15

Designing Data Visualizations

Data visualization, when done well, allows you to reveal patterns in your data and communicate insights to your audience. This chapter describes the conceptual and design skills necessary to craft *effective* and *expressive* visual representations of your data. In doing so, it introduces skills for each of the following steps in the visualization process:

1. Understanding the *purpose* of visualization
2. Selecting a *visual layout* based on your question and data type
3. Choosing optimal *graphical encodings* for your variables
4. Identifying visualizations that are able to *express* your data
5. Improving the *aesthetics* (i.e., making it readable and informative)

15.1 The Purpose of Visualization

> *"The purpose of visualization is insight, not pictures."*[1]

Generating visual displays of your data is a key step in the analytical process. While you should strive to design aesthetically pleasing visuals, it's important to remember that visualization is a *means to an end*. Devising appropriate renderings of your data can help expose underlying patterns in your data that were previously unseen, or that were undetectable by other tests.

[1] Card, S. K., Mackinlay, J. D., & Shneiderman, B. (1999). *Readings in information visualization: Using vision to think*. Burlington, MA: Morgan Kaufmann.

To demonstrate how visualization makes a distinct contribution to the data analysis process (beyond statistical tests), consider the canonical data set **Anscombe's Quartet** (which is included with the R software as the data set `anscombe`). This data set consists of four pairs of x and y data: (x_1, y_1), (x_x, y_2), and so on. The data set is shown in Table 15.1.

The challenge of Anscombe's Quartet is to identify differences between the four pairs of columns. For example, how does the (x_1, y_1) pair differ from the (x_2, y_2) pair? Using a nonvisual approach to answer this question, you could compute a variety of descriptive statistics for each set, as shown in Table 15.2. Given these six statistical assessments, these four data sets *appear* to be identical. However, if you graphically represent the relationship between each x and y pair, as in Figure 15.1, you reveal the distinct nature of their relationships.

While computing summary statistics is an important part of the data exploration process, it is only through visual representations that differences across these sets emerge. The simple graphics in Figure 15.1 expose variations in the **distributions** of x and y values, as well as in the **relationships** between them. Thus the choice of representation becomes paramount when analyzing and presenting data. The following sections introduce basic principles for making that choice.

Table 15.1 **Anscombe's Quartet: four data sets with two features each**

x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
10.00	8.04	10.00	9.14	10.00	7.46	8.00	6.58
8.00	6.95	8.00	8.14	8.00	6.77	8.00	5.76
13.00	7.58	13.00	8.74	13.00	12.74	8.00	7.71
9.00	8.81	9.00	8.77	9.00	7.11	8.00	8.84
11.00	8.33	11.00	9.26	11.00	7.81	8.00	8.47
14.00	9.96	14.00	8.10	14.00	8.84	8.00	7.04
6.00	7.24	6.00	6.13	6.00	6.08	8.00	5.25
4.00	4.26	4.00	3.10	4.00	5.39	19.00	12.50
12.00	10.84	12.00	9.13	12.00	8.15	8.00	5.56
7.00	4.82	7.00	7.26	7.00	6.42	8.00	7.91
5.00	5.68	5.00	4.74	5.00	5.73	8.00	6.89

Table 15.2 **Anscombe's Quartet: the (*X*, *Y*) pairs share identical summary statistics**

Set	Mean *X*	Std. Deviation *X*	Mean *Y*	Std. Deviation *Y*	Correlation	Linear Fit
1	9.00	3.32	7.50	2.03	0.82	$y = 3 + 0.5x$
2	9.00	3.32	7.50	2.03	0.82	$y = 3 + 0.5x$
3	9.00	3.32	7.50	2.03	0.82	$y = 3 + 0.5x$
4	9.00	3.32	7.50	2.03	0.82	$y = 3 + 0.5x$

