

Lab 12

Data Visualization

Lab Objective: *Correctly presenting and interpreting data with visualizations is both a science and an art. In this lab we discuss and demonstrate the principles of good data visualization, including the key characteristics of good graphs, common mistakes, and how to avoid unethical visualizations.*

We strongly recommend completing this lab as a Jupyter Notebook.

The Importance of Visualizations

Visualizations of data often reveal insights that may not immediately be obvious from simple statistics. The following data set, known as *Anscombe's quartet*, is a famous example of the importance of graphing data.

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Each section of Anscombe's quartet shares the following statistical properties:

- The means are 9 for x and 7.5 for y .
- The sample variances are 11 for x and 3.75 for y .
- The correlation between x and y is .816.
- The linear least squares regression line is $y = \frac{1}{2}x + 3$.

Despite these similarities, each section is quite different from the others.

Problem 1. The file `anscombe.npy` contains Anscombe's quartet in the same format as displayed above. Plot each section of the quartet separately as a scatter plot. Also plot the regression line $y = \frac{1}{2}x + 3$ on the domain $x \in [0, 20]$ over each scatter plot.

Write a few sentences describing what makes each section unique.

Principles of Good Data Visualization

Understanding a data set through visualizations is an iterative process. Examine the data, start with an initial visualization, then adjust the original visualization or create new ones. Ask the following questions while searching for insights:

1. Does the visualization represent the data accurately?
2. Would a different visualization communicate more information?
3. Would visualizing a subset of the data provide more information?
4. Would transforming the data reveal a hidden pattern?

Effectively visualizing data involves technical expertise combined with design knowledge. No visualization is perfect, but every good visualization must contain the following essential elements.

- **Clarity.** Good visualizations are as self-explanatory as possible. Always use specific titles and axis labels, and include units of measure. Use a legend or other annotations where appropriate.
- **Simplicity.** A good visualization communicates everything that it needs to, but nothing more. Anything in a plot that fails to communicate information or that misrepresents the data is called *chartjunk*, a term coined by Edward Tufte. Make visualizations as simple, clean, and readable as possible.
- **Integrity.** Tell the truth, the whole truth, and nothing but the truth. It is usually alarmingly easy to manipulate a visualization so that it supports a specific agenda. Resist the temptation to morph a visualization into something that misrepresents the true nature of the data, even though that misrepresentation might support your hypotheses about the data.

Every visualization should be presented together with information on who created it, where the data was obtained, how it was collected, whether it was cleaned or transformed, and whether there are conflicts of interest or possible biases present. Cite your sources.

This list could be expanded, but virtually every good data visualization principle fits into these three categories in one way or another.

Improving Specific Types of Visualizations

Data can be visualized in many forms and styles. However, Most data sets are more naturally described with one type of visualization than another. In the following sections, we explore how the plots we commonly use can be improved and refined.

Line Plots

A line plot connects ordered (x, y) points with straight lines, and is therefore best for visualizing one or two ordered arrays, such as functional outputs over an ordered domain or a sequence of values over time.

When creating a line plot, consider the following details.

- What do the axes represent? How should they be labeled?
- Would a linear scale or a logarithmic scale most clearly reveal patterns?
- What should the window limits be?
- Is the line an appropriate thickness and color?
- Should each point be distinctly marked, or is a smooth line preferable?
- Should multiple lines be in the same plot, or in separate subplots?

The *Chebyshev polynomials* are a family of orthogonal polynomials that are recursively defined as follows.

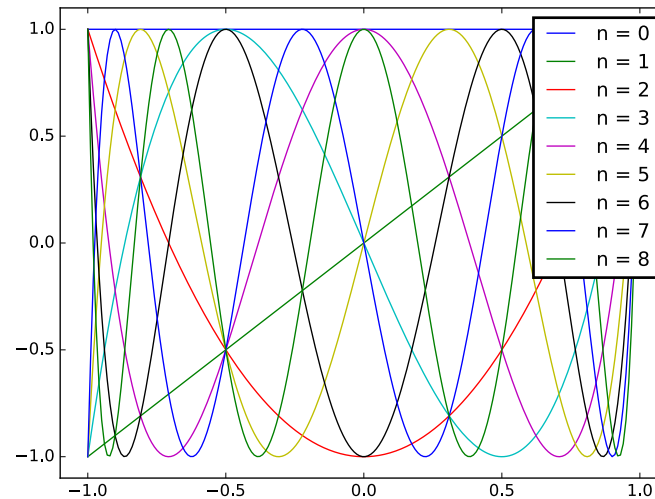
$$T_0(x) = 1 \quad T_1(x) = x \quad T_{n+1} = 2xT_n(x) - T_{n-1}(x)$$

NumPy's `polynomial` module has a convenient tool for constructing these and other important polynomials.¹ However, plotting several the polynomials on top of each other, especially with a legend, results in a very cluttered visual.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> %matplotlib inline           # Display notebook plots inline.

