10 Pandas 2: Plotting

Lab Objective: *Clear, insightful visualizations are a crucial part of data analysis. To facilitate quick data visualization, pandas includes several tools that wrap around matplotlib. These tools make it easy to compare different parts of a data set, explore the data as a whole, and spot patterns and correlations the data.*

Note

This lab will be done using Colab Notebooks. These notebooks are similar to Jupyter Notebooks but run remotely on Google's servers. Open a Google Colab notebook by going to your Google Drive account and creating a new Colaboratory file. If making a Colaboratory file is not an option, download the application Colaboratory onto your Google Drive. Once opening a new Colab Notebook, upload the file `pandas2.ipynb`. To make the data files accessible, run the following at the top of the lab:

```
>>> from google.colab import files

>>> uploaded = files.upload()
```

This will prompt you upload files for this notebook. For this lab, upload `crime_data.csv` and `college.csv`.

Once the lab is complete, delete BOTH lines of code used for uploading files (the import statement and the upload statement) and download as a `.py` file to your git repository. Push the newly made `pandas2.py` file.

Overview of Plotting Tools

The main tool for visualization in pandas is the `plot()` method for `Series` and `DataFrames`. The method has a keyword argument `kind` that specifies the type of plot to draw. The valid options for `kind` are detailed below.

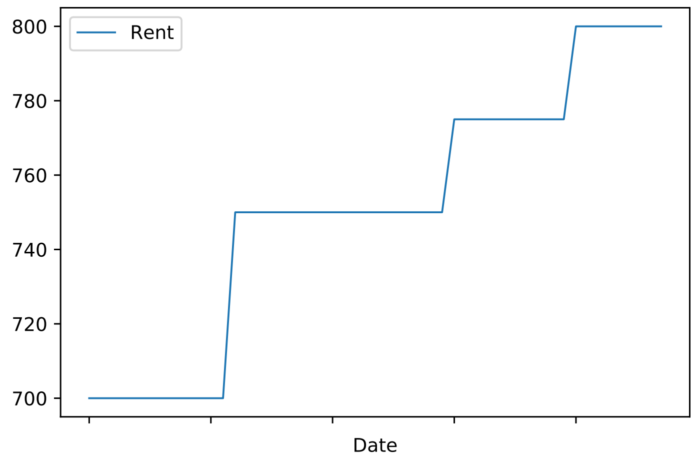
Plot Type	`plot()` ID	Uses and Advantages
Line plot	`"line"`	Show trends ordered in data; easy to compare multiple data sets
Scatter plot	`"scatter"`	Compare exactly two data sets, independent of ordering
Bar plot	`"bar"`, `"barh"`	Compare categorical or sequential data
Histogram	`"hist"`	Show frequencies of one set of values, independent of ordering
Box plot	`"box"`	Display min, median, max, and quartiles; compare data distributions
Hexbin plot	`"hexbin"`	2D histogram; reveal density of cluttered scatter plots

Table 10.1: Types of plots in pandas. The plot ID is the value of the keyword argument `kind`. That is, `df.plot(kind="scatter")` creates a scatter plot. The default `kind` is `"line"`.

The `plot()` method calls `plt.plot()`, `plt.hist()`, `plt.scatter()`, and other matplotlib plotting function, but it also assigns axis labels, tick marks, legends, and a few other things based on the index and the data. Most calls to `plot()` specify the `kind` of plot and which `Series` to use as the `x` and `y` axes. By default, the `index` of the `Series` or `DataFrame` is used for the `x` axis.

```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt

>>> budget = pd.read_csv("new_budget.csv", index_col="Year")
>>> budget.plot(y="Rent")  # Plot rent against the index (date).
```



In this case, the call to the `plot()` method is essentially equivalent to the following code.

```
>>> plt.plot(budget.index, budget['Rent'], label='Rent')
>>> plt.xlabel(budget.index.name)
>>> plt.xlim(min(budget.index), max(budget.index))
>>> plt.legend(loc='best')
```

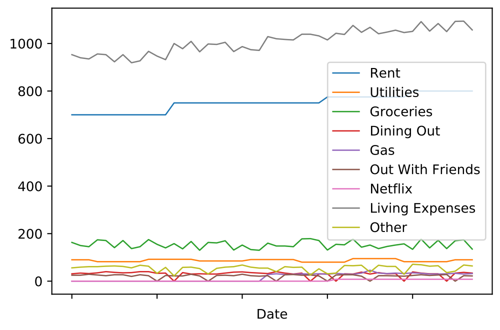
The `plot()` method also takes in many keyword arguments for matplotlib plotting and annotation functions. For example, setting `legend=False` disables the legend, providing a value for `title` sets the figure title, `grid=True` turns a grid on, and so on. For more customizations, see `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html`.

