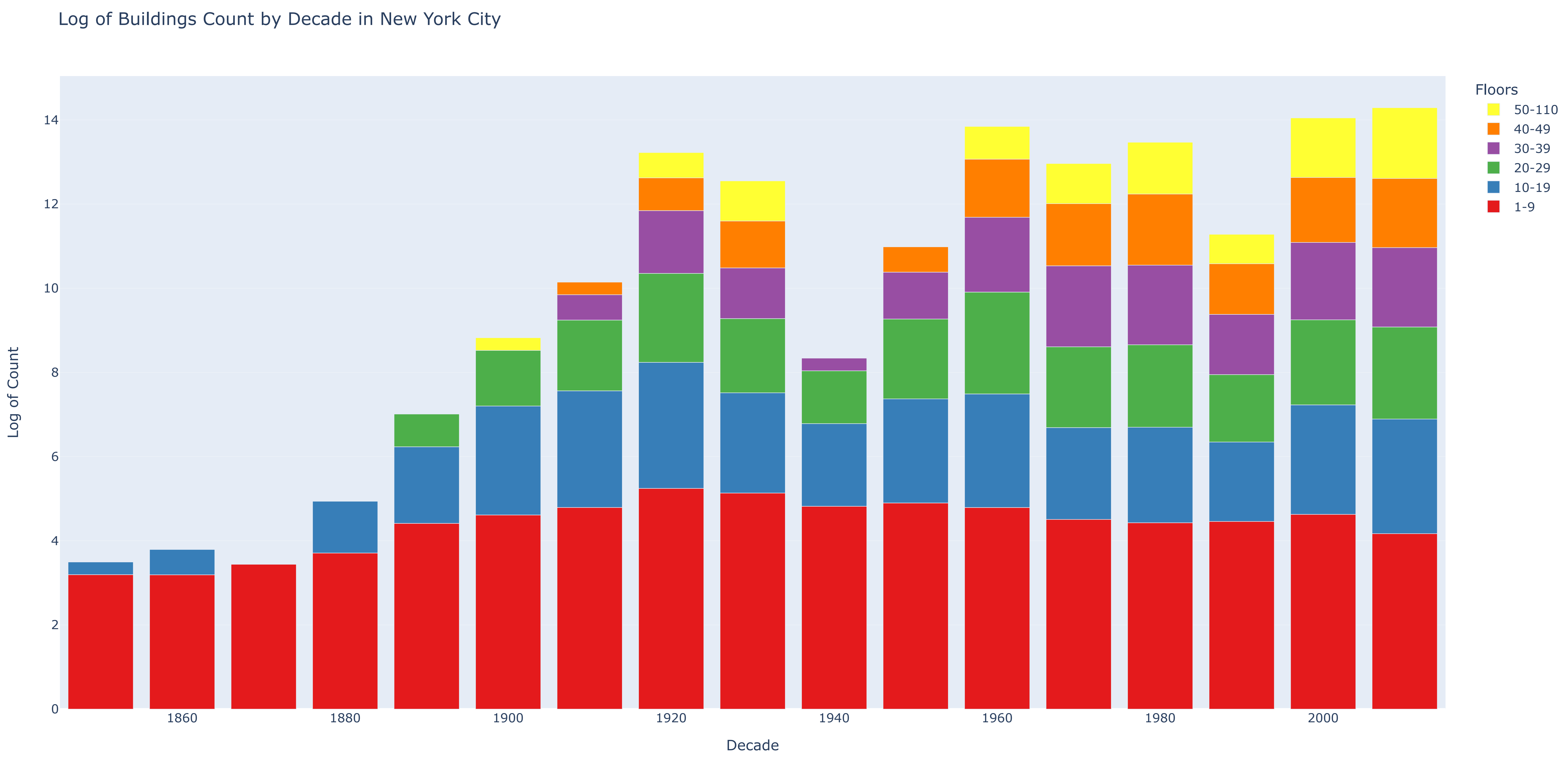
In [14]:

```
# Graph the stacked bar chart the log of count

fig_ny = px.bar(ny_group, x="decade", y="count_base10", color="floor_bins",
               color_discrete_sequence=px.colors.qualitative.Set1,
               height=800,
               title="Log of Buildings Count by Decade in New York City",
```

```
            labels={
               "decade": "Decade",
               "count_base10": "Log of Count",
               "floor_bins": "Floors"
            })

# Order the legend to match the chart and order of numbers
fig_ny.update_layout(legend_traceorder="reversed")
fig_ny.show()
```



We can see that the other floors are now clearly visible with the log of counts used as the y-axis

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 [15]:
```
fig = go.FigureWidget(
    data = [
        go.Histogram2d(x=ny['yearbuilt'], y=ny['numfloors'], autobiny=False, ybins={'size': 1}, colorscale='Greens')
    ]
)

fig
```

This shows us the distribution, but it's subject to some biases discussed in the Anaconda notebook Plotting Perils.