# Plot the first 9 Chebyshev polynomials in the same plot.
>>> T = np.polynomial.Chebyshev.basis
>>> x = np.linspace(-1, 1, 200)
>>> for n in range(9):
...     plt.plot(x, T(n)(x), label="n = "+str(n))
...
>>> plt.axis([-1.1, 1.1, -1.1, 1.1])    # Set the window limits.
>>> plt.legend()
```

¹`numpy.polynomial` also has tools for computing other important polynomial families, including the Legendre, Hermite, and Laguerre polynomials.



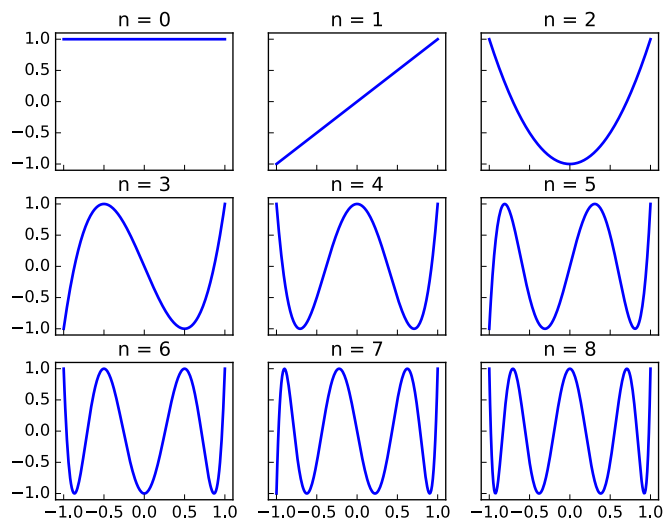
This line plot can be improved in several easy ways.

1. Use subplots to split the visualization into smaller, comparable pieces. Instead of using a legend, give each subplot a title. This method, called *small multiples*, was made famous by Edward Tufte.
2. Increase the line thicknesses to 2 or 3 (the default is 1).
3. Remove extra tick marks and axis labels.

Matplotlib's `plt.tick_params()`, summarized below, controls which tick marks and labels are displayed.

Argument	Options	Description
<code>axis</code>	<code>'x', 'y', "both"</code>	Axis on which to operate.
<code>which</code>	<code>"major", "minor", "both"</code>	Operate on major or minor ticks.
<code>color</code>	Any Matplotlib color	Tick color.
<code>labelcolor</code>	Any Matplotlib color	Tick label color.
<code>bottom, top, left, right</code>	<code>"on", "off"</code>	Turn ticks on or off.
<code>labelbottom, labeltop, labelleft, labelright</code>	<code>"on", "off"</code>	Turn tick labels on or off.

```
>>> for n in range(9):
...     plt.subplot(3, 3, n+1)
...     plt.plot(x, T(n)(x), lw=2)
...     plt.axis([-1.1, 1.1, -1.1, 1.1])
...
...     # Turn off extra tick marks and axis labels.
...     plt.tick_params(which="both", top="off", right="off")
...     if n < 6:
...         # Remove x-axis label on upper plots.
...         plt.tick_params(labelbottom="off")
...     if n % 3:
...         # Remove y-axis label on right plots.
...         plt.tick_params(labelleft="off")
...     plt.title("n = "+str(n))
```



NOTE

Matplotlib titles and annotations can be formatted with \LaTeX , a system for creating technical documents.^a To do so, use an ‘r’ before the string quotation mark and surround the text with dollar signs. For example, try replacing the final line of code in the previous example with the following line.