Visualizing an Entire Data Set

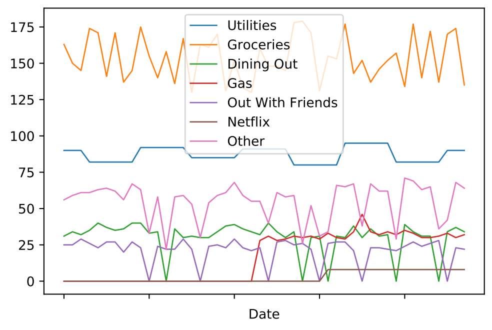
A good way to start analyzing an unfamiliar data set is to visualize as much of the data as possible to determine which parts are most important or interesting. For example, since the columns in a `DataFrame` share the same index, the columns can all be graphed together using the index as the x-axis. By default, the `plot()` method attempts to plot **every** `Series` (column) in a `DataFrame`. This is especially useful with sequential data, like the budget data set.

```
>>> # Plot all columns together against the index.
>>> budget.plot(linewidth=1)

>>> # Plot all columns together except for 'Living Expenses' and 'Rent'.
>>> budget.drop(["Living Expenses", "Rent"], axis=1).plot(linewidth=1)
```



(a) All columns of the budget data set on the same figure, using the index as the x-axis.



(b) All columns of the budget data set except `"Living Expenses"` and `"Rent"`.

Figure 10.1

While plotting every `Series` at once can give an overview of all the data, the resulting plot is often difficult for the reader to understand. For example, the budget data set has 9 columns, so the resulting figure, Figure 10.1a, is fairly cluttered.

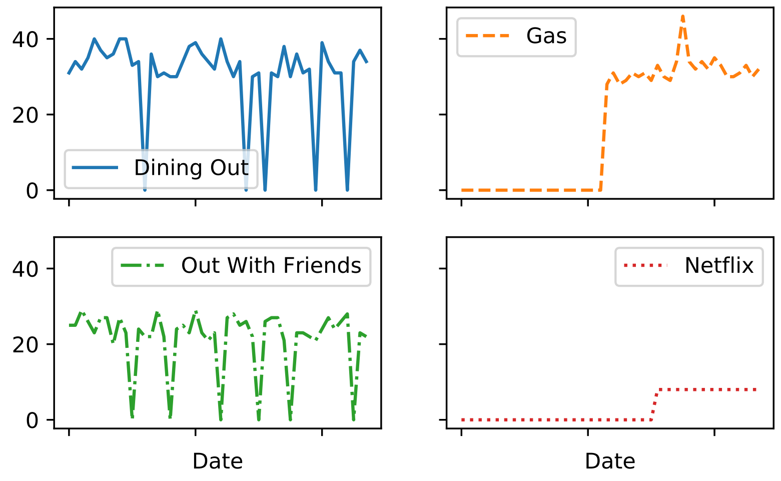
One way to declutter a visualization is to examine less data. Notice that `'Living Expenses'` has values much bigger than the other columns. Dropping this column, as well as `'Rent'`, gives a better overview of the data, shown in Figure 10.1b.

Achtung!

Often plotting all data at once is unwise because columns have **different units of measure**. Be careful not to plot parts of a data set together if those parts do not have the same units or are otherwise incomparable.

Another way to declutter a plot is to use subplots. To quickly plot several columns in separate subplots, use `subplots=True` and specify a shape tuple as the `layout` for the plots. Subplots automatically share the same x-axis. Set `sharey=True` to force them to share the same y-axis as well.

```
>>> budget.plot(y=['Dining Out','Gas','Out With Friends', 'Netflix'],
...                     subplots=True, layout=(2,2), sharey=True,
...                     style=['-','--','-.',':'])
```



As mentioned previously, the `plot()` method can be used to plot different kinds of plots. One possible kind of plot is a histogram. Since plots made by the `plot()` method share an x-axis by default, histograms turn out poorly whenever there are columns with very different data ranges or when more than one column is plotted at once.

```
>>> # Plot three histograms together.
>>> budget.plot(kind='hist',y=['Gas','Dining Out','Out With Friends'],
...                  alpha=.7,bins=10)

>>> # Plot three histograms, stacking one on top of the other.
>>> budget.plot(kind='hist',y=['Gas','Dining Out','Out With Friends'],
...                     bins=10,stacked=True)
```

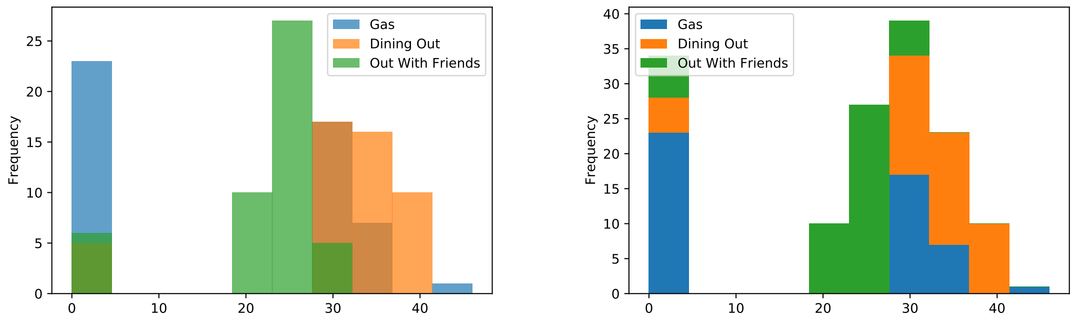


Figure 10.2: Two examples of histograms that are difficult to understand because multiple columns are plotted.

Thus, histograms are good for examining the distribution of a **single** column in a data set. For histograms, use the `hist()` method of the `DataFrame` instead of the `plot()` method. Specify the number of bins with the `bins` parameter. Choose a number of bins that accurately represents the data; the wrong number of bins can create a misleading or uninformative visualization.

```
>>> budget[["Dining Out","Gas","Other","Out With Friends"]].hist(grid=False, ↩
    bins=10)
```

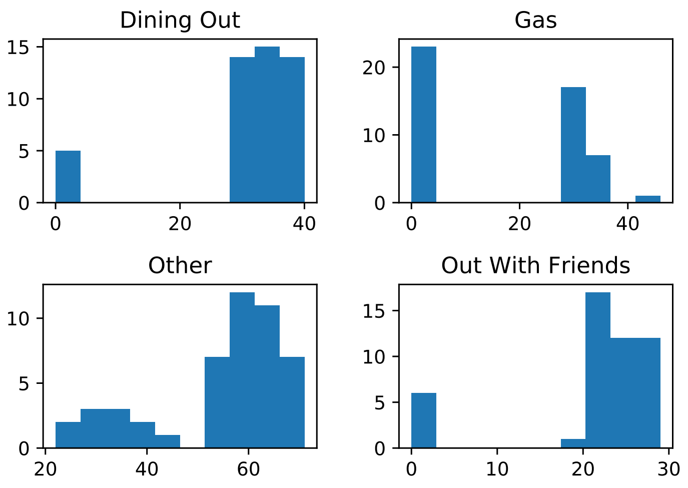


Figure 10.3: Histograms of "Dining Out", "Gas", "Other", and "Out With Friends".

Problem 1. Create 2 visualizations for the data in `crime_data.csv`. Make one of the visualizations a histogram. The visualizations should be well labeled and easy to understand. Include a short description of your plots as a caption.

Patterns and Correlations

After visualizing the entire data set initially, a good next step is to closely compare related parts of the data. This can be done with different types of visualizations. For example, Figure 10.1b suggests that the "Dining Out" and "Out With Friends" columns are roughly on the same scale. Since this data is sequential (indexed by time), start by plotting these two columns against the index. Next, create a scatter plot of one of the columns versus the other to investigate correlations that are independent of the index. Unlike other types of plots, using kind="scatter" requires both x and y columns as arguments.

```
>>> # Plot 'Dining Out' and 'Out With Friends' as lines against the index.
>>> budget.plot(y=["Dining Out", "Out With Friends"])

>>> # Make a scatter plot of 'Dining Out' against 'Out With Friends'
>>> budget.plot(kind="scatter", x="Dining Out", y="Out With Friends",
...                         alpha=.8)
```

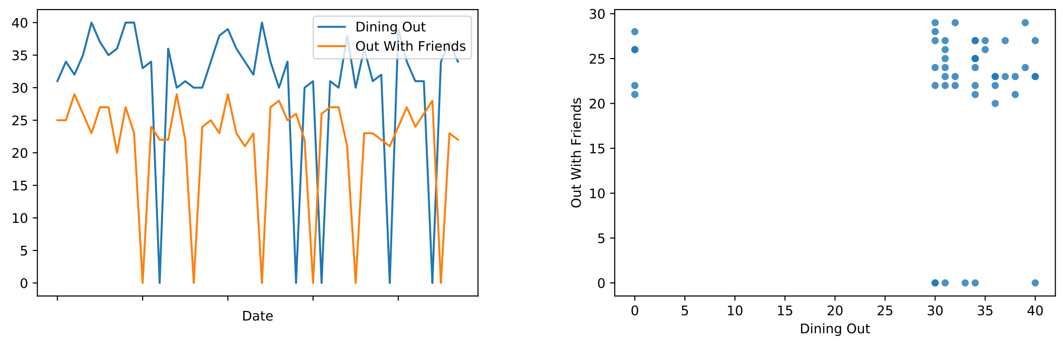


Figure 10.4: Correlations between "Dining Out" and "Out With Friends".

The first plot shows us that more money is spent on dining out than being out with friends overall. However, both categories stay in the same range for most of the data. This is confirmed in the scatter plot by the block in the upper right corner, indicating the common range spent on dining out and being out with friends.

Achtung!

When analyzing data, especially while searching for patterns and correlations, **always** ask yourself if the data makes sense and is trustworthy. What lurking variables could have influenced the data measurements as they were being gathered?

The crime data set from Problem 1 is somewhat suspect in this regard. The murder rate is likely accurate, since murder is conspicuous and highly reported, but what about the rape rate? Are the number of rapes increasing, or is the percentage of rapes being reported increasing? It's probably both! Be careful about drawing conclusions for sensitive or questionable data.

Another useful visualization used to understand correlations in a data set is a scatter matrix.

The function `pd.plotting.scatter_matrix()` produces a table of plots where each column is plotted against each other column in separate scatter plots. The plots on the diagonal, instead of plotting a column against itself, displays a histogram of that column. This provides a very quick method for an initial analysis of the correlation between different columns.

```
>>> pd.plotting.scatter_matrix(budget[['Living Expenses','Other']])
```

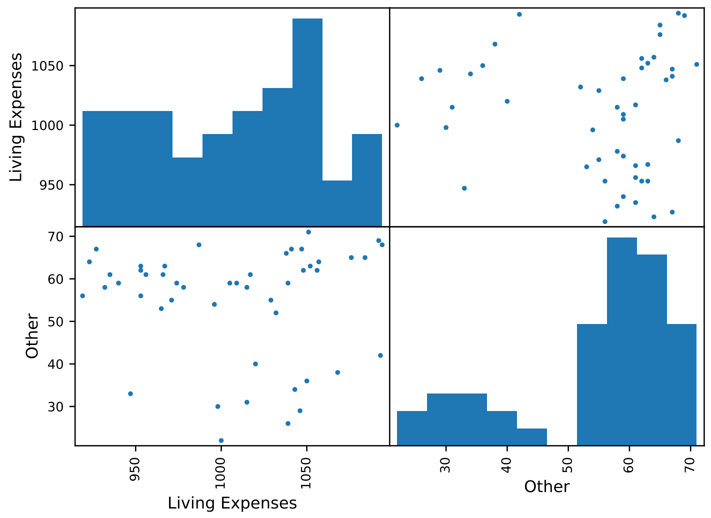


Figure 10.5: Scatter matrix comparing `"Living Expenses"` and `"Other"`.

Bar Graphs

Different types of graphs help to identify different patterns. Note that the data set `budget` gives monthly expenses. It may be beneficial to look at one specific month. Bar graphs are a good way to compare small portions of the data set.

As a general rule, horizontal bar charts (`kind="hbar"`) are better than the default vertical bar charts (`kind="bar"`) because most humans can detect horizontal differences more easily than vertical differences. If the labels are too long to fit on a normal figure, use `plt.tight_layout()` to adjust the plot boundaries to fit the labels in.

```
>>> # Plot all data for the last month in the budget
>>> budget.iloc[-1,:].plot(kind='barh')
>>> plt.tight_layout()

>>> # Plot all data for the last month without 'Rent' and 'Living Expenses'
>>> budget.drop(['Rent','Living Expenses'],axis=1).iloc[-1,:].plot(kind='barh')
>>> plt.tight_layout()
```

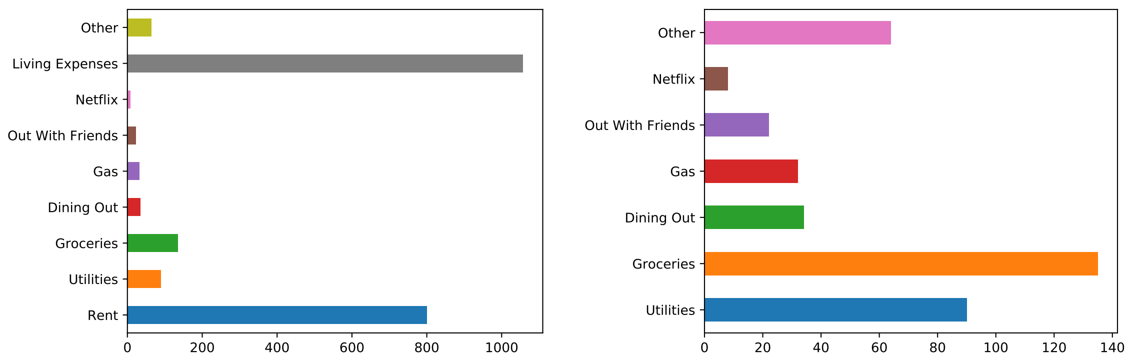


Figure 10.6: Bar graphs showing expenses paid in the last month of `budget`.

Problem 2. Using the crime data from the previous problem, identify if a trend exists between `Forcible Rape` and the following variables:

1. `Violent`
2. `Burglary`
3. `Aggravated Assault`

Make sure each graph is clearly labelled and readable. Include a caption explaining whether there is a visual trend between the variables.

Distributional Visualizations

While histograms are good at displaying the distributions for one column, a different visualization is needed to show the distribution of an entire set. A *box plot*, sometimes called a "cat-and-whisker" plot, shows the five number summary: the minimum, first quartile, median, third quartile, and maximum of the data. Box plots are useful for comparing the distributions of relatable data. However, box plots are a basic summary, meaning that they are susceptible to miss important information such as how many points were in each distribution.

```
>>> # Compare the distributions of four columns.
>>> budget.plot(kind="box", y=["Gas","Dining Out","Out With Friends","Other"])

>>> # Compare the distributions of all columns but 'Rent' and 'Living Expenses↩
    '.
>>> budget.drop(["Rent", "Living Expenses"], axis=1).plot(kind="box",
>>>                                                        vert=False)
```

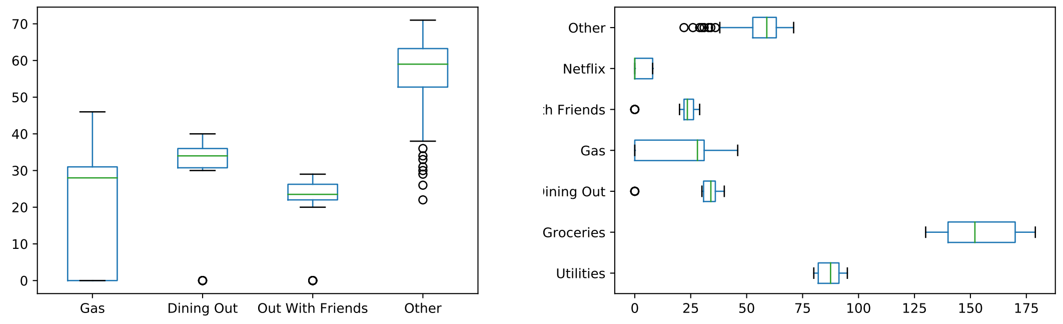


Figure 10.7: Vertical and horizontal box plots of `budget` dataset.

Hexbin Plots

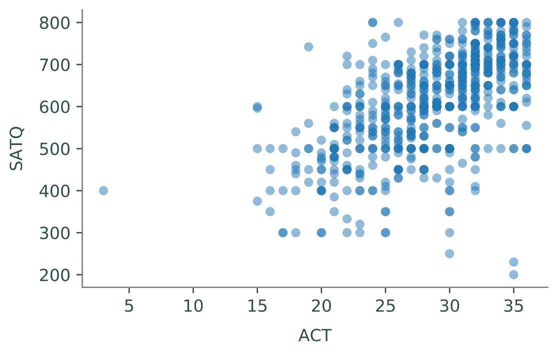
A scatter plot is essentially a plot of samples from the joint distribution of two columns. However, scatter plots can be uninformative for large data sets when the points in a scatter plot are closely clustered. *Hexbin plots* solve this problem by plotting point density in hexagonal bins—essentially creating a 2-dimensional histogram.

The file `sat_act.csv` contains 700 self reported scores on the SAT Verbal, SAT Quantitative and ACT, collected as part of the Synthetic Aperture Personality Assessment (SAPA) web based personality assessment project. The obvious question with this data set is "how correlated are ACT and SAT scores?" The scatter plot of ACT scores versus SAT Quantitative scores, Figure 10.8a, is highly cluttered, even though the points have some transparency. A hexbin plot of the same data, Figure 10.8b, reveals the **frequency** of points in binned regions.

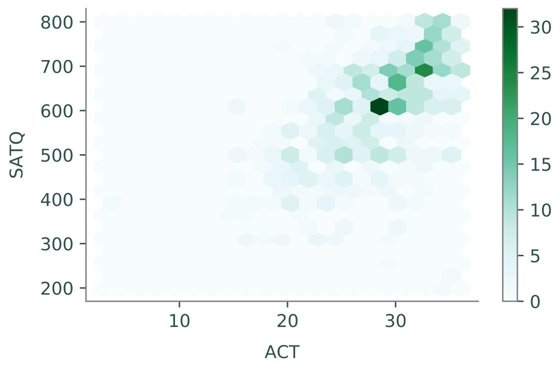
```
>>> satact = pd.read_csv("sat_act.csv", index_col="ID")
>>> list(satact.columns)
['gender', 'education', 'age', 'ACT', 'SATV', 'SATQ']

# Plot the ACT scores against the SAT Quant scores in a regular scatter plot.
>>> satact.plot(kind="scatter", x="ACT", y="SATQ", alpha=.8)

# Plot the densities of the ACT vs. SATQ scores with a hexbin plot.
>>> satact.plot(kind="Hexbin", x="ACT", y="SATQ", gridsize=20)
```



(a) ACT vs. SAT Quant scores.



(b) Frequency of ACT vs. SAT Quant scores.

Figure 10.8: Scatter plots and hexbin plot of SAT and ACT scores.

Just as choosing a good number of `bins` is important for a good histogram, choosing a good `gridsize` is crucial for an informative hexbin plot. A large `gridsize` creates many small bins and a small `gridsize` creates fewer, larger bins.

Note

Since hexbins are based on frequencies, they are prone to being misleading if the dataset is not understood well. For example, when plotting information that deals with geographic position, increases in frequency may be results in higher populations rather than the actual information being plotted.

See `http://pandas.pydata.org/pandas-docs/stable/visualization.html` for more types of plots available in Pandas and further examples.

Problem 3. Use `crime_data.csv` to display the following distributions.

1. The distributions of `Burglary`, `Violent`, and `Vehicle Theft` across all crimes,
2. The distributions of `Vehicle Thefts` over the values of `Robbery`.

As usual, all plots should be labeled and easy to read.

Principles of Good Data Visualization

Data visualization is a powerful tool for analysis and communication. When writing a paper or report, the author must make many decisions about how to use graphics effectively to convey useful information to the reader. Here we will go over a simple process for making deliberate, effective, and efficient design decisions.

Attention to Detail

Consider the plot in Figure 10.9. 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. It has not specified the units for the x or the y axis, nor does it tell what `cons` is. There is no title, and the source of the data is unknown.

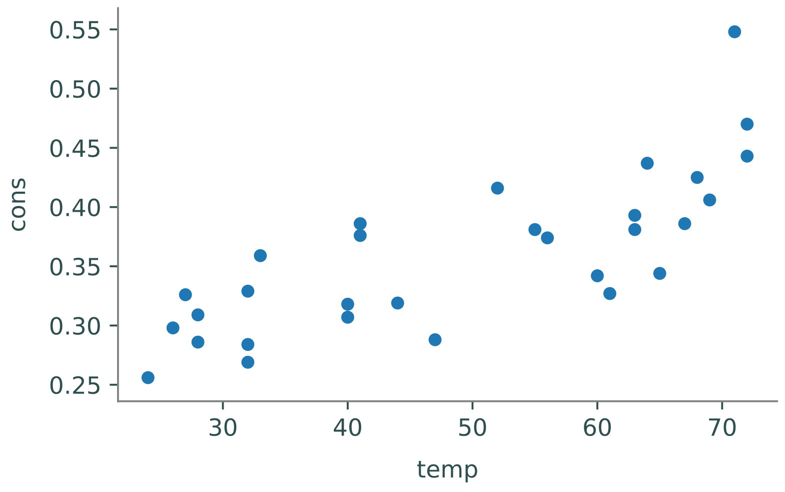


Figure 10.9: Non-specific data.

Labels and Citations

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, even plagiaristic. Data needs to be explained in a useful manner that includes all of the vital information.

Consider again Figure 10.9. This figure comes from the `Icecream` dataset within the `pydataset` package, which we store here in a dataframe and then plot:

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

This code produces the rather substandard plot in Figure 10.9. Examining the source of the dataset can give important details to create better plots. When plotting data, make sure to understand what the variable names represent and where the data was taken from. Use this information to create a more effective plot.

The ice cream data used in Figure 10.9 is better understood with 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."

This information gives important details that can be used in the following code. As seen in previous examples, pandas automatically generates legends when appropriate. Pandas also automatically labels the x and y axes, however 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 in units of *pints per head*, in place of the ambiguous 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)")
```

To add the necessary text to the figure, use either `plt.annotate()` or `plt.text()`. Alternatively, add text immediately below wherever the figure is displayed.

```
>>> plt.text(20, .1, r"Source: Hildreth, C. and J. Lu (1960) \emph{Demand"
...     "relations with autocorrelated disturbances}\nTechnical Bulletin No"
...     "2765, Michigan State University.", fontsize=7)
```

Both of these methods are imperfect but can normally be easily replaced by a caption attached to the figure. Again, we 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 in Figure 10.10.

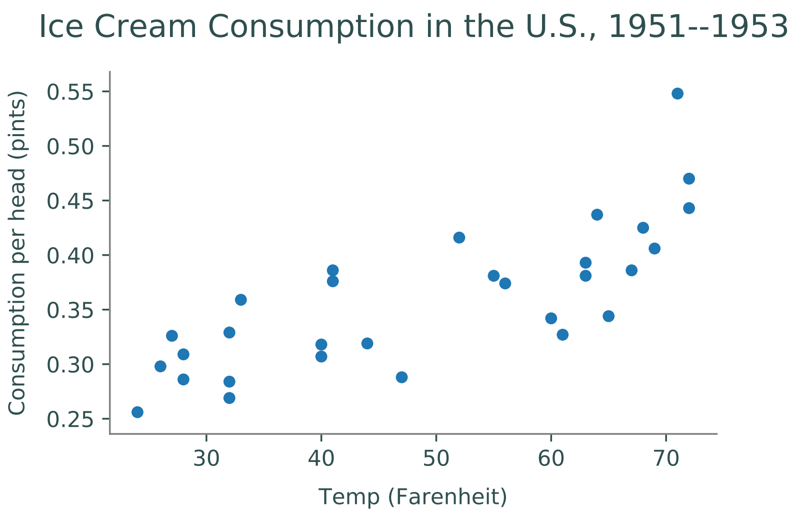


Figure 10.10: Source: Hildreth, C. and J. Lu (1960) *Demand relations with autocorrelated disturbances*, Technical Bulletin No 2765, Michigan State University.

Achtung!

Visualizing data can inherit many biases of the visualizer and as a result can be intentionally misleading. Examples of this include, but are not limited to, visualizing subsets of data that do not represent the whole of the data and having purposely misconstrued axes. Every data visualizer has the responsibility to avoid including biases in their visualizations to ensure data is being represented informatively and accurately.

Problem 4. The dataset `college.csv` contains information from 1995 on universities in the United States. To access information on variable names, go to `https://cran.r-project.org/web/packages/ISLR/ISLR.pdf`. Create 3 plots that compare variables or universities. These plots should answer questions about the data, e.g. what is the distribution of graduation rates or do schools with more PhD students also take more students from the Top 10 percent of their high school class. These plots should be easy to understand, have clear variable names, and citations.