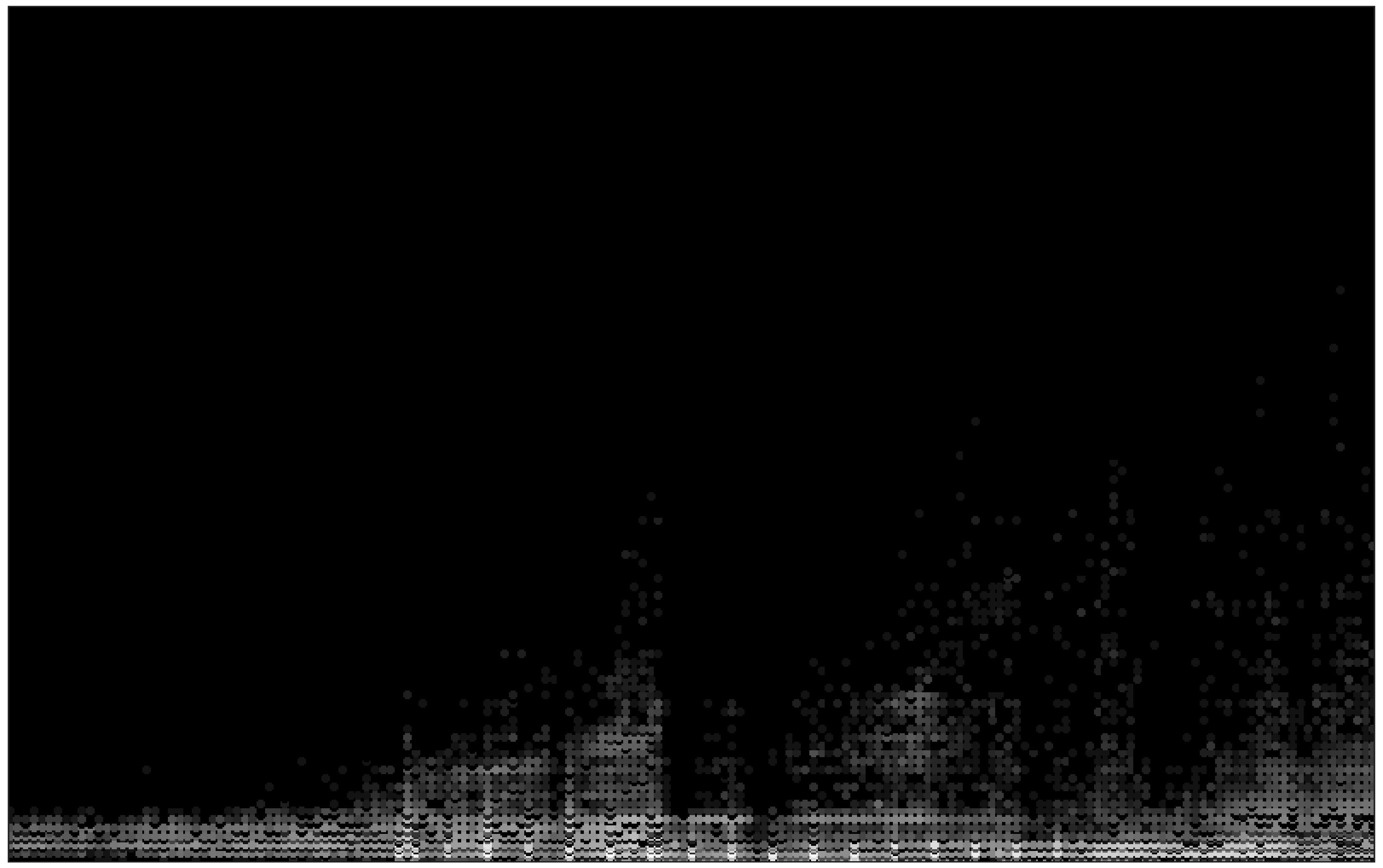
Here is what the same plot would look like in datashader:

In [16]:
```
# Defining some helper functions for DataShader
background = "black"
export = partial(export_image, background = background, export_path="export")
cm = partial(colormap_select, reverse=(background!="black"))

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[16]:



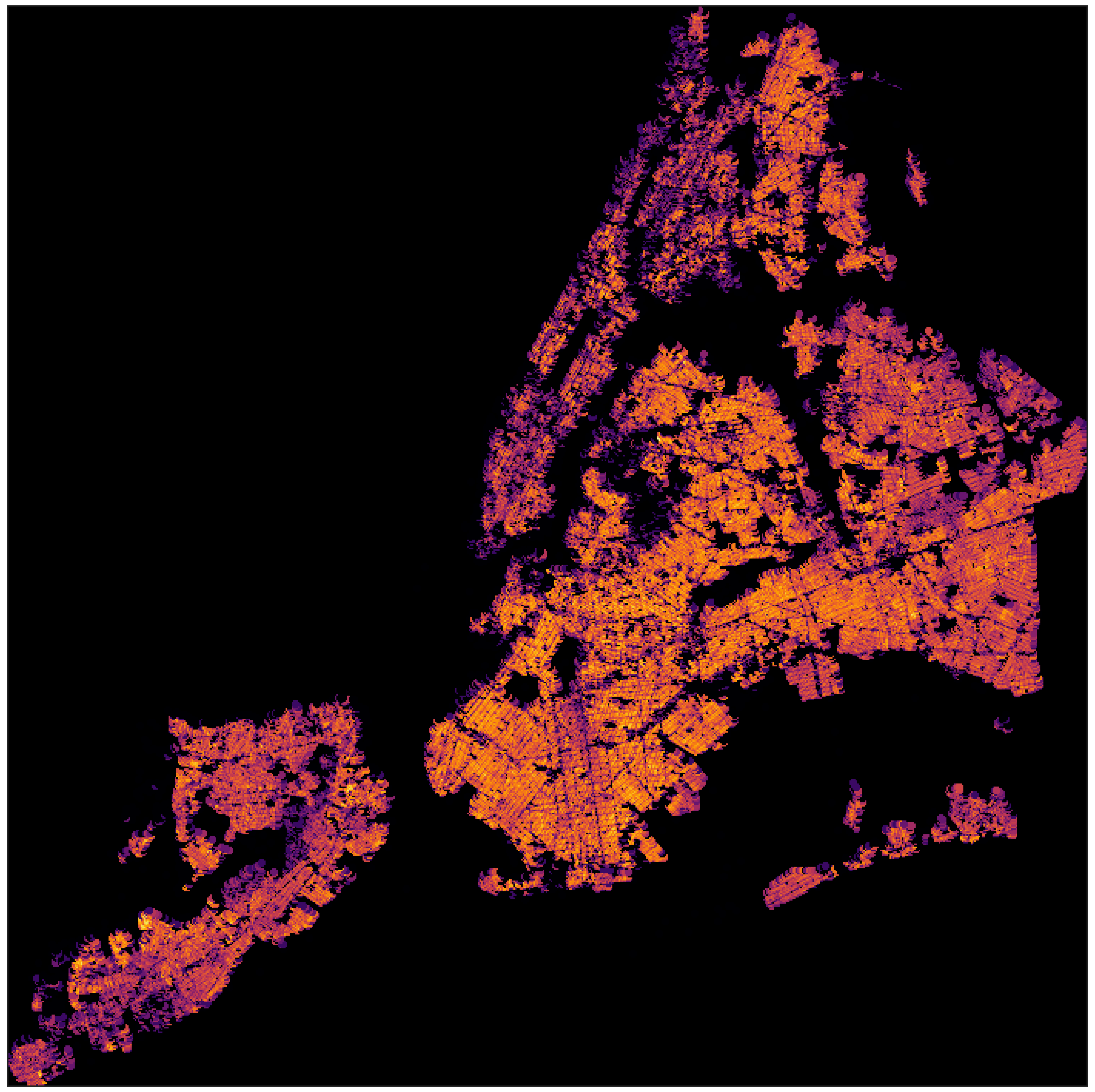
That's technically just a scatterplot, but the points are smartly placed and colored to mimic what one gets in a heatmap. Based on the pixel size, it will either display individual points, or will color the points of denser regions.

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 [17]:

```
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[17]:



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. I would focus on the visualization pipeline and the US Census 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, 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, 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 [18]:

```
from matplotlib import cm as mcm

# cmap as red
NewYorkCity   = (( 913164.0,  1067279.0), (120966.0, 272275.0))
cvs = ds.Canvas(700, 700, *NewYorkCity)
agg = cvs.points(ny, 'xcoord', 'ycoord', ds.sum('assesstot'))
nyc_image1 = tf.shade(agg, cmap = mcm.Reds, how='eq_hist', alpha=100)

export(nyc_image1, 'Assessment Total Value')
```

Out[18]:

From the map, we can see that Manhattan is overbuilt compared to the rest of the city. Also, places near and around Manhattan (portions of Brooklyn, Queens, and the Bronx that are closer to Manhattan) appear to be over-built as well, but as you move further away from Manhattan, you start to see areas that are underbuilt. For example, A little north of Bronx appear to be under-built, also, Broklyn appear to be overbuilt compared to Queens and Staten Island which is just south of Manhattan and Brooklyn appear to be the least built (most under-built) compared to the rest of the city.

In [19]:

```
# cmap as green
NewYorkCity   = (( 913164.0,  1067279.0), (120966.0, 272275.0))
cvs = ds.Canvas(700, 700, *NewYorkCity)
agg = cvs.points(ny, 'xcoord', 'ycoord', ds.sum('assessland'))
nyc_image2 = tf.shade(agg, cmap = mcm.Greens, how='eq_hist', alpha=100)

export(nyc_image2, 'Assessment Land Value')
```

Out[19]: