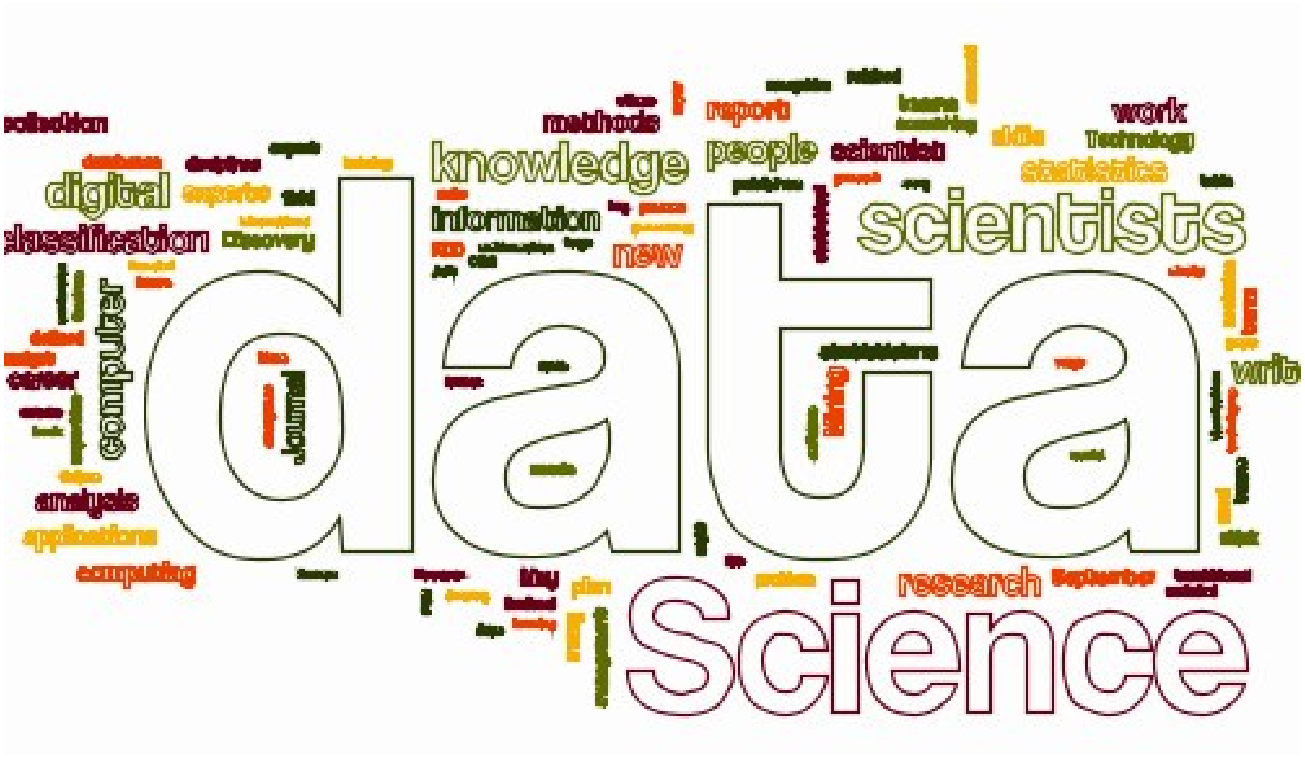
CSC 1315!

Data Science



Data Visualization

Based on:

Python for Data Analysis: http://hamelg.blogspot.com/2015/
Learning IPython for Interactive Computation and Visualization by C. Rossant

Plotting with pandas

- Visualizations are one of the most powerful tools at your disposal for exploring data and communicating data insights.
- The pandas library includes basic plotting capabilities that let you create a variety of plots from *DataFrames*.
- Plots in pandas are built on top of a popular Python plotting library called `matplotlib`, which comes with the Anaconda Python distribution.
- We start by loading some packages:

```
import numpy as np
import pandas as pd
import matplotlib
%matplotlib inline
```

- The `%matplotlib inline` tells `matplotlib` to render figures as static images in the Notebook

Diamond Dataset

- We are going to look at the diamonds data set that is provided in blackboard as a CSV file.
- Let's explore the structure of the data before going any further.

Diamond Dataset

In[1]:

```
diamonds.shape # Check data shape
```

Out[1]:

```
(53940, 10)
```

In[2]:

```
diamonds.head(5)
```

Out[2]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75

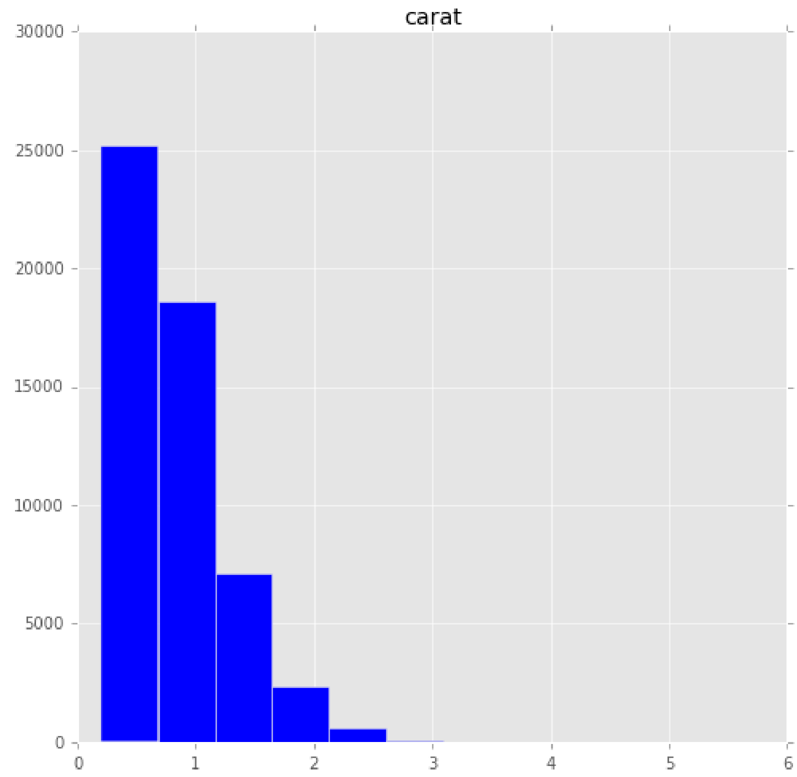
- The output shows that data set contains 10 features of 53940 different diamonds, including both numeric and categorical variables.

Histograms

- A histogram is a univariate plot (a plot that displays one variable) that groups a numeric variable into *bins* and displays the number of observations that fall within each bin.
- A histogram is a useful tool for getting a sense of the distribution of a numeric variable.
- Let's create a histogram of diamond *carat weight* with the `df.hist()` function:

Diamonds Histogram

```
diamonds.hist(column="carat",          # Column to plot
              figsize=(8,8),           # Plot size
              color="blue")            # Plot color
```

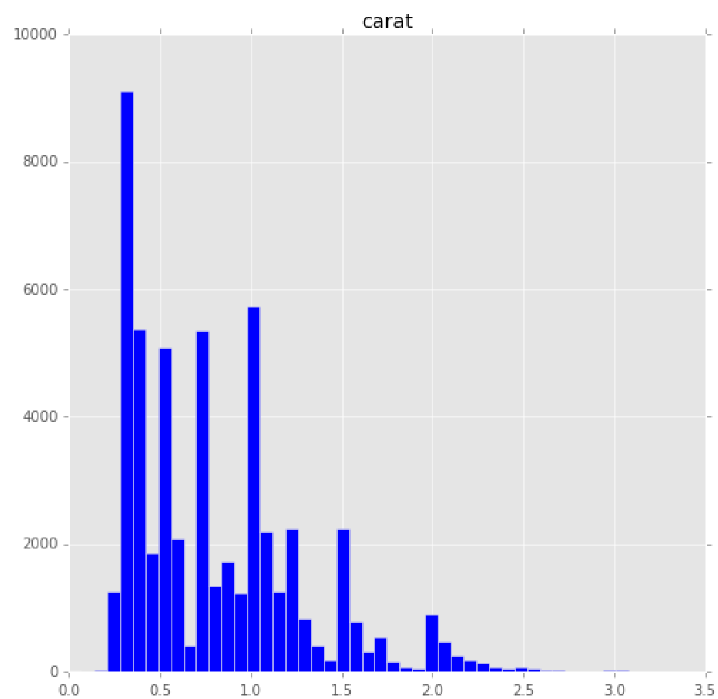


We see immediately that the carat weights are positively skewed: most diamonds are around 1 carat or below but there are extreme cases of larger diamonds.

Diamonds Histogram

- The previous plot has fairly wide bins and there doesn't appear to be any data beyond a carat size of 3.5.
- We can try to get more out of hour histogram by adding some additional arguments to control the size of the bins and limits of the x-axis:

```
diamonds.hist(column="carat",              # Column to plot
                      figsize=(8,8),        # Plot size
                      color="blue",         # Plot color
                      bins=50,              # Use 50 bins
                      range= (0,3.5))       # Limit x-axis range
```



Choosing Chart Range

- This histogram gives us a better sense of some subtleties within the distribution, but we can't be sure that it contains all the data.
- Limiting the X-axis to 3.5 might have cut out some outliers with counts so small that they didn't show up as bars on our original chart.
- Let's check to see if any diamonds are larger than 3.5 carats:

```
diamonds[diamonds["carat"] > 3.5]
```

	carat	cut	color	clarity	depth	table	price	x	y	z
23644	3.65	Fair	H	I1	67.1	53	11668	9.53	9.48	6.38
25998	4.01	Premium	I	I1	61.0	61	15223	10.14	10.10	6.17
25999	4.01	Premium	J	I1	62.5	62	15223	10.02	9.94	6.24
26444	4.00	Very Good	I	I1	63.3	58	15984	10.01	9.94	6.31
26534	3.67	Premium	I	I1	62.4	56	16193	9.86	9.81	6.13
27130	4.13	Fair	H	I1	64.8	61	17329	10.00	9.85	6.43
27415	5.01	Fair	J	I1	65.5	59	18018	10.74	10.54	6.98
27630	4.50	Fair	J	I1	65.8	58	18531	10.23	10.16	6.72
27679	3.51	Premium	J	VS2	62.5	59	18701	9.66	9.63	6.03

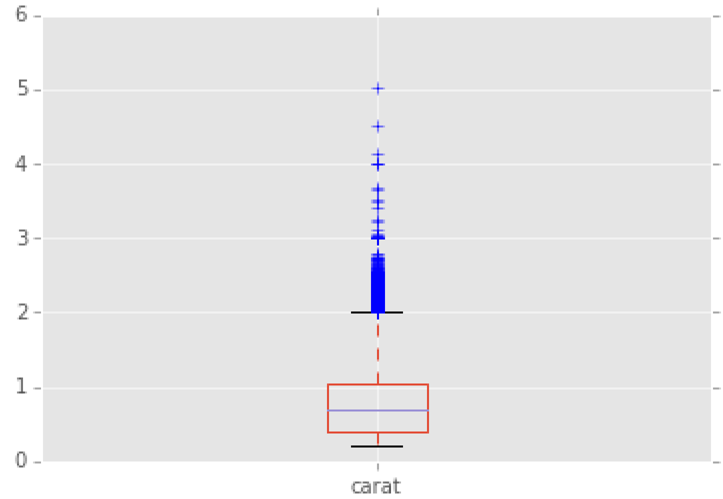
Not for Outliers

- It turns out that 9 diamonds are bigger than 3.5 carats.
- Should cutting these diamonds out concern us?
- On one hand, these outliers have very little bearing on the shape of the distribution.
- On the other hand, limiting the X-axis to 3.5 implies that no data lies beyond that point.
- For our own exploratory purposes this is not an issue but if we were to show this plot to someone else, it could be misleading.
- Including a note that *9 diamonds lie beyond the chart range* could be helpful.

Boxplots

- Boxplots are another type of univariate plot for summarizing distributions of numeric data graphically.
- Let's make a boxplot of carat using the `pd.boxplot()` function:

```
diamonds.boxplot(column="carat")
```



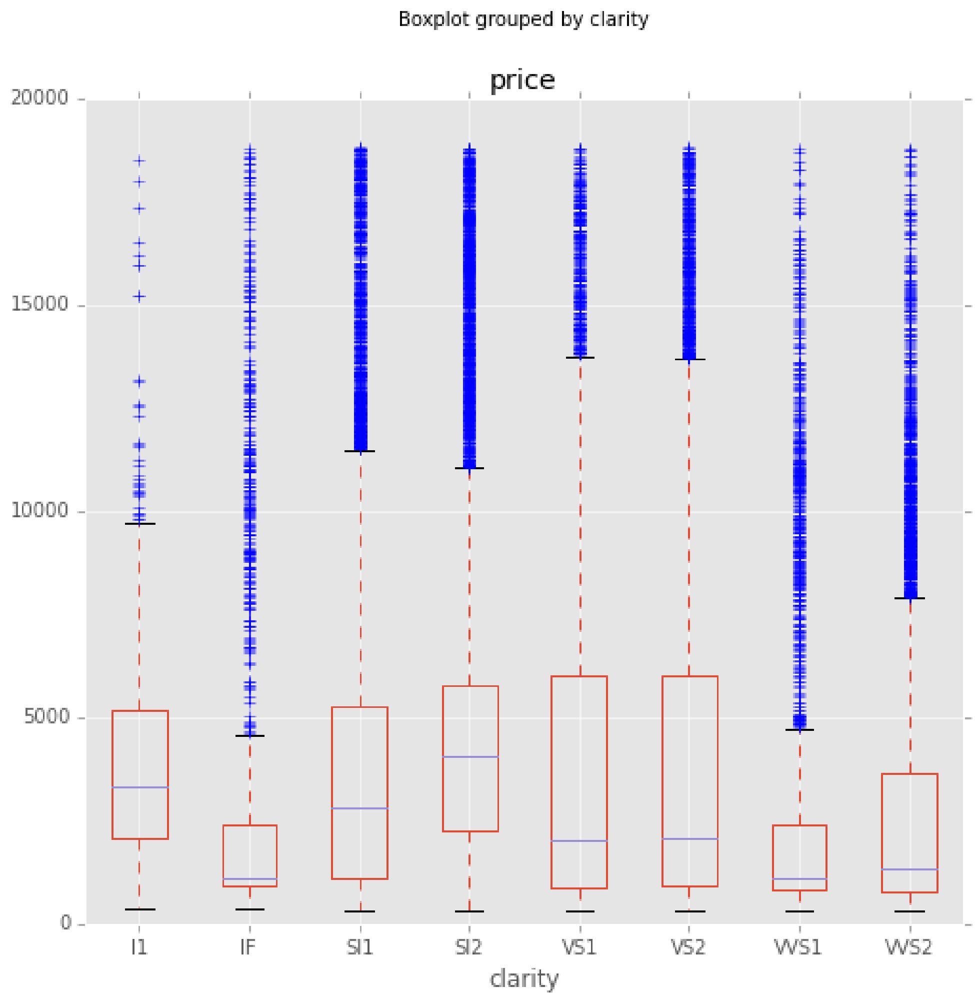
- The central box of the boxplot represents the middle 50% of the observations, the central bar is the median and the bars at the end of the dotted lines (whiskers) encapsulate the great majority of the observations.
- Circles that lie beyond the end of the whiskers are data points that may be outliers.

Side-by-side boxplot

- In this case, our data set has over 50,000 observations and we see many data points beyond the top whisker.
- We probably wouldn't want to classify all of those points as outliers, but the handful of diamonds at 4 carats and above are definitely far outside the norm.
- One of the most useful features of a boxplot is the ability to make side-by-side boxplots. A
- side-by-side boxplot takes a numeric variable and splits it on based on some categorical variable, drawing a different boxplot for each level of the categorical variable.

```
diamonds.boxplot(column="price",    # Column to plot
                 by= "clarity",     # Column to split upon
                 figsize= (8,8))    # Figure size
```

Side-by-side boxplot

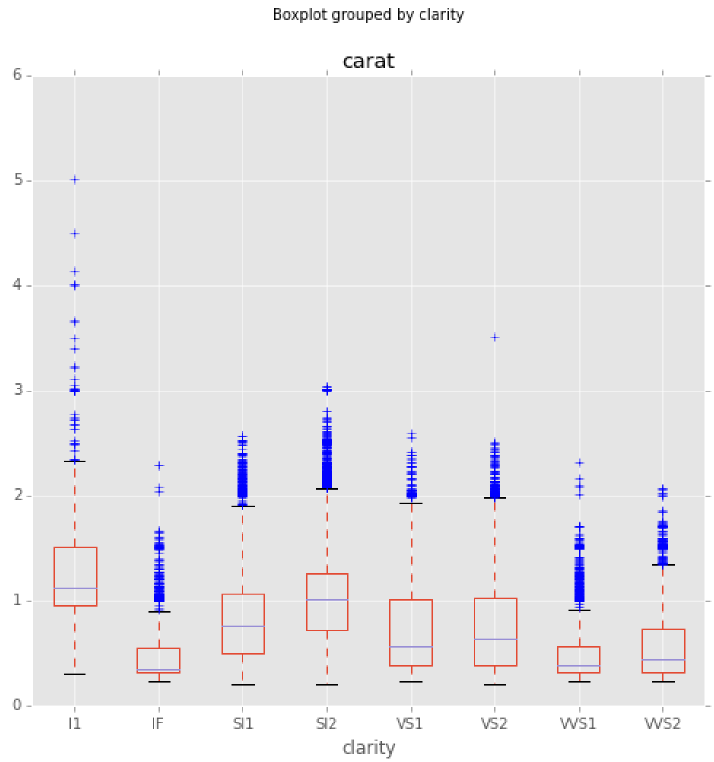


The boxplot above is curious: we'd expect diamonds with better clarity to fetch higher prices and yet diamonds on the highest end of the clarity spectrum (IF = internally flawless) actually have lower median prices than low clarity diamonds!

Side-by-side boxplot

- Perhaps another boxplot can shed some light on this situation:

```
diamonds.boxplot(column="carat",      # Column to plot
                 by= "clarity",       # Column to split upon
                 figsize= (8,8))      # Figure size
```

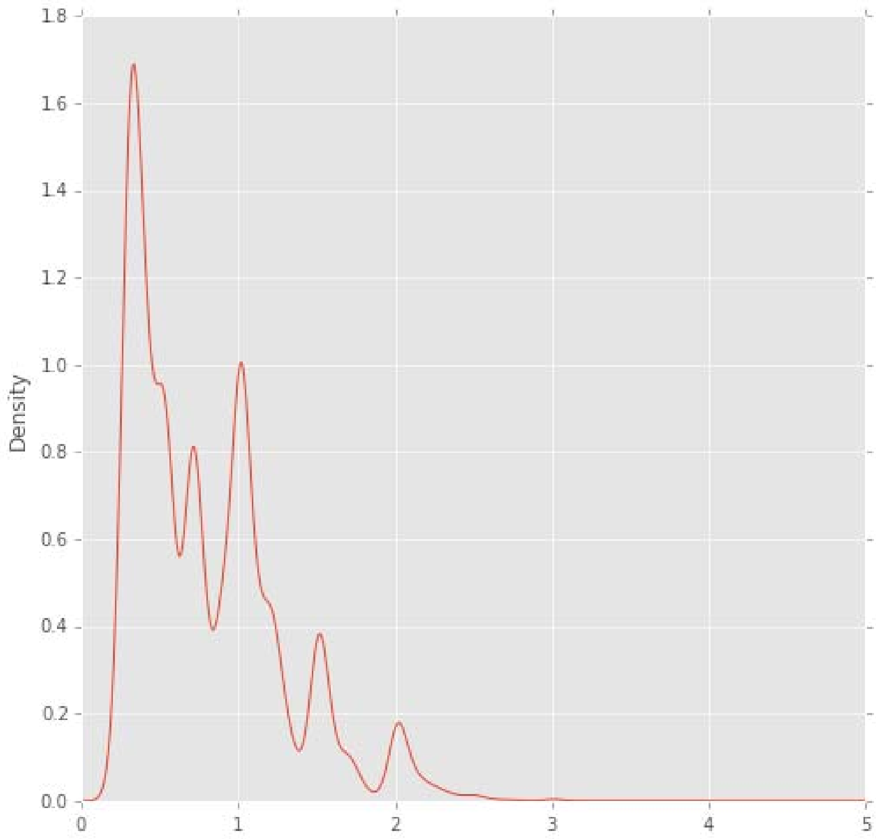


- The plot shows that diamonds with low clarity ratings also tend to be larger.
- Since size is an important factor in determining a diamond's value, it isn't too surprising that low clarity diamonds have higher median prices.

Density Plots

- A density plot shows the distribution of a numeric variable with a continuous curve.
- It is similar to a histogram but without discrete bins, a density plot gives a better picture of the underlying shape of a distribution.
- Create a density plot with series.plot(kind="density")

```
diamonds["carat"].plot(kind="density",  # Create density plot
                       figsize=(8,8),    # Set figure size
                       xlim= (0,5))      # Limit x axis values
```



Barplots

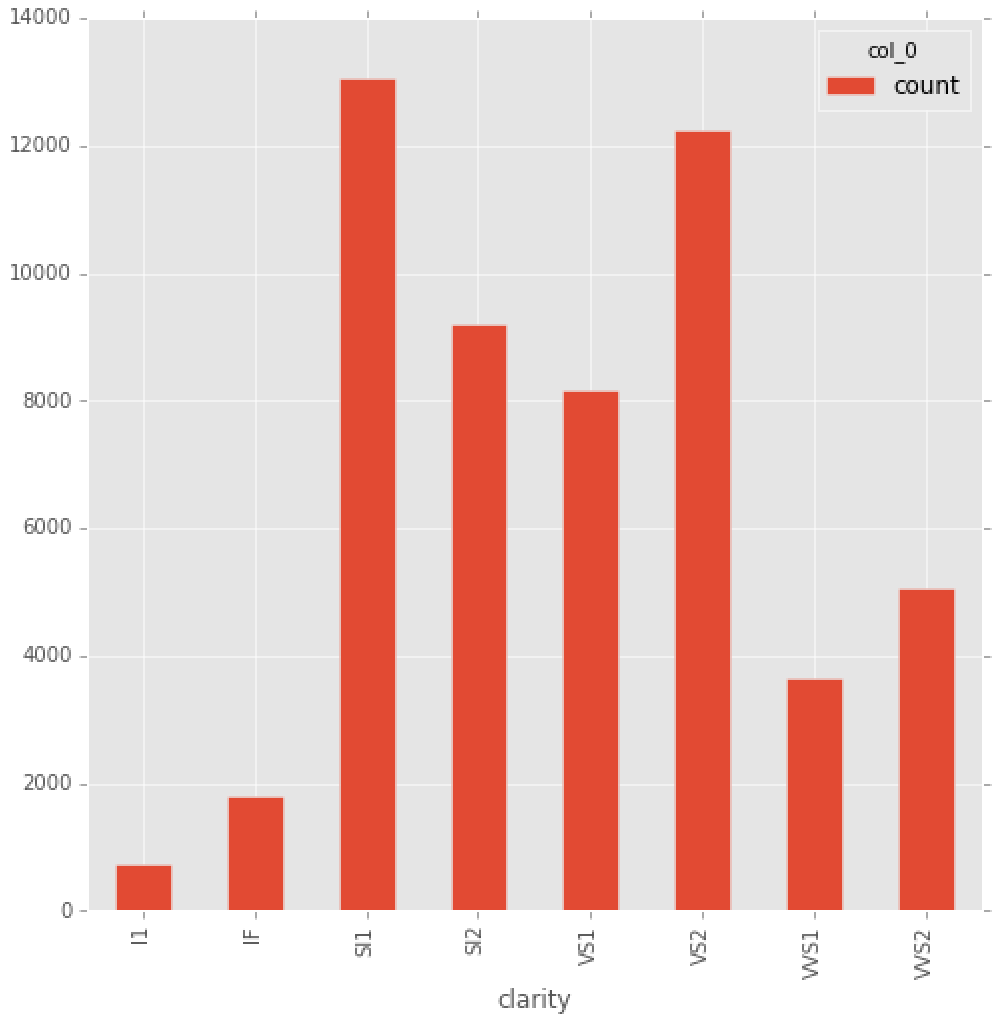
- Barplots are graphs that visually display counts of categorical variables.
- We can create a barplot by creating a table of counts for a certain variable using the `pd.crosstab()` function and then passing the counts to `df.plot(kind="bar")`:

```
carat_table = pd.crosstab(index=diamonds["clarity"],
                          columns="count")

carat_table
```

col_0	count
clarity	
I1	741
IF	1790
SI1	13065
SI2	9194
VS1	8171
VS2	12258
VVS1	3655
VVS2	5066

Barplots

```
carat_table.plot(kind="bar",
                 figsize=(8,8))
```



Stacked Barplots

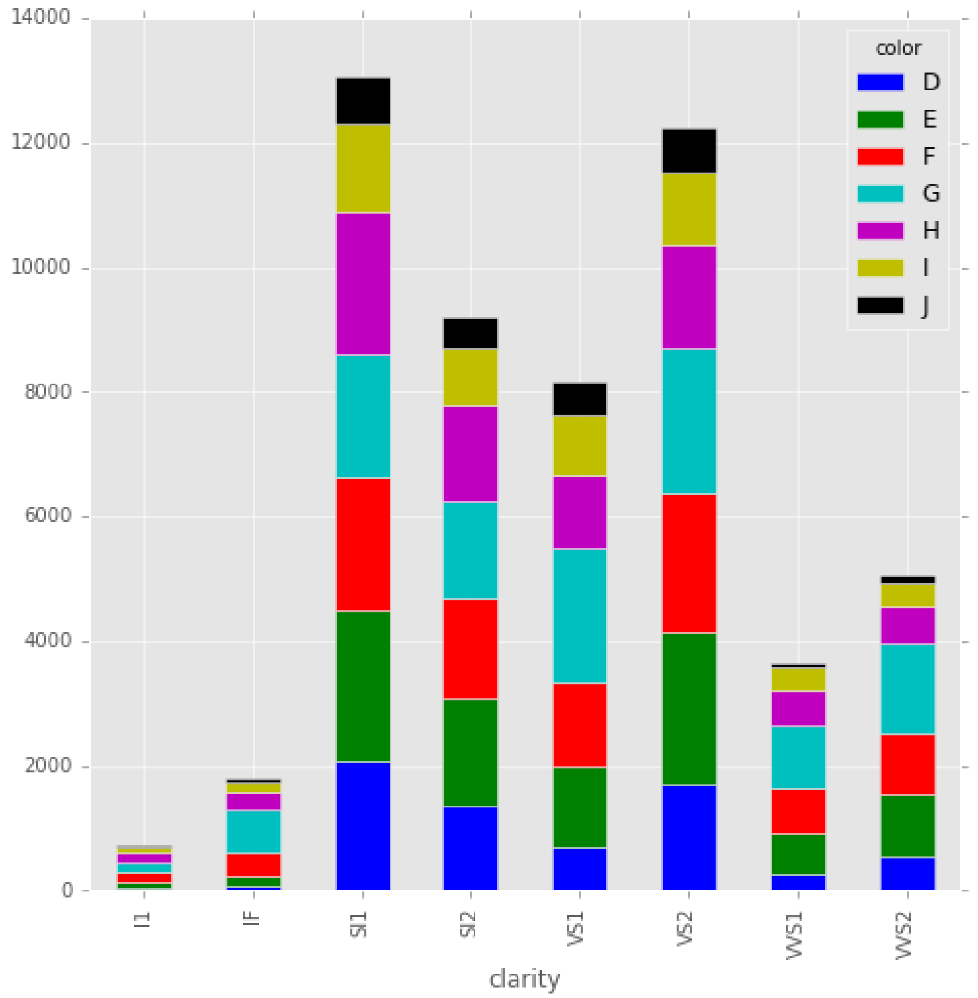
- You can use a two dimensional table to create a stacked barplot.
- Stacked barplots show the distribution of a second categorical variable within each bar:

```
carat_table = pd.crosstab(index=diamonds["clarity"],
                          columns=diamonds["color"])

carat_table
```

color	D	E	F	G	H	I	J
clarity							
I1	42	102	143	150	162	92	50
IF	73	158	385	681	299	143	51
SI1	2083	2426	2131	1976	2275	1424	750
SI2	1370	1713	1609	1548	1563	912	479
VS1	705	1281	1364	2148	1169	962	542
VS2	1697	2470	2201	2347	1643	1169	731
VVS1	252	656	734	999	585	355	74
VVS2	553	991	975	1443	608	365	131

Barplots

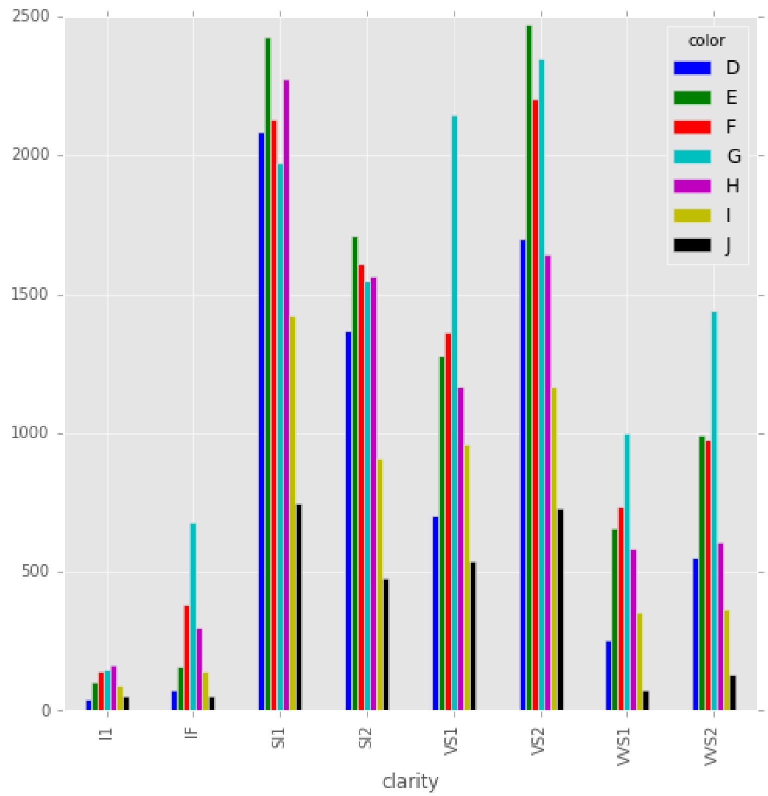
```
carat_table.plot(kind="bar",
                 figsize=(8,8),
                 stacked=True)
```



Grouped Barplots

- A grouped barplot is an alternative to a stacked barplot that gives each stacked section its own bar.
- To make a grouped barplot, do not include the stacked argument (or set `stacked=False`):

```
carat_table.plot(kind="bar",
                 figsize=(8,8),
                 stacked=False)
```

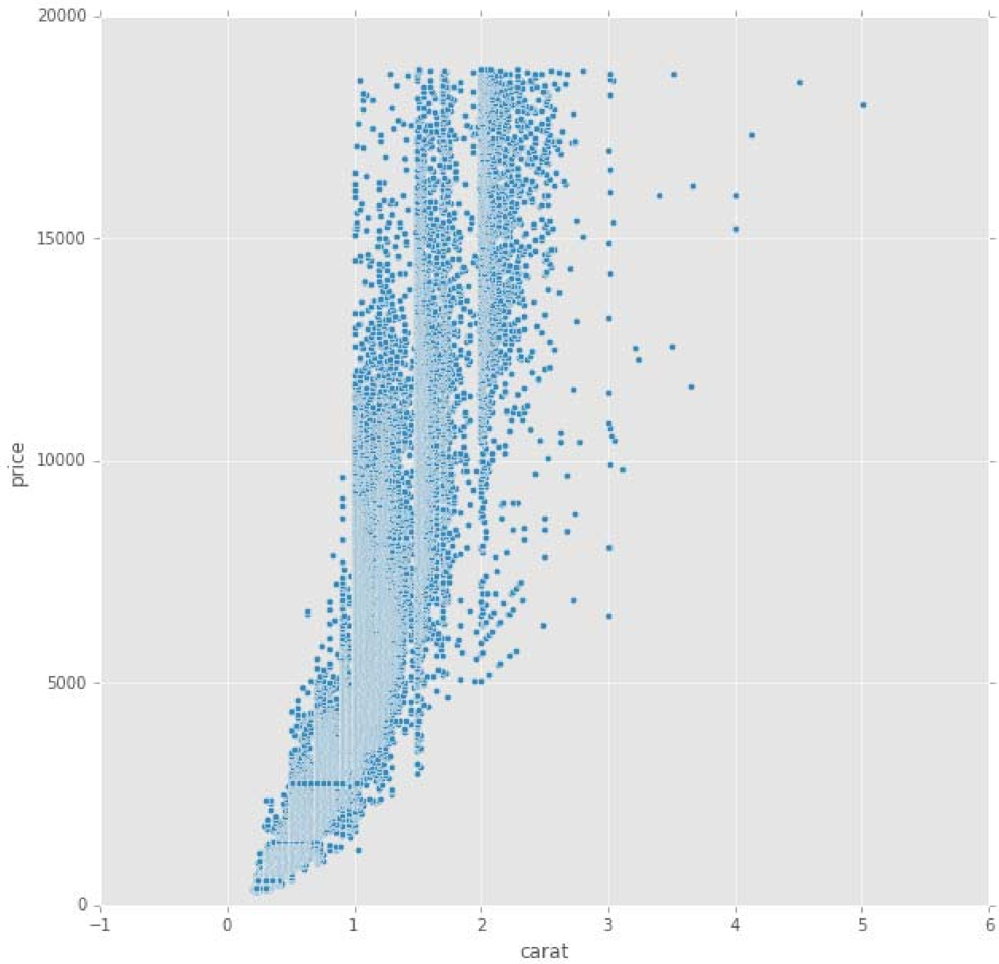


Scatterplots

- Scatterplots are bivariate (two variable) plots that take two numeric variables and plot data points on the *x/y* plane.
- To create a single scatterplot, use `df.plot(kind="scatter")`.

```
diamonds.plot(kind="scatter",     # Create a scatterplot
              x="carat",          # Put carat on the x axis
              y="price",          # Put price on the y axis
              figsize=(10,10),
              ylim=(0,20000))
```

Scatterplots



- Although the scatterplot above has many overlapping points, it still gives us some insight into the relationship between diamond carat weight and price: bigger diamonds are generally more expensive.

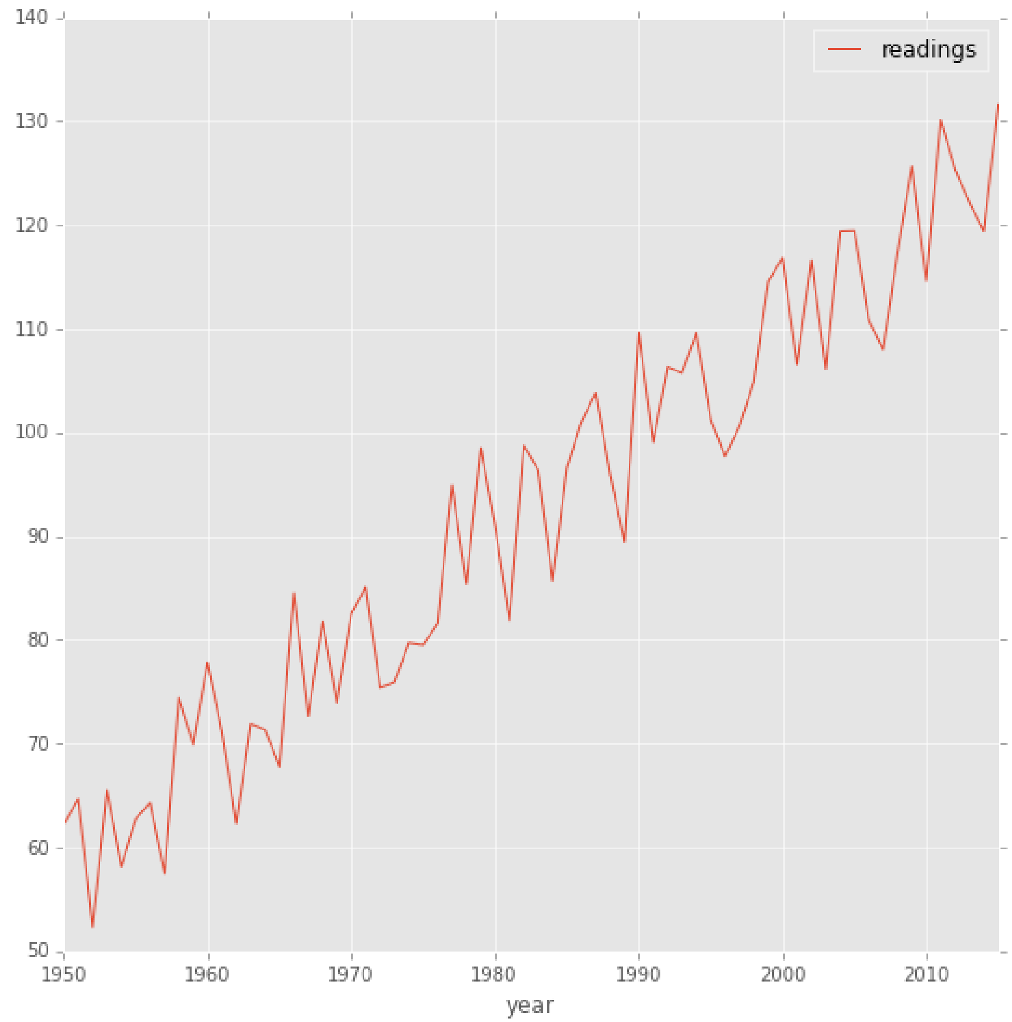
Line Plots

- Line plots are charts used to show the change in a numeric variable based on some other ordered variable.
- Line plots are often used to plot time series data to show the evolution of a variable over time.
- Line plots are the default plot type when using `df.plot()` so you don't have to specify the kind argument when making a line plot in pandas.
- Let's create some fake time series data and plot it with a line plot.

```
# Create some data
years = [y for y in range(1950,2016)]

readings = [(y+np.random.uniform(0,20)-1900) for y in years]

time_df = pd.DataFrame({"year":years,
                        "readings":readings})

# Plot the data
time_df.plot(x="year",
             y="readings",
             figsize=(9,9))
```

Line Plots

```
years = [y for y in range(1950,2016)]
readings = [(y+np.random.uniform(0,20)-1900) for y in years]
time_df = pd.DataFrame({"year":years,
                        "readings":readings})
time_df.plot(x="year",
             y="readings",
             figsize=(9,9))
```



Saving Plots

- If you want to save plots for later use, you can export the plot figure (plot information) to a file.
- First get the plot figure with `plot.get_figure()` and then save it to a file with `figure.savefig("filename").`
- You can save plots to a variety of common image file formats, such as `png, jpeg` and `pdf`.

```
my_plot = time_df.plot(x="year",     # Create the plot and save to a variable
             y="readings",
             figsize=(9,9))

my_fig = my_plot.get_figure()            # Get the figure

my_fig.savefig("line_plot_example.png")  # Save to file
```

Plotting with `mathplotlib`

- For the next pasrt, we will explore a dataset containing the taxi trips made in New York City 2013.
    - Maintained by the **New York City Taxi** and Limousine Commission,
- This 50GB dataset contains the date, time, geographical coordinates of pickup and drop-off locations, fare, and other information for 170 million taxi trips

The subset of the dataset

- To keep the analysis times reasonable we will analyze a subset of this dataset containing 0.5% of all trips (about 850,000 rides).
- Compressed, this subset data represents a little less than 100MB.
- You will find the data subset we will be using in this part in the minibook folder.
- The original 50GB dataset contained 24 zipped CSV files (a data and a fare file for every month).
- A Python script was used to go through all of these files and extracting one row out of 200 rows.
- Then, the rows were ordered by chronological order (using the pickup time).
- All rows with inconsistent coordinates has bee removed.
- The coordinates of a rectangle surrounding Manhattan has been defined (to restrict to this area only the rows where both pickup and drop-off locations were within this rectangle has been kept)

Using data science libraries

Let's import again a few packages we will need

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

- It is common practice to import matplotlib's interface named `pyplot` with `plt`

Using the data subsets

- Download the data subset, available here https://github.com/ipython-books/minibook-2nd-data and extract it in the current directory.
- Move to the `chapter2` subdirectory in the minibook's directory

- `nyc_data.csv` contains information about the rides
- `nyc_fare.csv` contains information about the fares

Reading files

```
data_filename ='data/nyc_data.csv'
fare_filename ='data/nyc_fare.csv'
```

- Pandas provides a powerful `read_csv()` function that can read virtually any CSV file
- Here we just need to specify which columns contain the dates so that pandas can parse them correctly

```
data = pd.read_csv(data_filename, parse_dates=['pickup_datetime', 'dropoff_datetime'])
fare=pd.read_csv(fare_filename, parse_dates=['pickup_datetime'])
data.head(3)
```

	medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag	pickup_datetim
0	76942C3205E17D7E7FE5A9F709D16434	25BA06A87905667AA1FE5990E33F0E2E	VTS	1	NaN	2013-01-01 00:00:00
1	517C6B330DBB3F055D007B07512628B3	2C19FBEE1A6E05612EFE4C958C14BC7F	VTS	1	NaN	2013-01-01 00:05:00
2	ED15611F168E41B33619C83D900FE266	754AEBD7C80DA17BA1D81D89FB6F4D1D	CMT	1	N	2013-01-01 00:05:52

Displaying the dataset

```
data.describe()
```

	rate_code	passenger_count	trip_time_in_secs	trip_distance	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_la
count	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.000000	846945.00
mean	1.026123	1.710272	812.523879	9.958211	-73.975155	40.750490	-73.974197	40.750967
std	0.223480	1.375266	16098.305145	6525.204888	0.035142	0.027224	0.033453	0.030766
min	0.000000	0.000000	-10.000000	0.000000	-74.098305	40.009911	-74.099998	40.009911
25%	1.000000	1.000000	361.000000	1.050000	-73.992371	40.736031	-73.991570	40.735207
50%	1.000000	1.000000	600.000000	1.800000	-73.982094	40.752975	-73.980614	40.753597
75%	1.000000	2.000000	960.000000	3.200000	-73.968048	40.767460	-73.965157	40.768227
max	6.000000	6.000000	4294796.000000	6005123.000000	-73.028473	40.996132	-73.027061	40.998592

- The `describe()` method shows basic statistics of all columns

Making plots with matplotlib

- Here we will display the pickup and dropoff locations of all trips

```
data.columns
```

```
Out[8]:
        Index(['medallion', 'hack_license', 'vendor_id',
        'rate_code', 'store_and_fwd_flag', 'pickup_datetime',
        'dropoff_datetime', 'passenger_count',
        'trip_time_in_secs', 'trip_distance',
        'pickup_longitude', 'pickup_latitude',
        'dropoff_longitude', 'dropoff_latitude'],
        dtype='object')
```

- Four columns mention `latitude` and `longitude`
- We load these columns

```
p_lng = data.pickup_longitude
p_lat = data.pickup_latitude
d_lng = data.dropoff_longitude
d_lat = data.dropoff_latitude
```

Selecting columns

- With pandas, every column of a DataFrame can be obtained with the `mydataframe.columnname` syntax.
- An alternative syntax is `mydataframe['columnname']`.
- We created four variables with the coordinates of the *pickup* and *dropoff* locations.
- These variables are all `Series` objects

`p_lng` generates:

```
0 -73.955925
1 -74.005501
2 -73.969955
3 -73.991432
4 -73.966225
5 -73.955238
6 -73.985580
…….
```

Coordinate into pixels

- Before we can make a plot, we need to get the coordinates of points in pixels instead of geographical coordinates.
- We use the following function (Mercator projection).

```
def lat_lng_to_pixels(lat, lng):
    lat_rad = lat*np.pi / 180.0
    lat_rad = np.log(np.tan((lat_rad + np.pi / 2.0) / 2.0))
    x = 100 * (lng + 180.0) / 360.0
    y= 100 * (lat_rad - np.pi) / (2.0 * np.pi)
    return(x,y)
```

- NumPy implements many math functions.
- They work on scalar numbers and also on pandas objects such as series
- The following function call returns two new series `px` and `py`.

```
px, py = lat_lng_to_pixels(p_lat, p_lng)
```

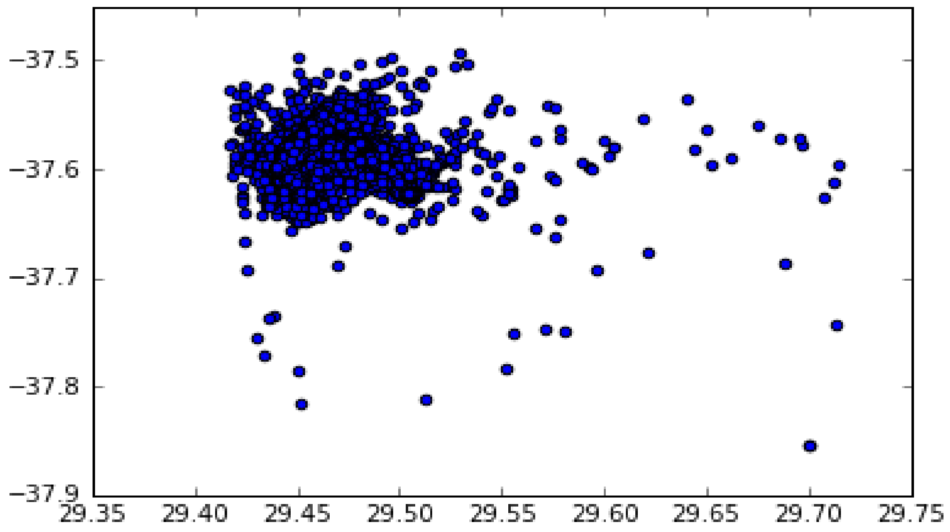
The px Series

```
In [7]: px    In [7]:

        0           29.456688
        1           29.442916
        2           29.452790
        3           29.446824
        4           29.453826
        5           29.456878
        6           29.448450
        7           29.444608
        8           29.446617
        9           29.442624
        10          29.452091
        11          29.442427
        ......
```

Using scatter plot

- The matplotlib `scater()` function takes two arrays with `x` and `y` coordinates as inputs.
- A **scatter plot** is a 2D figure showing points with various positions, sizes, color and marker shapes
- The following command displays all pickup locations

```
In [7]:   plt.scatter(px, py)
```



A customized scatter plot

- In the previous scatter plot the markers are too big
- Second, there are too many points,
- We could make them a bit transparent to have a better of their distribution.
- Third we may want to zoom around Manhattan
- Forth we could make this figure bigger
- And finally, we don't need the axes here.
- Fortunately, matplotlib is customizable and all aspects of the plot can be changes , as shown below.

```
plt.figure(figsize=(16,12))
plt.scatter(px,py, s=.3, alpha=.03)
plt.axis('equal')
plt.xlim(29.40,29,55)
plt.ylim(-37.63, -37.54)
plt.axis('off')
```

- The `scatter()` function accepts many keyword arguments
- With a small alpha value the points become nearly transparent
- We use an equal aspect ratio with `axis('equal')`
- We zoom in by specifying the limits of *x* and *y* axes.

A better scatter plot

Plotting with seaborn

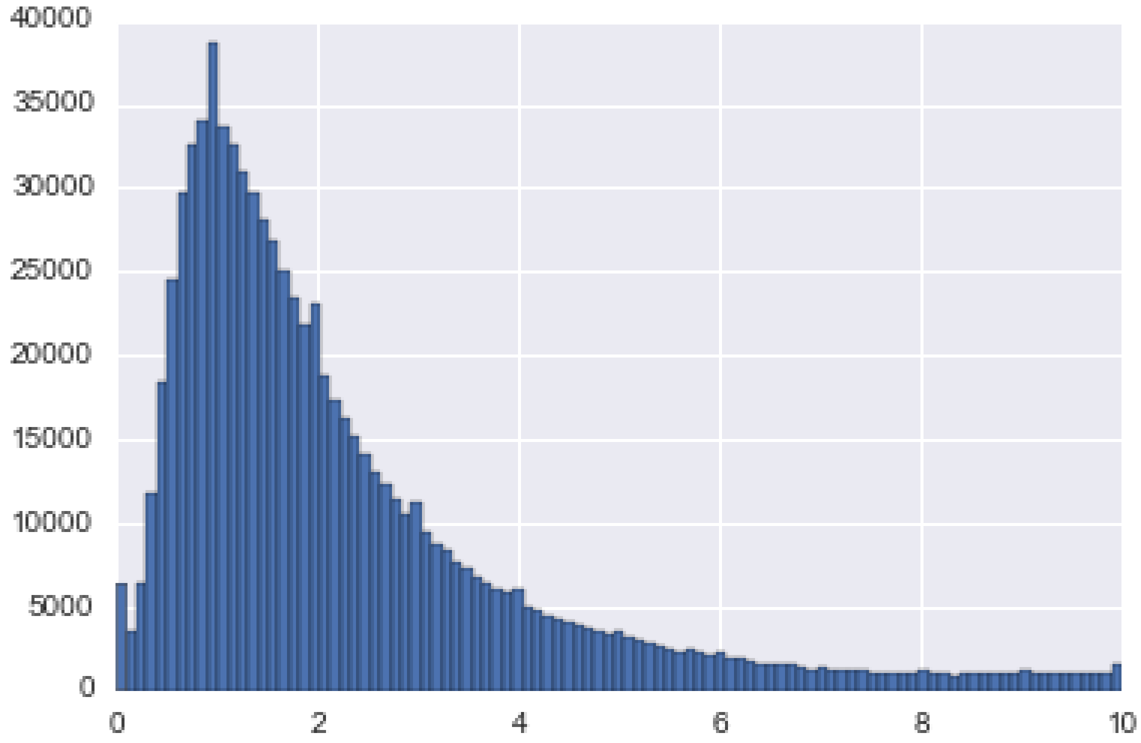
- Matplotlib is the main plotting package in Python.
- Although powerful and flexible, it sometimes require a significant amount of manual tuning to generate quality publication-ready figures.
- Seaborn offers simple user interface for high-quality plotting.
- First we need to install seaborn

```
!conda install seaborn –q –y
import seaborn as sns
```

- seaborn improves the aesthetics and color palettes of matplotlib figures.
- It also provides several easy-to-use statistical plotting functions
- We'll display a histogram of the trip distances.
  - Pandas provides a few simple plotting methods for `DataFrame` and `Series`
  - We can specify the histogram bins with the `bins` keyword argument
  - We use Numpy's `linspace()` function to generate 100 linearly spaced bins between 0 and 10

Plotting a histogram

```
In [8]:  data.trip_distance.hist(bins=np.linspace(0.,10.,100))
```

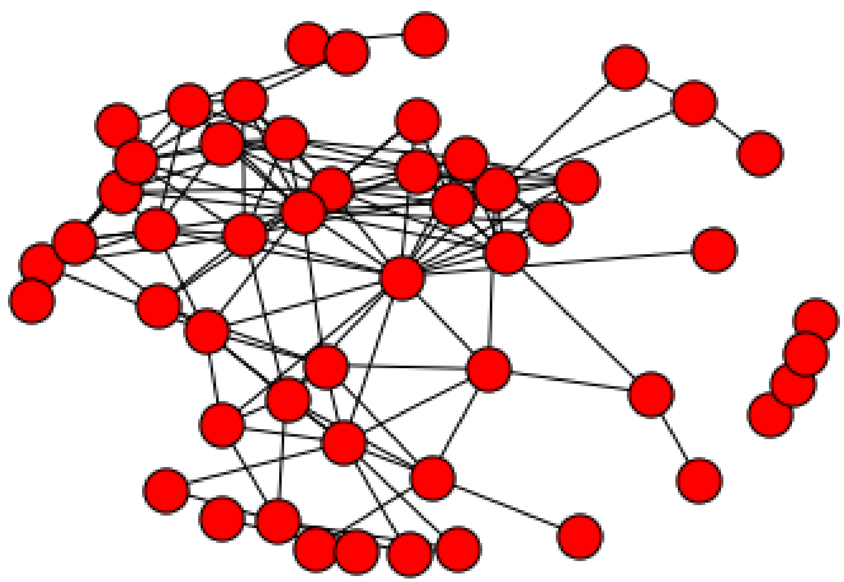


Visualizing Graphs

- *NetworkX* is a Python software package for the creation, manipulation of the structure, dynamics, and functions of complex networks.
- NetworkX provides also basic functionality for visualizing graphs.
- We will illustrate this plotting functionality with graph from Facebook, `3980.edges` provided in minibook.

Portion of Facebook graph

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import networkx as nx

graph = nx.read_edgelist('3980.edges')
len(graph.nodes())
len(graph.edges())
nx.draw(graph)
plt.show()
```



Wrap Up

- Pandas plotting functions let you visualize and explore data quickly.
- Pandas plotting functions don't offer all the features of dedicated plotting package like `matplotlib`, `seaborn` or `networkx` but they are often enough to get the job done.