```
... plt.title(r"$T_{\{x\}}$.format(n))
```

The string’s `format()` method inserts the input n at the curly braces. The title of the sixth subplot, instead of being “n = 5,” will then be “ $T_5(x)$.”

^aSee <http://www.latex-project.org/> for more information.

Problem 2. The $n + 1$ Bernstein basis polynomials of degree n are defined as follows.

$$b_{v,n} = \binom{n}{v} x^v (1-x)^{n-v}, \quad v = 0, 1, \dots, n$$

Plot at least the first 10 Bernstein basis polynomials ($n = 0, 1, 2, 3$) as small multiples on the domain $[0, 1] \times [0, 1]$. Label the subplots for clarity, adjust tick marks and labels for simplicity, and set the window limits of each plot to be the same. Consider arranging the subplots so that the rows correspond with n and the columns with v .

(Hint: The constant $\binom{n}{v} = \frac{n!}{v!(n-v)!}$ is called the *binomial coefficient* and can be efficiently computed with `scipy.special.binom()` or `scipy.misc.comb()`.)

Scatter Plots

A scatter plot draws (x, y) points without connecting them. Connecting the points would imply an order or relation between the points, so scatter plots are best for displaying data sets without a natural order, or where each point is a distinct, individual instance.

Consider the following questions when making a scatter plot. Note that some of them are the same questions that should be asked when creating a line plot.

- What do the axes represent? How should they be labeled?
- Would a linear scale or a logarithmic scale most clearly reveal patterns?
- What should the window limits be?
- Which marker is best? Are the markers an appropriate size and color?

A scatter plot can be drawn with either `plt.plot()` (specify a point marker such as `'.'`, `'o'`, `'x'`, `'o'`, or `'+'`) or `plt.scatter()`. While `plt.plot()` is the more flexible function in general, `plt.scatter()` provides a few extra tools. Most useful are the keywords `s` and `c`, which correspond to marker size and marker color, respectively. Each keyword can either be a single entry or an array. Using an array specifies the sizes or colors of each individual marker, allowing a scatter plot to have up to four dimensions of information.

Consider a collection of rectangular boxes where the lengths, widths, and heights are given. A scatter plot of length against width mostly describes the sizes of the boxes; tying the third dimension (height) to the color of the points can provide the additional information. Setting the marker size as the volume of the boxes also adds some depth to the visualization, though modifying both the color and the size might be considered overkill.

Since adjusting the marker size may lead to overlapping points, we specify the *alpha value* of the color to make the markers slightly transparent. The keyword `alpha` accepts a value in the interval $[0, 1]$; 0 makes the markers completely transparent, while a 1 makes the markers completely opaque.

Finally, as with heat maps and contour plots, a color bar can be added with `plt.colorbar()` to indicate the values that the colors represent. This color bar can be given a label as well.

```
>>> length, width, height = np.random.randint(1, 20, (3,50))

>>> plt.subplot(221)                # Plot length against width.
>>> plt.scatter(length, width, s=100)
>>> plt.grid()
>>> plt.ylabel("Width (inches)")
>>> plt.tick_params(labelbottom="off")
>>> plt.axis([0, 20, 0, 20])

# Continued on the next page.
```

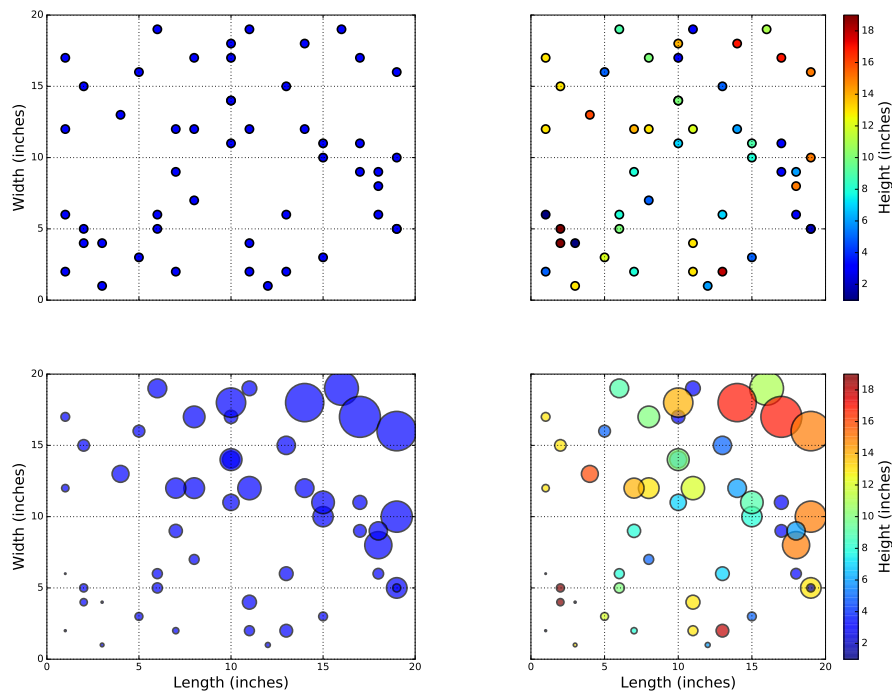
```

>>> plt.subplot(222)                    # Set the marker color to the height.
>>> plt.scatter(length, width, c=height, s=100)
>>> cbar = plt.colorbar()
>>> cbar.set_label("Height (inches)")
>>> plt.grid()
>>> plt.tick_params(labelbottom="off", labelleft="off")
>>> plt.axis([0, 20, 0, 20])

>>> plt.subplot(223)                    # Set the marker size to half the volume.
>>> plt.scatter(length, width, s=length*width*height/2., alpha=.7)
>>> plt.grid()
>>> plt.xlabel("Length (inches)")
>>> plt.ylabel("Width (inches)")
>>> plt.axis([0, 20, 0, 20])

>>> plt.subplot(224)                    # Use color and marker size together.
>>> plt.scatter(length, width, c=height, s=length*width*height/2., alpha=.7)
>>> cbar = plt.colorbar()
>>> cbar.set_label("Height (inches)")
>>> plt.grid()
>>> plt.tick_params(labelleft="off")
>>> plt.xlabel("Length (inches)")
>>> plt.axis([0, 20, 0, 20])

```



In scatter plots, connecting the points into a line plot usually results in extreme clutter. A regression line, however, highlights a pattern in the data without overshadowing the actual data points.²

²See the Least Squares lab (QR 2), especially Problems 2–4, for a refresher on regression lines.

Problem 3. The file `MLB.npy` contains measurements from over 1,000 recent Major League Baseball players, compiled by UCLA.^a Each row in the array represents a different player; the columns are the player's height (in inches), weight (in pounds), and age (in years), in that order.

Describe the data with at least one scatter plot. Your graph(s) should demonstrate whether height, weight, or age correlated with each other in the MLB. Consider plotting linear regression lines to indicate trends.

^aSee http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_MLB_HeightsWeights.

Histograms

A histogram partitions an interval into a number of bins and counts the number of values that fall into each bin. Histograms are ideal for visualizing how unordered data in a single array is distributed over an interval. If the data are draws from a probability distribution, a histogram approximates the distribution's probability density function (PDF).

The following are important factors to consider when constructing a histogram.

- What does the x -axis represent? How should it be labeled?
- Is a linear or a logarithmic scale more appropriate for the frequency axis?
- How many bins should be used? Over what range should the bins be? This is perhaps the most important question, as a histogram with too few or too many bins usually fails to give a clear view of the distribution.

For most histograms, the most desirable insight is the general shape of the distribution. The lines separating the bins, axis tick marks, and even the labels on the y axis are all unnecessary (and potentially distracting) details. Removing the lines between bins is easy: set the line width to 0. To ensure the gaps where the lines used to be are filled, specify the `histtype` as `"stepfilled"` (using `histtype="step"` will draw the outline without filling it in). Finally, to get rid of extra markings on the axes, use `plt.tick_params()`.

A histogram can be converted into a line plot by using `np.histogram()`. This function returns the number of values in each bin and the locations of the edges of the bins (in fact, `plt.hist()` employs this function). Use the edges of the bins to calculate the center of the bins, then plot the bin centers against the frequency.

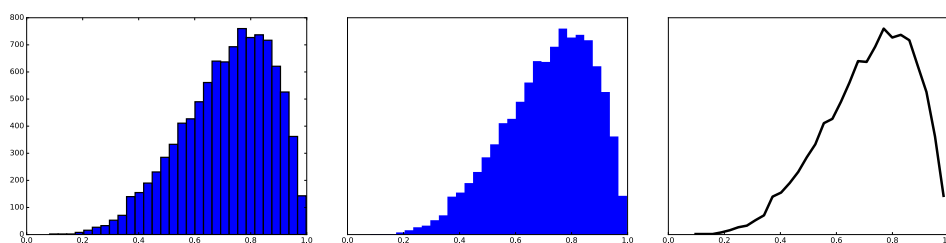
```
# Get 10,000 random samples from a Beta distribution.
>>> data = np.random.beta(a=5, b=2, size=N)

>>> plt.subplot(131)                # Draw a regular histogram.
>>> plt.hist(data, bins=30)

>>> plt.subplot(132)                # Draw a clean histogram.
>>> plt.hist(data, bins=30, lw=0, histtype="stepfilled")
>>> plt.tick_params(left="off", top="off", right="off", labelleft="off")
# Continued on the next page.
```



```
>>> plt.subplot(133)                                # Convert the histogram to a line plot.
>>> freq, bin_edges = np.histogram(data, bins=30)
>>> bin_centers = (bin_edges[:-1] + bin_edges[1:])/2.
>>> plt.plot(bin_centers, freq, 'k-', lw=4)
>>> plt.tick_params(left="off", top="off", right="off", labelleft="off")
```



Finally, if the frequency domain is better visualized on a logarithmic scale, use `log=True` as an argument to `plt.hist()`. This is the histogram equivalent of using `plt.semilogy()` for line or scatter plots.

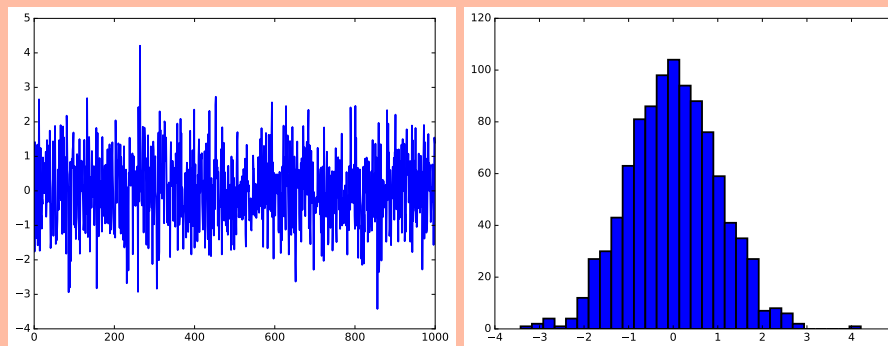
ACHTUNG!

Line plots should **not** be used for data that has no natural progression. For example, consider a collection of random draws from a statistical distribution. In this case, a plain line plot is completely useless, because consecutive random draws are completely unrelated. A histogram, on the other hand, provides an approximation of the distribution's probability density function.

```
# Get 1,000 random samples from the standard normal distribution.
>>> data = np.random.normal(size=1000)

>>> plt.subplot(121)                                # Regular line plot of the data.
>>> plt.plot(data)

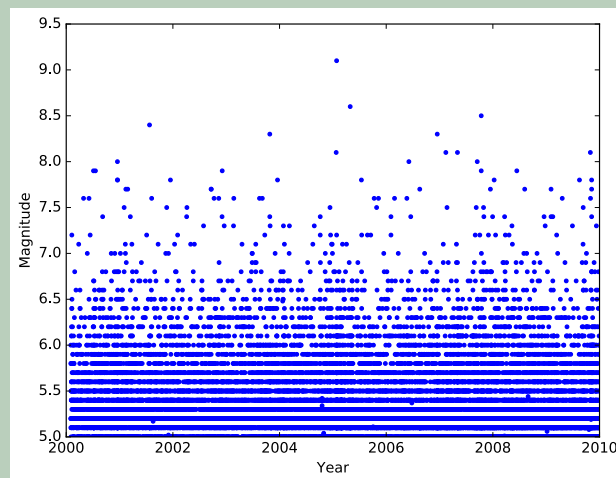
>>> plt.subplot(122)                                # Histogram of the data.
>>> plt.hist(data, bins=30)
```



Problem 4. The file `earthquakes.npy` contains data from over 17,000 earthquakes between 2000 and 2010 that were at least a 5 on the Richter scale.^a Each row in the array represents a different earthquake; the columns are the earthquake's date (as a fraction of the year), magnitude (on the Richter scale), longitude, and latitude, in that order.

Because each earthquake is a distinct event, a good way to start visualizing this data might be a scatter plot of the years versus the magnitudes of each earthquake.

```
>>> year, magnitude, longitude, latitude = np.load("earthquakes.npy").T
>>> plt.plot(year, magnitude, '.')
>>> plt.xlabel("Year")
>>> plt.ylabel("Magnitude")
```



Unfortunately, this plot communicates very little information because the data is so cluttered. Describe the data with two or three better visualizations, including line plots, scatter plots, and histograms as appropriate. Your plots should clearly answer the following questions:

1. How many earthquakes happened every year?
2. How often do stronger earthquakes happen compared to weaker ones?
3. Where do earthquakes happen? Where do the strongest earthquakes happen?

(Hint: Use `plt.axis("equal")` to fix the aspect ratio, which may improve comparisons between longitude and latitude.)

^aSee <http://earthquake.usgs.gov/earthquakes/search/>.

Heat Maps and Contour Plots

Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ be a scalar-valued function on a 2-dimensional domain. A heat map of f assigns a color to each (x, y) point in the domain based on the value of $f(x, y)$, while a contour plot is a drawing of the *level curves* of f . The level curve corresponding to the constant c is the set $\{(x, y) \mid c = f(x, y)\}$. A filled contour plot, which colors in the sections between the level curves, is a discretized version of a heat map.

Consider the following questions when plotting a heat map or contour plot:

- Is the domain sufficiently refined?
- Which color scheme is most clear and effective?
- How many / which contour lines should be drawn, if any?
- Is a linear or a logarithmic scale more appropriate for the color?

It is often sufficient to choose a fixed number of level curves. In this case, the values of c corresponding to the level curves are automatically chosen to be evenly spaced over the range of values of f on the domain. However, it is sometimes better to strategically specify the curves by providing a list of c constants.

Consider the function $f(x, y) = y^2 - x^3 + x^2$ on the domain $[-\frac{3}{2}, \frac{3}{2}] \times [-\frac{3}{2}, \frac{3}{2}]$. A heat map of f reveals that it has a large basin around the origin. Then since $f(0, 0) = 0$, choosing several level curves close to 0 more closely describes the topography of the basin. The fourth subplot in the following example uses the curves with $c = -1, -\frac{1}{4}, 0, \frac{1}{4}, 1$, and 4.

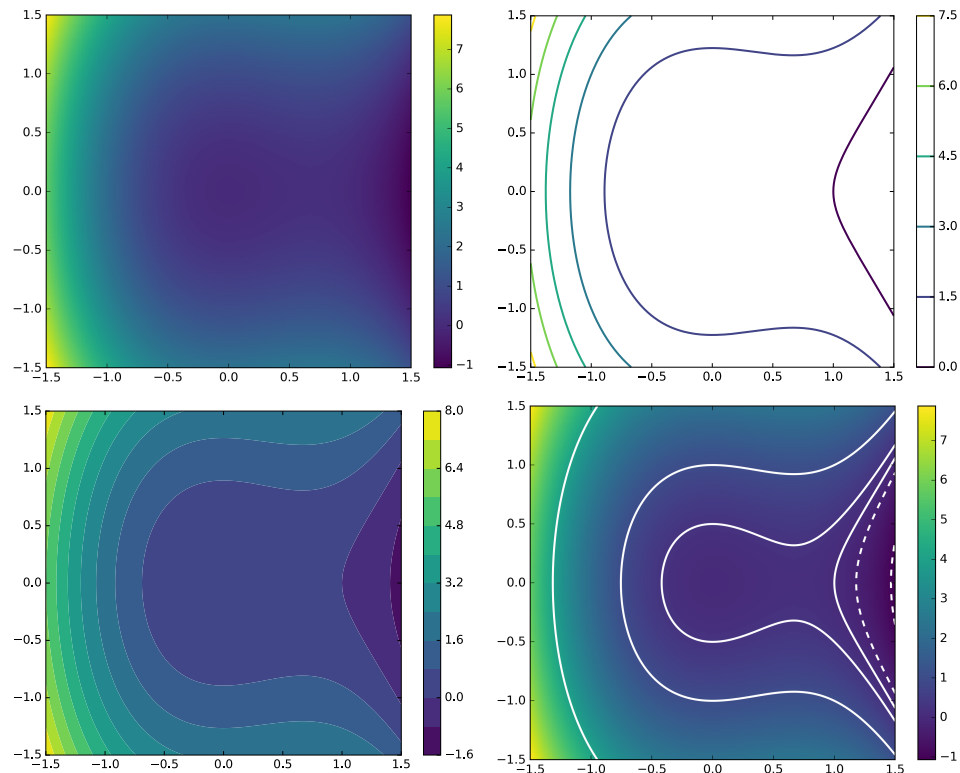
```
# Construct a 2-D domain with np.meshgrid() and calculate f on the domain.
>>> x = np.linspace(-1.5, 1.5, 200)
>>> X, Y = np.meshgrid(x, x.copy())
>>> Z = Y**2 - X**3 + X**2

>>> plt.subplot(221)                # Plot a heat map of f.
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(222)                # Plot a contour map with 6 level curves.
>>> plt.contour(X, Y, Z, 6, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(223)                # Plot a filled contour map with 12 levels.
>>> plt.contourf(X, Y, Z, 12, cmap="viridis")
>>> plt.colorbar()

>>> plt.subplot(224)                # Plot specific level curves and a heat map.
>>> plt.contour(X, Y, Z, [-1, -.25, 0, .25, 1, 4], colors="white")
>>> plt.pcolormesh(X, Y, Z, cmap="viridis")
>>> plt.colorbar()
```



There are two main kinds of color maps: sequential and diverging. Sequential color maps, like "hot" and "cool", transition very gradually between two colors; diverging color maps, like "seismic" and "coolwarm", transition very rapidly from one color to another at the mean value. When in doubt, use "viridis" or "plasma", two specialized sequential color schemes. For the complete list of Matplotlib color maps, see http://matplotlib.org/examples/color/colormaps_reference.html.

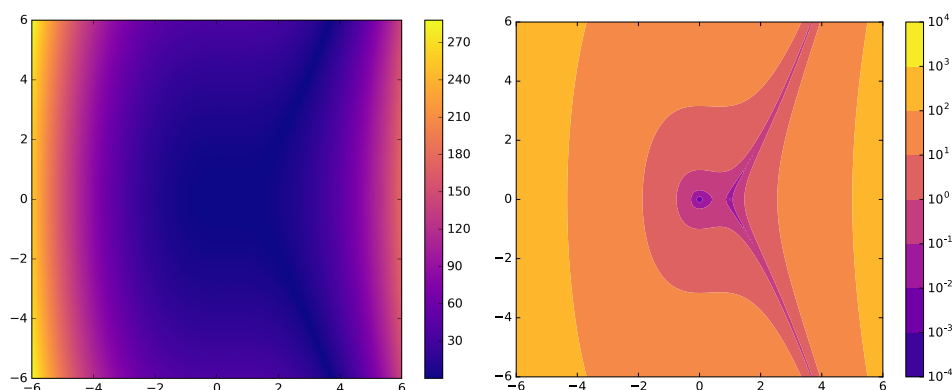
The color map can be changed to a logarithmic scale by using the keyword argument `norm=matplotlib.colors.LogNorm()` (this works for `plt.scatter()` as well). As an example, consider the same f defined above on the larger domain $[-6, 6] \times [-6, 6]$. Log scaling can only be done on arrays of all positive values, so we visualize $|f|$.

```
>>> from matplotlib.colors import LogNorm

>>> x = np.linspace(-6, 6, 200)
>>> X, Y = np.meshgrid(x, x.copy())
>>> Z = np.abs(Y**2 - X**3 + X**2)

>>> plt.subplot(121)                                # Plot a regular heat map of |f|.
>>> plt.pcolormesh(X, Y, Z, cmap="plasma")
>>> plt.colorbar()

>>> plt.subplot(122)                                # Plot a filled contour plot with log scaling.
>>> plt.contourf(X, Y, Z, 6, cmap="plasma", norm=LogNorm())
>>> plt.colorbar()
```



Problem 5. The *Rosenbrock function* is defined as follows.

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

The minimum value of f is 0, which occurs at the point $(1, 1)$ at the bottom of a steep, banana-shaped valley of the function.

Use heat maps and contour plots to visualize the Rosenbrock function in such a way that the minimum value, and the valley that it lies in, is apparent. Consider plotting the minimizer $(1, 1)$ on top of the plot as well.

Bar Charts

A bar chart plots categorical data in a sequence of bars. They are best for small, discrete, one-dimensional data sets. Use `plt.bar()` (vertical) or `plt.barh()` (horizontal) to create a bar chart in Matplotlib. These functions receive the locations of each bar, then the height of each bar (as lists or arrays).

Consider the following questions when constructing a bar chart.

- What are the different categories in the data?
- How should the data be sorted?
- Should the value axis have a linear or a logarithmic scale?

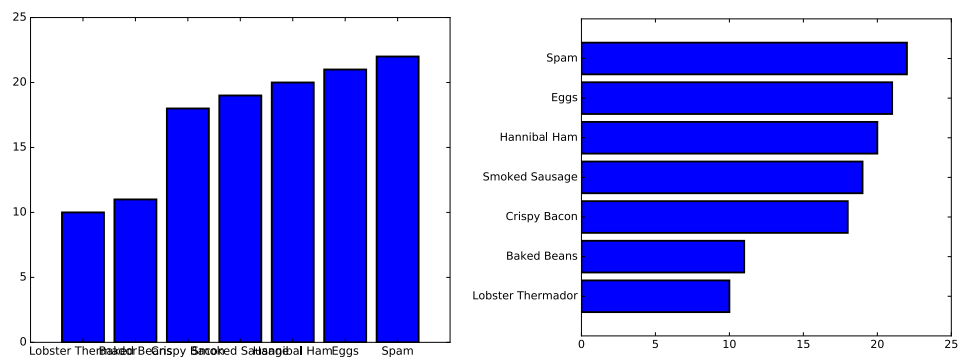
Horizontal bar charts are usually preferable to vertical bar charts because horizontal labels are easier to read than vertical labels. Labels can be rotated, but it is better when the reader doesn't have to turn his or her head.

Data in a bar chart should also be sorted in a logical way. Sort the categories by bar size, alphabetize the labels, or use some other intuitive ordering.

```
>>> labels = ["Lobster Thermador", "Baked Beans", "Crispy Bacon",
...           "Smoked Sausage", "Hannibal Ham", "Eggs", "Spam"]
>>> values = [10, 11, 18, 19, 20, 21, 22]
>>> positions = np.arange(len(labels))

>>> plt.subplot(121)
>>> plt.bar(positions, values, align="center")
>>> plt.xticks(positions, labels)

>>> plt.subplot(122)
>>> plt.barh(positions, values, align="center")
>>> plt.yticks(positions, labels)
```



Practices to Avoid

Bad Types of Visualization

Some kinds of visualizations, especially for categorical data, are popular even though they interfere with the reader's ability to interpret the displayed information. For example, it is more difficult to see the differences in a pie chart than on a bar chart since differences in area are more difficult to detect than differences in length. By the same logic, bar chart with several bars per category are almost always better with the bars side by side rather than stacked.

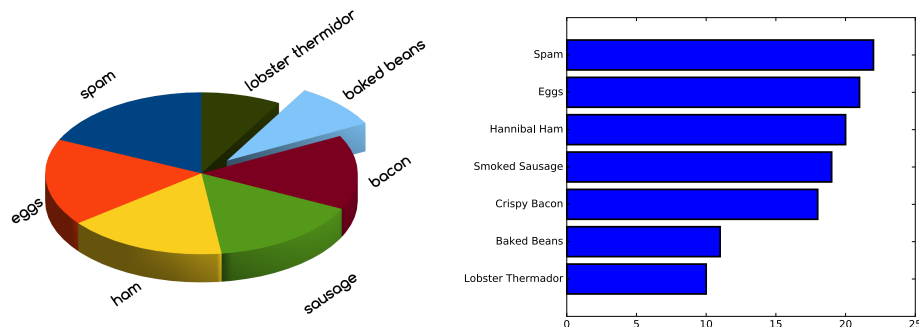


Figure 12.7: Transforming the horrendous 3-D pie chart on the left into the horizontal bar chart on the right both simplifies and clarifies the information.

Misrepresenting Data

To conclude, we return to the issue of integrity. It is very easy to misrepresent the true nature of a set of data with a visualization. Perhaps the easiest way to do so is with poorly chosen window limits or axis scales. Consider the following example of a scatter plot, presented four different ways. The fourth subplot is the only one that correctly shows the position of the points in relation to the origin and that has a reasonable scales on both axes.

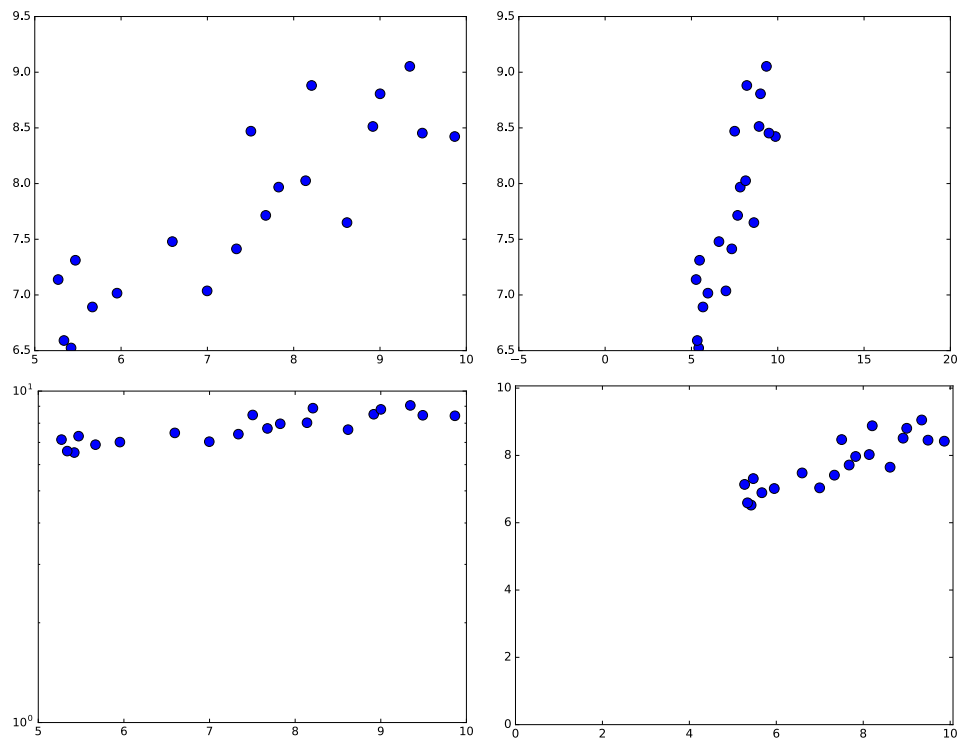
```
>>> x = np.linspace(5, 10, N) + np.random.normal(size=N)/3.
>>> y = .5*x + 4 + np.random.normal(size=N)/2.

>>> plt.subplot(221)                # Plot a default scatter plot of the data.
>>> plt.plot(x, y, 'o', ms=10)

>>> plt.subplot(222)                # Change the window limits so that the
>>> plt.plot(x, y, 'o', ms=10)      # data appears steep in the y direction.
>>> plt.xlim(-5,20)

>>> plt.subplot(223)                # Change the y-axis scale so that the data
>>> plt.semilogy(x, y, 'o', ms=10) # appears flat in the y direction.

>>> plt.subplot(224)                # Plot the data with equal axis scales and
>>> plt.plot(x, y, 'o', ms=10)      # presenting the relation to the origin.
>>> plt.xlim(xmin=0)
>>> plt.ylim(ymin=0)
```



Do everything in your power to avoid visualizing data unethically.

Problem 6. The file `countries.npy` contains information from 20 different countries. Each row in the array represents a different country; the columns are the 2015 population (in millions of people), the 2015 GDP (in billions of US dollars), the average male height (in centimeters), and the average female height (in centimeters), in that order.^a

The countries corresponding are listed below in order.

```
countries = ["Austria", "Bolivia", "Brazil", "China",  
             "Finland", "Germany", "Hungary", "India",  
             "Japan", "North Korea", "Montenegro", "Norway",  
             "Peru", "South Korea", "Sri Lanka", "Switzerland",  
             "Turkey", "United Kingdom", "United States", "Vietnam"]
```

Visualize this data set with at least four plots, using at least one scatter plot, one histogram, and one bar chart. Write a few sentences summarizing the major insights that your visualizations reveal.

(Hint: consider using `np.argsort()` and fancy indexing to sort the data for the bar chart.)

^a See [https://en.wikipedia.org/wiki/List_of_countries_by_GDP_\(nominal\)](https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)), <http://www.averageheight.co/>, and https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population.