Plotting with Pandas

Nathaniel Merrill, Shane McQuarrie, Amelia Henriksen, Marissa Graham

February 23, 2017

Objective: *Pandas has many built-in plotting methods that wrap around matplotlib. Since Pandas provides tools for organizing and correlating large sets of data, it is important to be able to visualize these relationships. This lab consists of two parts: learning to plot with Pandas, and using those techniques to visualize data in informative ways.*

Solutions Format: *IPython / Jupyter Notebook*

Plotting Data Frames

Recall that in Pandas, a *DataFrame* is an ordered collection of *Series*. A series is similar to a dictionary, with values assigned to various labels, or indices. Each series becomes a column in the data frame, with each row corresponding to an index. When several series are combined into a single data frame, it becomes very easy to compare and visualize data.

Data frames have several methods for easy plotting. Most are simple wrappers around matplotlib commands, but some produce styles of plots that we have not previously seen.

For these examples, we will use the data found in the file `crime_data.txt`.

```
>>> import pandas as pd
>>> crime = pd.read_csv("crime_data.txt", header=1, index_col=0)
```

Line Plots

Using matplotlib, we can plot the data in a single pandas `Series` against the data frame index (the years, in this case). With a few extra lines we modify the x-axis label and limits and create a legend.

```
>>> plt.plot(crime["Population"], label="Population")
>>> plt.xlabel("Year")
>>> plt.xlim(min(crime.index), max(crime.index))
>>> plt.legend(loc="best")
>>> plt.show()
```

Equivalently, we can produce the exact same plot with a single line using the DataFrame method `plot()`. Specify the y values as a keyword argument. The x values default to the index of the Series. See Figure 1.

```
>>> crime.plot(y="Population")
>>> plt.show()
```

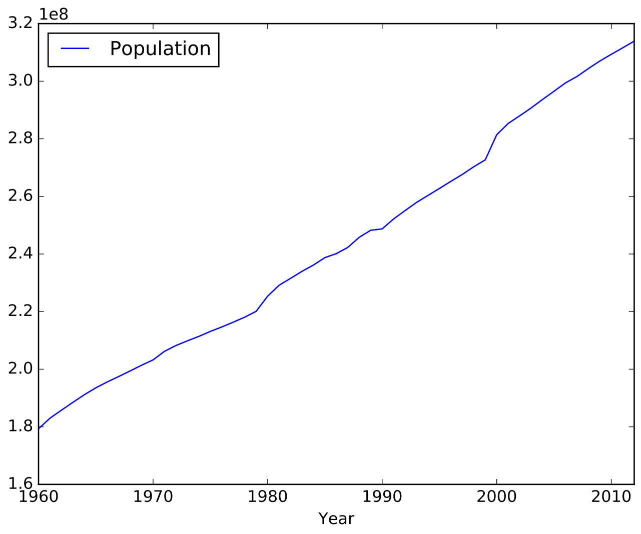


Figure 1: Population by Year.

We can also plot two series against each other (ignoring the index). In matplotlib:

```
>>> plt.plot(crime["Population"], crime["Burglary"])
>>> plt.show()
```

With `DataFrame.plot()`, specify the x values as a keyword argument:

```
>>> crime.plot(x="Population", y="Burglary")
>>> plt.show()
```

Both procedures produce the same line plot, but the data frame method automatically sets the limits and labels of each axis and includes a legend. See Figure 13.

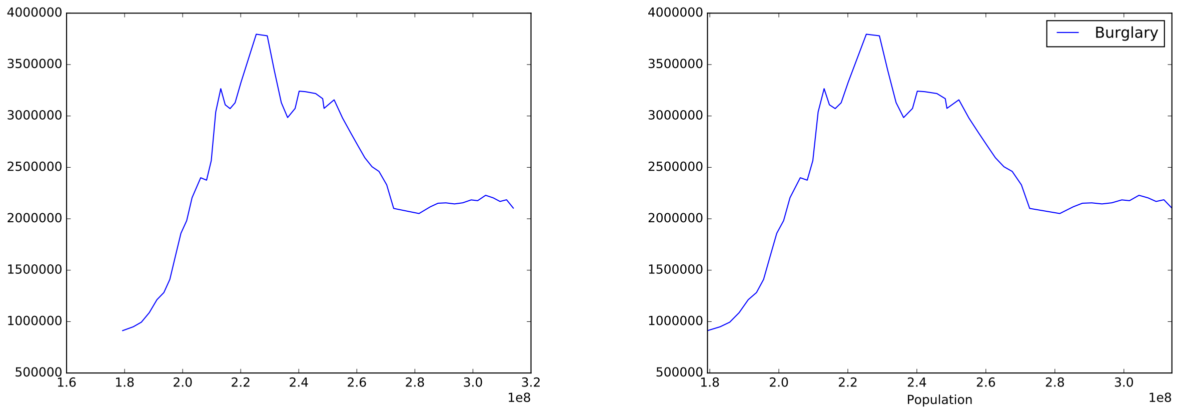


Figure 2: On the left, the result of `plt.plot()`. On the right, the result of `DataFrame.plot()`

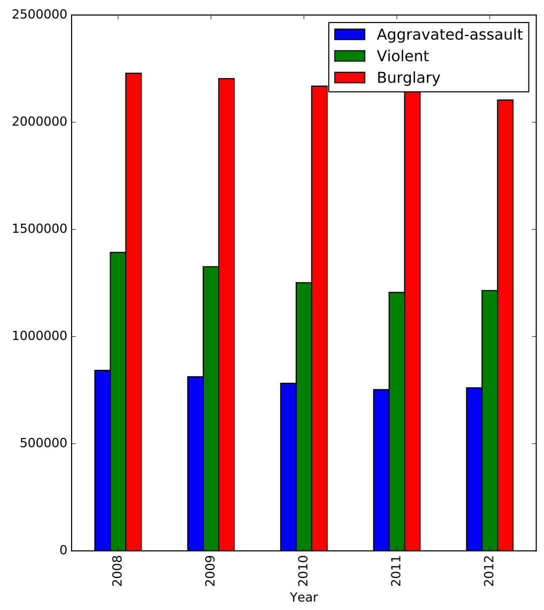
Standard matplotlib keyword arguments can be passed in as usual to `DataFrame.plot()`. This includes the ability to produce subplots quickly, modify the linestyle, and so on.

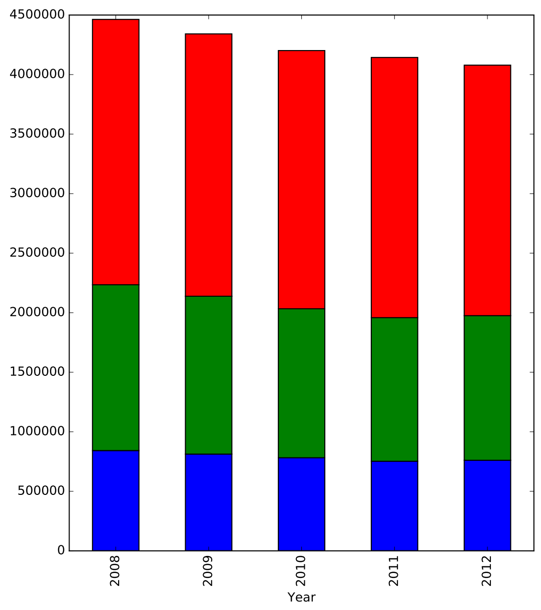
```
>>> crime.plot(subplots=True, layout=(4,3), linewidth=3, style="--", legend=False)
>>> plt.show()
```

Bar Plots

By default, the data frame's `plot()` function creates a line plot. We can create other types of plots easily by specifying the keyword `kind`. For example, bar plots are particularly useful for comparing several categories of data over time, or whenever there is a sense of progression in the index. Bar plots tend to give a little more detail than line plots, though line plots are the better choice when there are many data points. As an example, we compare three different types of crime over the last 5 years contained in the crime data set.

```
# Each call to plot() makes a separate figure automatically.
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="bar")
>>> crime.iloc[-5:][["Aggravated-assault", "Violent", "Burglary"]].plot(kind="bar", stacked=
    ↪ True, legend=False)
>>> plt.show()
```





Problem #1

The `pydataset` module[1] contains numerous data sets stored as Pandas data frames.

```
from pydataset import data
# "data" is a Pandas data frame with IDs and descriptions.
# Call data() to see the entire list.
# To load a particular data set, enter its ID as an argument to data().
titanic_data = data("Titanic")
```

[1]Run `pip install pydataset` if needed

```
# To see the information about a data set, give data() the dataset_id with show_doc=True.
data("Titanic", show_doc=True)
```

Examine the data sets with the following `pydataset` IDs:

1. `nottem`: Average air temperatures at Nottingham Castle in Fahrenheit for 20 years.
2. `VADeaths`: Death rates per 1000 in Virginia in 1940.
3. `Arbuthnot`: Ratios of male to female births in London from 1629-171.

Use line and bar plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

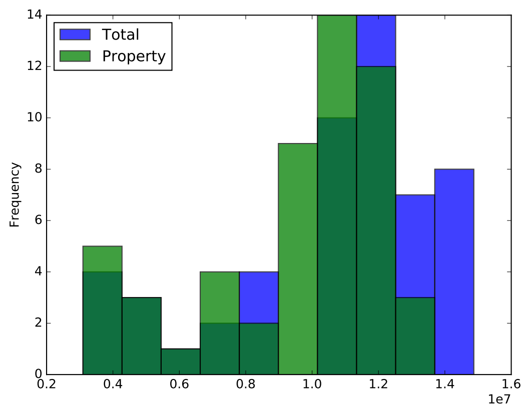
Histograms

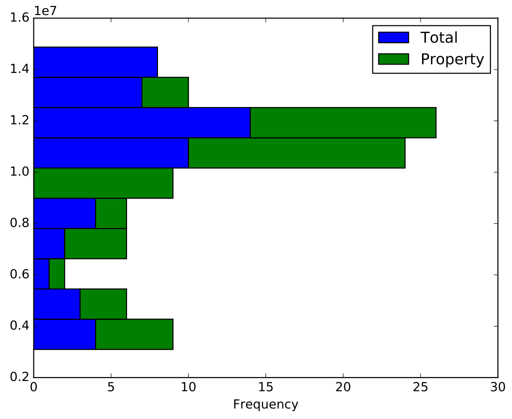
Line and bar plots work well when there is a logical progression in the index, such as time. However, when frequency of occurence is more important than the location of the data, histograms and box plots can be more informative. Use `plot(kind="hist")` to produce a histogram. Standard histogram options, such as the number of bins, are also accepted as keyword arguments. The `alpha` keyword argument makes each bin slightly transparent.

```
>>> crime[["Total", "Property"]].plot(kind="hist", alpha=.75)
>>> plt.show()
```

Alternatively, the bins can be stacked on top of each other by setting the `stacked` keyword argument to `True`. We can also mke the histogram horizontal by seting the keyword `orientation` to “horizontal”.

```
>>> crime[["Total", "Property"]].plot(kind="hist", stacked=True, orientation="horizontal")
>>> plt.show()
```

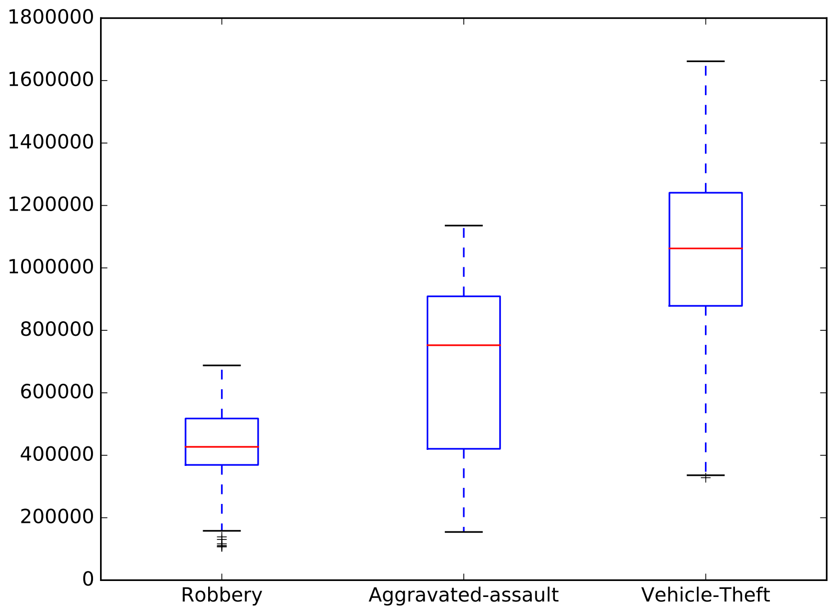




Box Plots

Sometimes it is helpful to visualize a distribution of values using the box-and-whisker plot which is commonly taught in elementary school, which displays the median, first and third quartiles, and outliers. Like the previous examples, select the columns to examine and plot them with `plot()`. To switch the orientation, use `vert=False`.

```
crime[["Robbery", "Aggravated-assault", "Vehicle-Theft"]].plot(kind="box")
plt.show()
```



Problem #2

Examine the data sets with the following `pydataset` IDs:

1. `trees`: Girth, height and volume for black cherry trees.
2. `road`: Road accident deaths in the United States.
3. `birthdeathrates`: Birth and death rates by country.

Use histograms and box plots to visualize each of these data sets. Decide which type of plot is more appropriate for each data set, and which columns to plot together. Write a short description of each data set based on the docstrings of the data and your visualizations.

Scatter Plots

Scatter plots are one of the most commonly used type of plot and have a simple implementation in pandas. Unlike other plotting commands, `scatter` needs both an `x` and a `y` column as arguments.

```
>>> mammal_speed = data("Mammals")
>>> mammal_speed.plot(kind="scatter", x="weight", y="speed"))
>>> plt.show()
```

The package includes many more scatter plot features which can be used to make the plots easier to understand. For example, we can change the size of the point based on another column. Consider the pydataset `HairEyeColor`, which contains the hair and eye color of various individuals. A scatter plot of hair color vs eye color is relatively useless unless we can see the frequencies with which each combination occurs. Including the keyword argument `s` allows us to control the size of each point. This can be set to a fixed value or the value in another column. In the example below, the size of each point is set to the frequency with which each observation occurs.

```
>>> hec = data("HairEyeColor")
>>> X = np.unique(hec["Hair"], return_inverse=True)
>>> Y = np.unique(hec["Eye"], return_inverse=True
>>> hec["Hair"] = X[1]
>>> hec["Eye"] = Y[1]
>>> hec.plot(kind="scatter", x="Hair", y="Eye", s=hec["Freq"]*20)
>>> plt.xticks([0,1,2,3], X[0])
>>> plt.yticks([0,1,2,3], Y[0])
>>> plt.show()
```

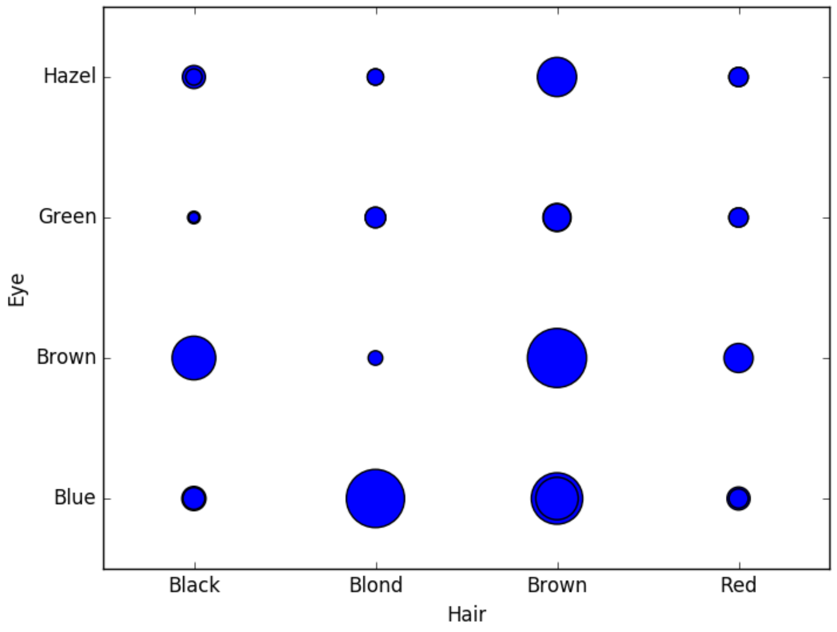


Figure 3: Frequency of Hair-Eye Color Combinations

Hexbins

While scatter plots are a great visualization tool, they can be uninformative for large datasets. It is nearly impossible to tell what is going on in a large scatter plot, and the visualization is therefore of little value. Hexbin plots solve this problem by plotting point density in hexagonal bins. With hexbins, the structure of the data is easy to see despite the noise that is still present. Following is an example using pydataset's `sat.act` plotting the SAT Quantitative score vs ACT score of students

```
>>> satact = data("sat.act")
>>> satact.plot(kind="scatter", x="ACT", y="SATQ")
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
>>> plt.show()
```

Note the scatter plot in Figure 4. While we can see the structure of the data, it is not easy to differentiate the densities of the areas with many data points. Compare this now to the hexbin plot in Figure 5. The two plots are clearly similar, but with the hexbin plot we have the added dimension of density.

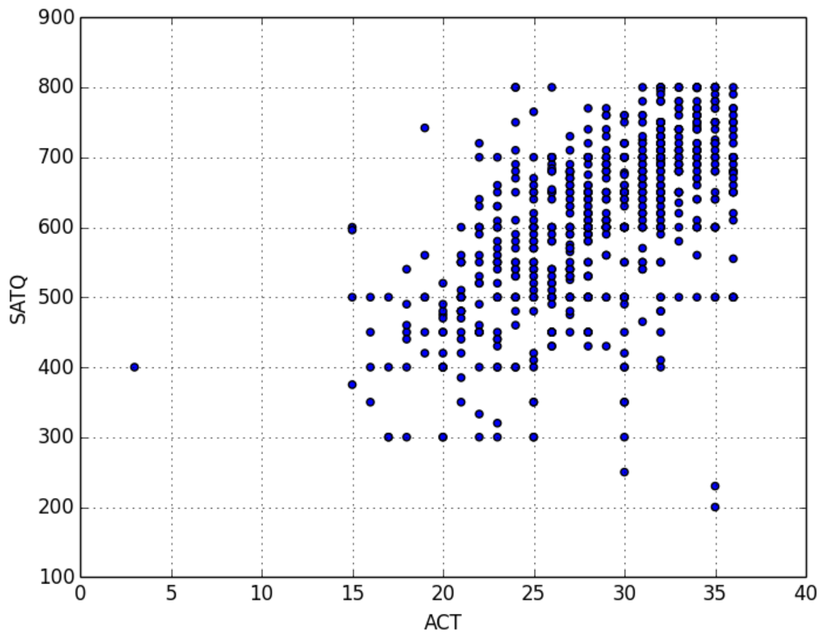


Figure 4: Scatter Plot

A key factor in creating an informative hexbin is choosing an appropriate `gridsize` parameter. This determines how large or small the bins will be. A large gridsize will give many small bins and a small gridsize gives a few large bins. Figure 6 shows the effect of changing the gridsize from 20 to 10.

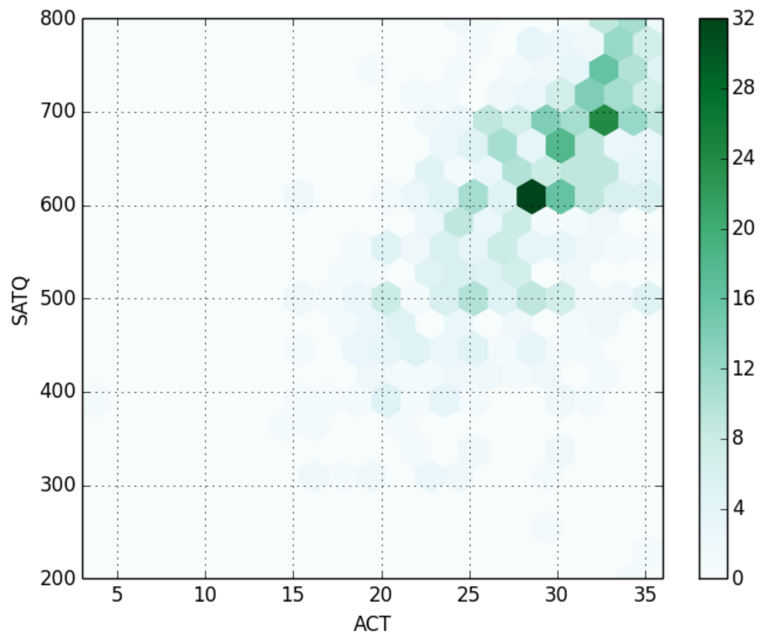


Figure 5: Hexbin Plot With Gridsize 20

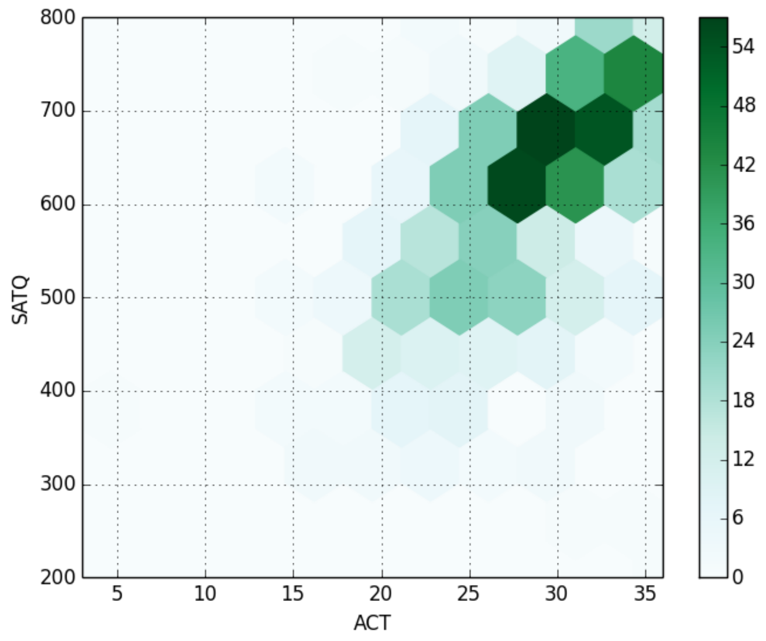


Figure 6: Hexbin Plot With Gridsize 10

Lag Plot

We are frequently interested in whether or not data which we have collected is random. Lag plots are used to investigate the randomness of a dataset. If the data is in fact random, then the lag plot will exhibit no structure, while nonrandom data will exhibit some kind of structure. Unfortunately, this does not give us an idea of what exactly that structure may be, but it is a quick and effective way to investigate the randomness of a given dataset.

```
>>> from pandas.tools.plotting import lag_plot
>>> randomdata = pd.Series(np.random.rand(1000))
>>> lag_plot(randomdata)
>>> plt.show()

>>> structureddata = pd.Series(np.sin(-np.pi, np.pi, num=1000))
>>> lag_plot(structureddata)
>>> plt.show()
```

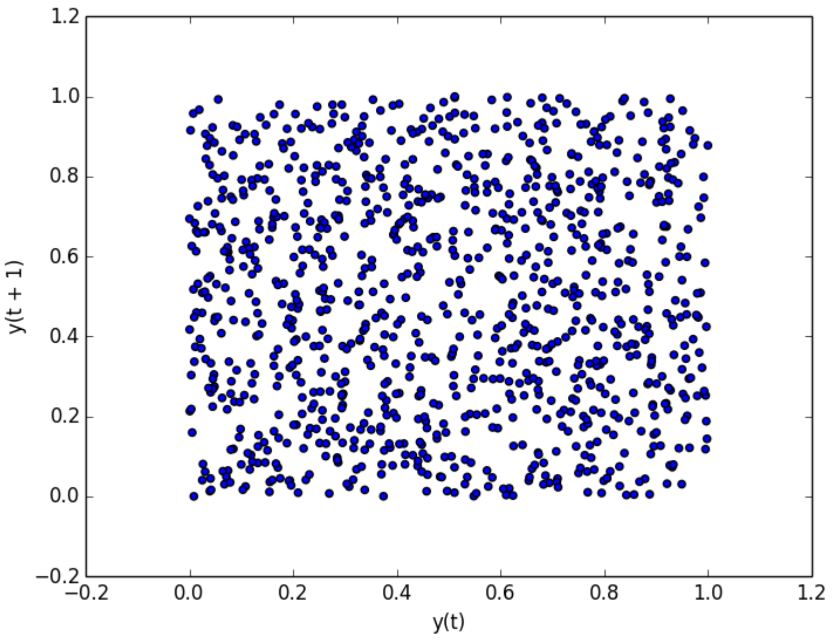


Figure 7: Lag Plot of Random Data

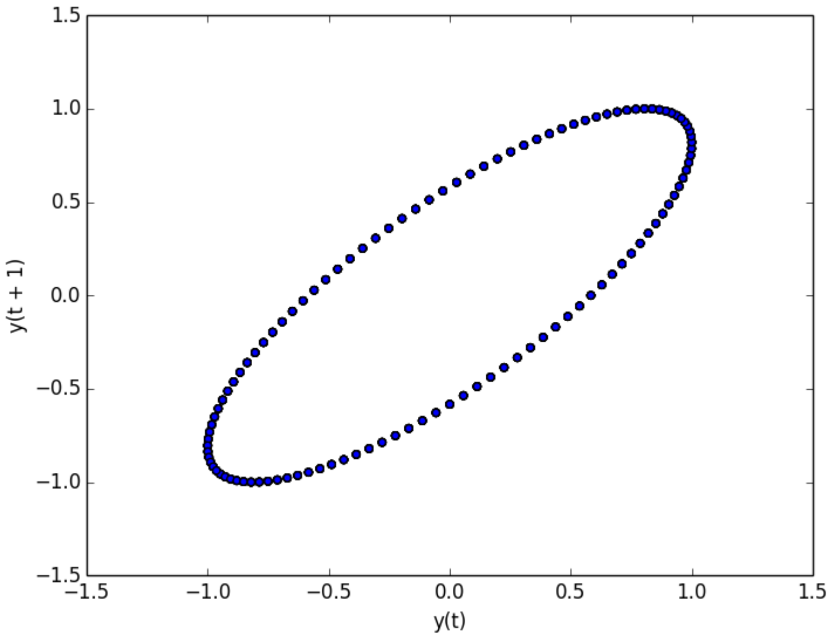


Figure 8: Lag Plot of Sine Wave

Problem #3

Choose a dataset provided in the `pydataset` and produce scatter and hexbin plots demonstrating some characteristic of the data. A list of datasets in the `pydataset` module can be produced using the `data()` command.

For more types of plots available in Pandas and further examples, see http://pandas.pydata.org/pandas-docs/stable/visualization.html.

Data Visualization

Visualization is much more than a set of pretty pictures scattered throughout a paper for the sole purpose of providing candy-colored filler. When properly implemented, data visualization is a powerful tool for analysis and communication. In this section we discuss the process of making deliberate, effective, and efficient design decisions.

Catching all of the Details

Consider the figure entitled "Plot 1". What does it depict? We can tell from a simple glance that it is a scatter plot of positively correlated data of some kind, with `temp`–likely temperature–on the x axis and `cons` on the y axis. However, the picture is not really communicating anything about the dataset. We have not specified the units for the x or the y axis, we have no idea what `cons` is, and we don't even know where the data came from in the first place.

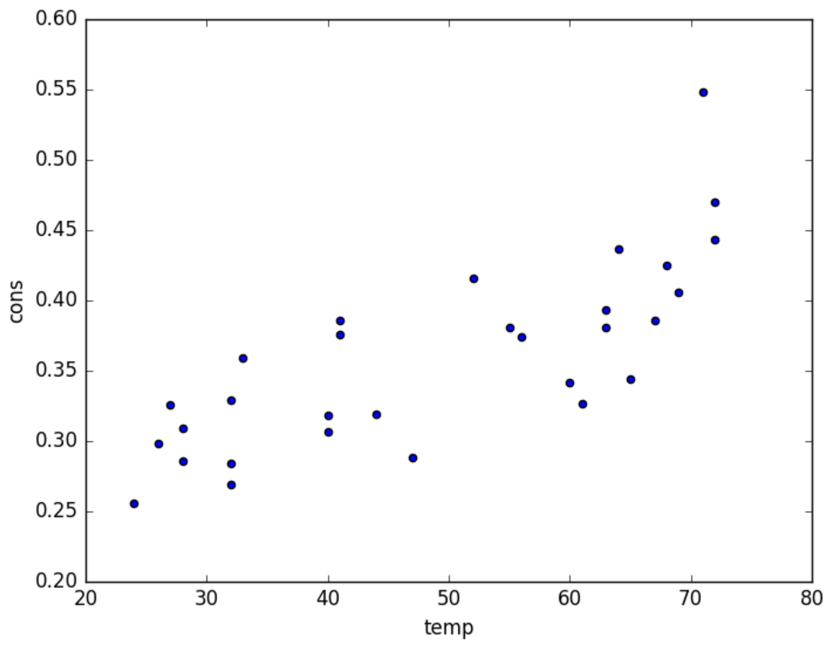


Figure 9: Plot 1

Labels, Legends, and Titles

In a homework or lab setting, we sometimes (mistakenly) think that it is acceptable to leave off appropriate labels, legends, titles, and sourcing. In a published report or presentation, this kind of carelessness is confusing at best and (when the source is not included) plagiaristic at worst. Here, we offer a brief overview of the pandas code that will allow us to add these vital elements.

Consider again "Plot 1". This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

```
>>> icecream = data("Icecream")
>>> icecream.plot(kind="scatter", x="temp", y="cons")
```

We have at this point reproduced the rather substandard plot in Figure 9. Using `data('` `↪ Icecream', show_doc=True)` we find the following information:

1. The dataset details ice cream consumption via four-weekly observations from March 1951 to July 1953 in the United States.

2. `cons` corresponds to "consumption of ice cream per head" and is measured in pints.

3. `temp` corresponds to temperature, degrees Fahrenheit.

4. The listed source is: "Hildreth, C. and J. Lu (1960) _Demand relations with autocorrelated disturbances_, Technical Bulletin No 2765, Michigan State University."

We add these important details using the following code. As we have seen in previous examples, Pandas automatically generates legends when appropriate. However, although Pandas also automatically labels the x and y axes, our data frame column titles may be insufficient. Appropriate titles for the x and y axes must also list appropriate units. For example, the y axis should specify that the consumption is *per head* with unit *pints*, in place of the ambigious label `cons`.

```
>>> icecream = data("Icecream")
# Set title via the title keyword argument
>>> icecream.plot(kind="scatter", x="temp", y="cons", title="Ice Cream Consumption in the U.
    ↪ S., 1951-1953",)
# Override pandas automatic labelling using xlabel and ylabel
>>> plt.xlabel("Temp (Farenheit)")
>>> plt.ylabel("Consumption per head (pints)")
>>> plt.show()
```

Unfortunately, there is no explicit function call that allows us to add our source information. To arbitrarily add the necessary text to the figure, we may use either `plt.annotate` or `plt.text`.

```
>>> plt.text(20, .1, "Source: Hildreth, C. and J. Lu (1960) _Demand relations with
    ↪ autocorrelated disturbances_\nTechnical Bulletin No 2765, Michigan State University."
    ↪ , fontsize=7)
```

Both of these methods are imperfect, however, and can normally be just as easily replaced by a caption attached to the figure in your presentation or document setting. We again reiterate how important it is that you source any data you use. Failing to do so is plagiarism.

Finally, we have a clear and demonstrative graphic:

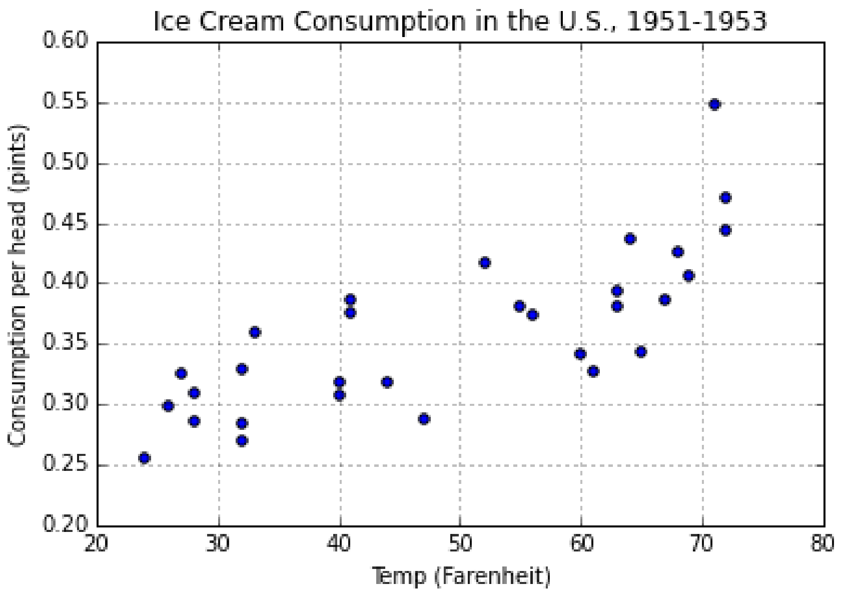


Figure 10: Source: Hildreth, C. and J. Lu (1960) _Demand relations with autocorrelated disturbances_, Technical Bulletin No 2765, Michigan State University.

Problem #4

Return to the plots you generated in problem one (datasets `nottem`, `VADeaths`, and `Arbuthnot`). Reproduce and modify these plots to include:

- A clear title, with relevant information for the period or region the data was collected in.
- Axes that specify units.
- A legend (for comparison data).
- The source. You may include the source information in your plot or print it after the plot at your discretion.

Note that in this and all subsequent problems, points will be taken off if any of these items are partially or fully missing from your graphs.

Choosing the Right Plot

Now that we know how to add the appropriate details for remaining plots, we return to the fundamental question of data visualization–which plot is the right plot for your data? In previous sections, we have already discussed the various strengths and weaknesses of available pandas plotting techniques. At this point, we know how to visualize data using bar charts and histograms or scatter plots and hexbins. However, perfectly plot-ready data sets–organized by a simple continuum or a convenient discrete set–are few and far between. In the real world, it is rare to find a dataframe that is already ready to become a meaningful visual.

As such, deciding how to organize and group the data within your dataframe is one of the most important parts of “choosing the right plot”.

Groupby

Many datasets are simply composed of tables of individuals, with a list of classifiers associated with each one. Data like this is difficult to sensibly plot in its raw format.

For example, consider the `msleep` dataset (found in `pydataset`). Each row consists of a single type of mammal and its corresponding identifiers, including genus and order, as well as sleep measurements such as total amount of sleep (in hours) and REM sleep, in hours. When we try to plot this data using plt.plot, the individual data points do not demonstrate overall trends.

```
>>> msleep = data("msleep")
>>> msleep.plot(y="sleep_total", title="Mammalian Sleep Data", legend=False)
>>> plt.xlabel("Animal Index")
>>> plt.ylabel("Sleep in Hours")
>>> plt.show()
```

The above code results in Figure 11.

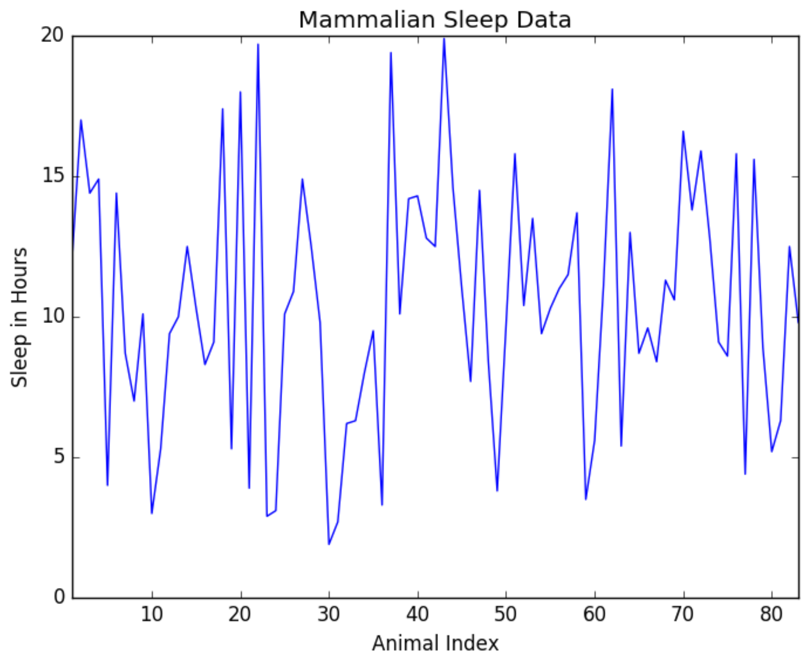


Figure 11: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

This seemingly random set of connected data points is not particularly revealing.

We thus decide to approach our dataset using pandas' powerful `groupby` method. Let's say that, for the `msleep` dataset, we want to visualize the differences in sleep data between herbivores, omnivores, insectivores and carnivores. Then, we simply call the groupby method on the `vore` column to obtain a groupby object organized diet classification.

```
>>> msleep = data("msleep")
>>> vore = msleep.groupby("vore")
```

You can also group the data by multiple columns, for example, both the `vore` and `order` classifications. To group and view this data, simply use:

```
>>> vorder = msleep.groupby(["vore", "order"])

# View groups within vorder
>>> vorder.describe
```

Pandas `groupby` objects are not lists of new dataframes associated with groupings. They are better thought of as a dictionary or generator-like object which can be *used* to produce the necessary groups. However, if you want to work within a dataframe for a specific group, you may use the `get_group()` method as follows:

```
# Get carnivore group
>>> Carni = vore.get_group("carni")
# Get herbivore group
>>> Herbi = vore.get_group("herbi")
```

The groupby object includes many useful methods that can help us make visual comparisons between groups. The mean() method, for example, returns a new dataframe consisting of the mean values attached to each group. For example, using this method on our vore object returns a nicely organized dataframe of the average sleep data for each mammalian diet pattern. We can similarly create a dataframe of the standard deviations for each group.

At this point, we have a nicely organized dataset that can easily be turned into a bar chart. Here, we use the dataframe.loc method to access three specific columns in the bar chart (sleep_total, sleep_rem, and sleep_cycle).

```
>>> means = vore.mean()
>>> errors = vore.std()

>>> means.loc[:,["sleep_total", "sleep_rem", "sleep_cycle"]].plot(kind="bar", yerr=errors,
    ↪ title="Mean Mammallian Sleep Data")
>>> plt.xlabel("Mammal diet classification (vore)")
>>> plt.ylabel("Hours")
>>> plt.show()
```

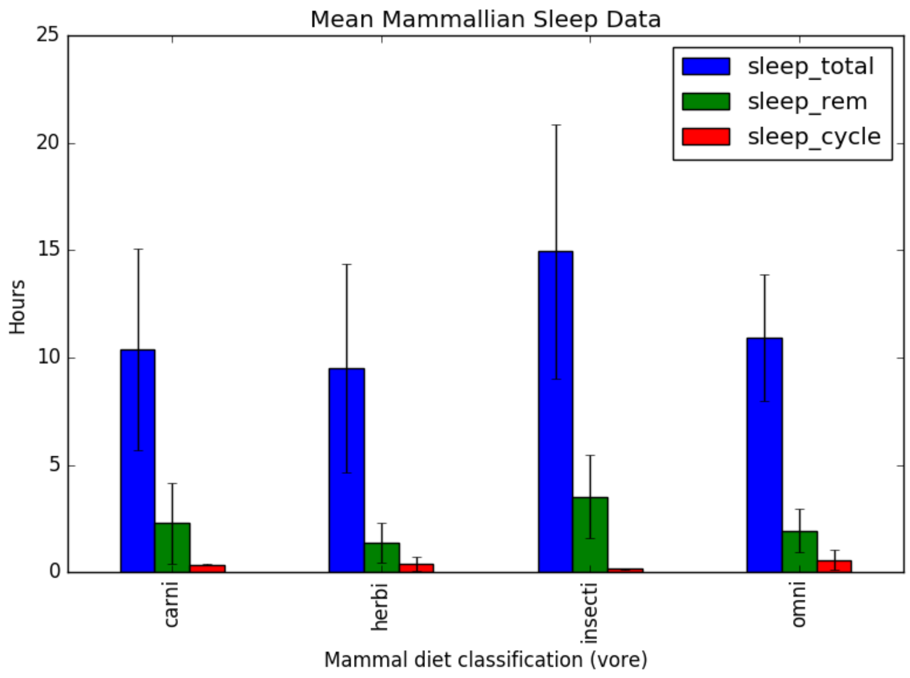


Figure 12: Source: Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007. Updates from V. M. Savage and G. B. West, with additional variables supplemented by Wikipedia.

The pandas groupby object has many different methods. We have discussed a few of these here; for more information please see:

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

Problem #5

Examine the `diamonds` dataset found in the `pydataset` module. This dataset contains the identifiers and attributes of 53,940 individual round cut diamonds. Using the `groupby` method, create three different visuals highlighting and comparing different aspects of the data. This can be in the form of a single plot or comparative subplots.

Print a few sentences for each plot explaining what type of graph you used, why, and what we learn about the dataset from your plot. Don't forget that points will be taken off for each plot without a title, clear labels, and sourcing.

The following is an example of the kind of plot we are looking for. This example may not be used as one of your three plots.

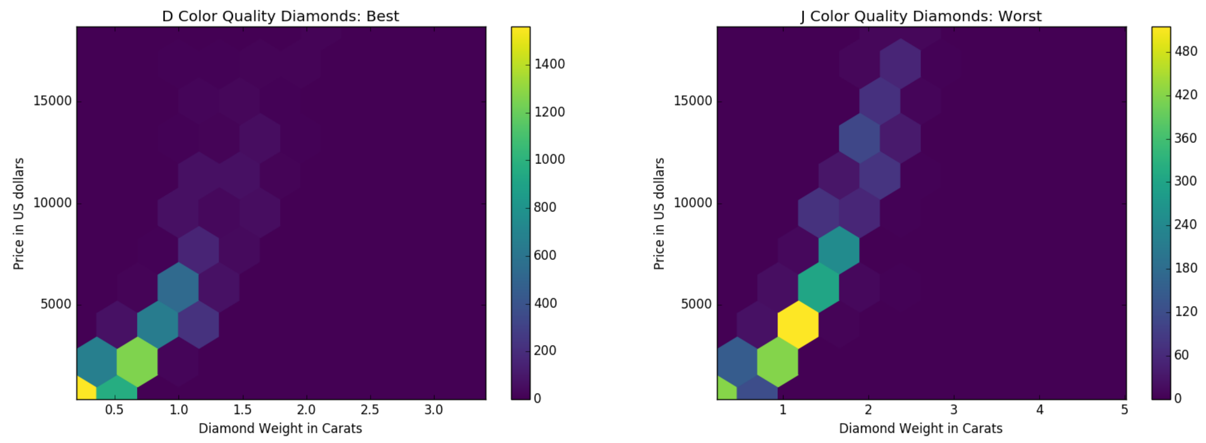


Figure 13: Source: Adopted from R Documentation

The code for the above figure is included here:

```
>>> Diamonds = data("diamonds")
>>> DiaColor = Diamonds.groupby("color")
>>> Ddiamond = DiaColor.get_group("D")
>>> Jdiamond = DiaColor.get_group("J")

>>> Ddiamond.plot(kind="hexbin", x="carat", y="price", gridsize=10, title="D Color Quality
    ↪ Diamonds: Best", cmap="viridis")
>>> plt.ylabel("Price in US Dollars")
>>> plt.xlabel("Diamond Weight in Carats")
>>> plt.tight_layout()
>>> plt.show()

>>> Jdiamond.plot(kind="hexbin", x="carat", y="price", gridsize=10, title="J Color Quality
    ↪ Diamonds: Worst", cmap="viridis")
>>> plt.ylabel("Price in US Dollars")
>>> plt.xlabel("Diamond Weight in Carats")
>>> plt.tight_layout()
>>> plt.show()
```

We now analyze the plots.

The above plots were created using `groupby` on the diamond colors and then using a hexbin comparing carats to price for the highest and lowest quality diamonds, respectively. This hexbin is particularly revealing for each set of thousands of diamonds because it meaningfully displays concentration of datapoints. Matplotlib's new `viridis` colorplot, with a dark background, reveals bins that would have been invisible with a white background. By comparing these plots, we note that the greatest number of J quality diamonds in the dataset are about 1.25 carats and \$4000 dollars in price, whereas the highest concentration of D quality diamonds are smaller and therefore cheaper. We may attribute this to D quality diamonds being rarer, but the colorbar on the side reveals that D diamond numbers are, in fact, far higher than those of the J color. Instead it is simply more likely that D quality diamonds of larger sizes are rarer than those of smaller sizes. Both hexbins reveal a linearity between diamond weight and diamond price, with D diamonds showing more variability and J diamonds displaying a strict linearity.