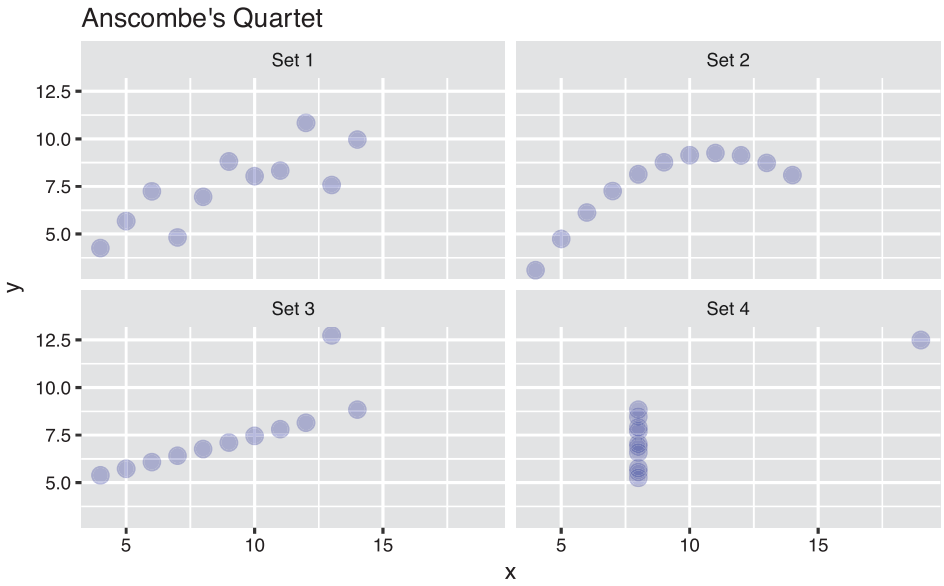

Figure 15.1 Anscombe's Quartet: scatterplots reveal four different (x, y) relationships that are not detectable using descriptive statistics.

15.2 Selecting Visual Layouts

The challenge of visualization, like many design challenges, is to identify an optimal solution (i.e., a visual layout) given a set of constraints. In visualization design, the primary constraints are:

1. The specific *question of interest* you are attempting to answer in your domain
2. The *type of data* you have available for answering that question
3. The limitations of the human *visual processing system*
4. The *spatial limitations* in the medium you are using (pixels on the screen, inches on the page, etc.)

This section focuses on the second of these constraints (data type); the last two constraints are addressed in Section 15.3 and Section 15.4. The first constraint (the question of interest) is closely tied to Chapter 10 on understanding data. Based on your domain, you need to hone in on a question of interest, and identify a data set that is well suited for answering your question. This section will expand upon the same data set and question from Chapter 10:

> *"What is the worst disease in the United States?"*

As with the Anscombe's Quartet example, most basic exploratory data questions can be reduced to investigating *how a variable is distributed* or *how variables are related* to one another. Once you have mapped from your question of interest to a specific data set, your visualization type will largely depend on the data type of your variables. The data type of each column—*nominal*, *ordinal*, or

continuous—will dictate how the information can be represented. The following sections describe techniques for visually exploring each variable, as well as making comparisons across variables.

15.2.1 Visualizing a Single Variable

Before assessing relationships *across* variables, it is important to understand how each individual variable (i.e., column or feature) is distributed. The primary question of interest is often *what does this variable look like?* The specific visual layout you choose when answering this question will depend on whether the variable is **categorical** or **continuous**. To use the disease burden data set as an example, you may want to know *what is the range* of the number of deaths attributable to each disease.

For continuous variables, a **histogram** will allow you to see the distribution and range of values, as shown in Figure 15.2. Alternatively, you can use a **box plot** or a **violin plot**, both of which are shown Figure 15.3. Note that **outliers** (extreme values) in the data set have been removed to better express the information in the charts.

While these visualizations display information about the distribution of the number of deaths by cause, they all leave an obvious question unanswered: *what are the names of these diseases*? Figure 15.4 uses a **bar chart** to label the top 10 causes of death, but due to the constraint of the page size, this display is able to express just a small subset of the data. In other words, bar charts don't easily scale to hundreds or thousands of observations because they are inefficient to scan, or won't fit in a given medium.

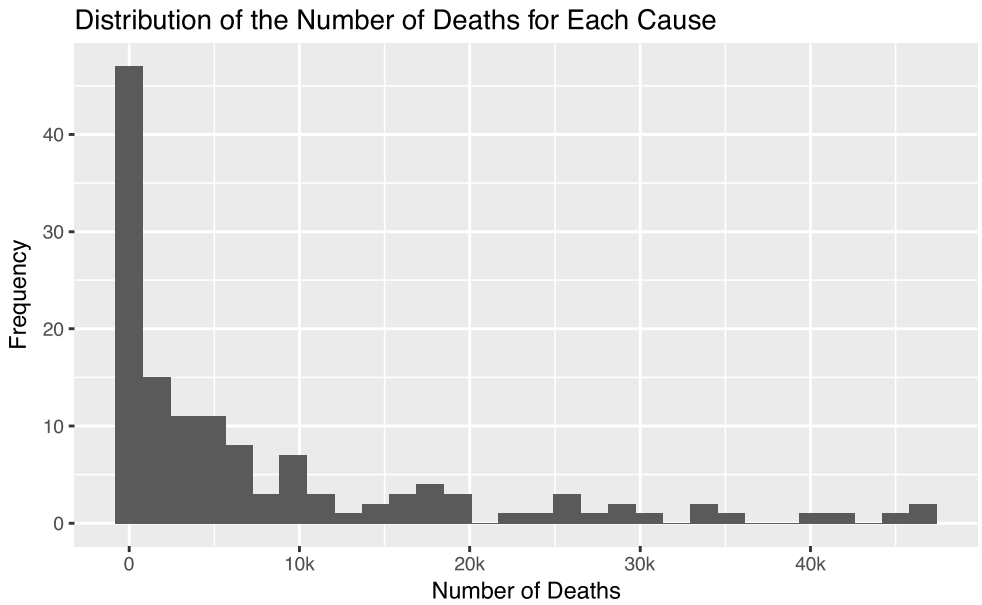

Figure 15.2 The distribution of the number of deaths attributable to each disease in the United States (a continuous variable) using a histogram. Some outliers have been removed for demonstration.

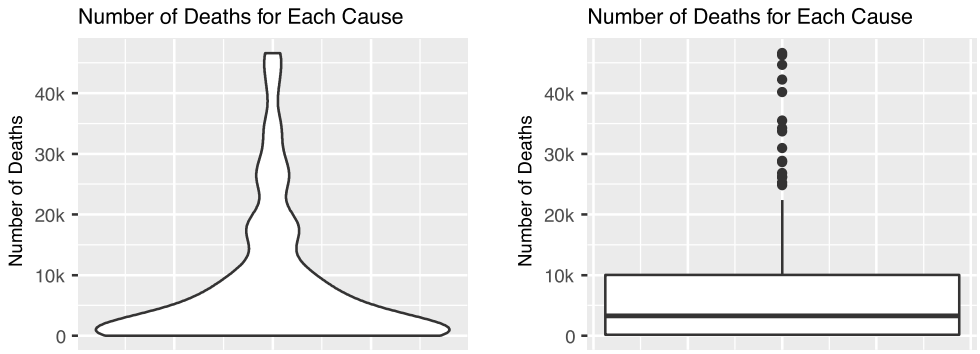

Figure 15.3 Alternative visualizations for showing distributions of the number of deaths in the United States: violin plot (left) and box plot (right). Some outliers have been removed for demonstration.

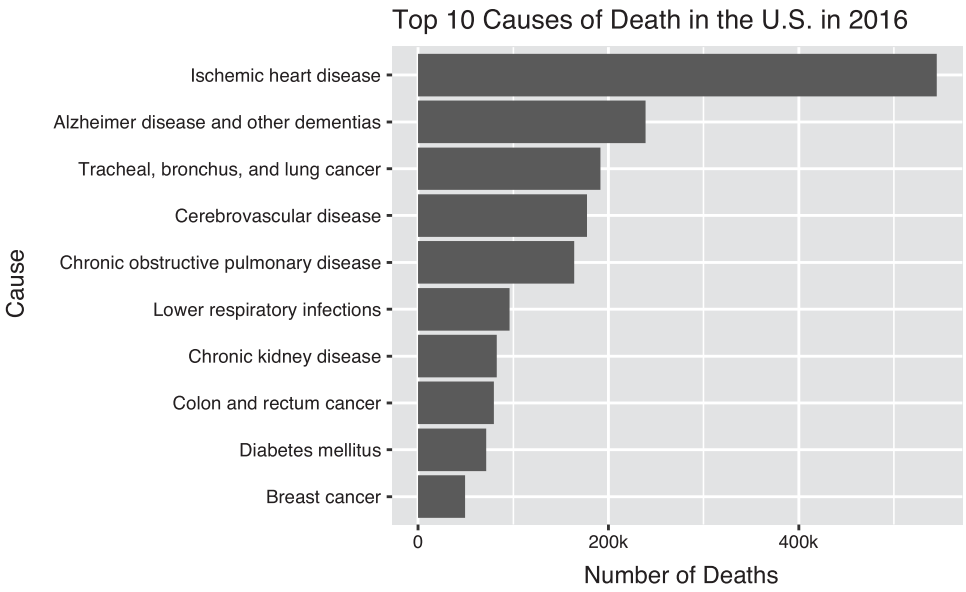

Figure 15.4 Top causes of death in the United States as shown in a bar chart.

15.2.1.1 Proportional Representations

Depending on the data stored in a given column, you may be interested in showing each value relative to the total of the column. For example, using the disease burden data set, you may want to express each value **proportional** to the total number of deaths. This allows you answer the question, *Of all deaths, what percentage is attributable to each disease?* To do this, you can transform the data to percentages, or use a representation that more clearly expresses *parts of a whole*. Figure 15.5 shows the use of a **stacked bar chart** and a **pie chart**, both of which more intuitively express proportionality. You can also use a **treemap**, as shown later in Figure 15.14, though the true

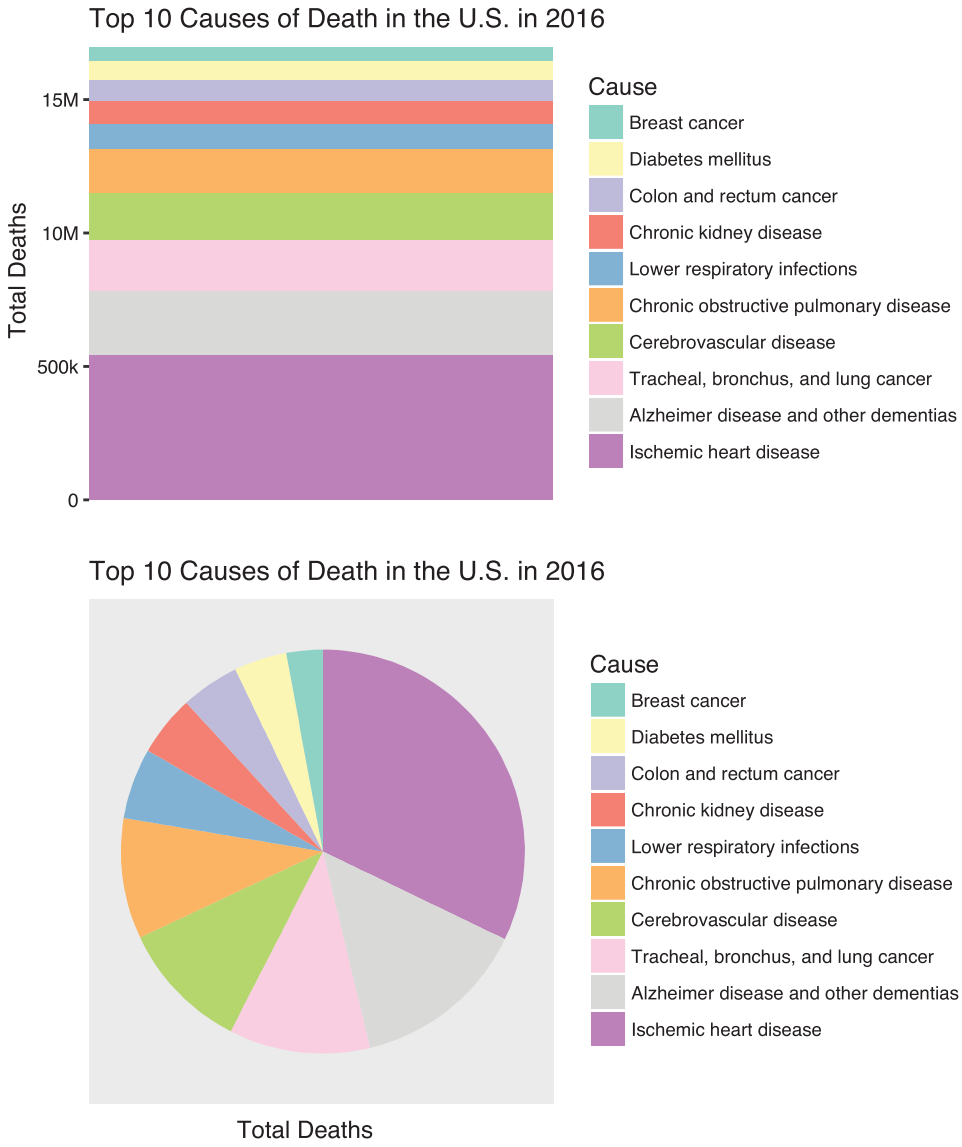

Figure 15.5 Proportional representations of the top causes of death in the United States: stacked bar chart (top) and pie chart (bottom).

benefit of a treemap is expressing hierarchical data (more on this later in the chapter). Later sections explore the trade-offs in *perceptual accuracy* associated with each of these representations.

If your variable of interest is a categorical variable, you will need to *aggregate your data* (e.g., count the number of occurrences of different categories) to ask similar questions about the distribution.

Once doing so, you can use similar techniques to show the data (e.g., bar chart, pie chart, treemap). For example, the diseases in this data set are categorized into three types of diseases: *non-communicable diseases*, such as heart disease or lung cancer; *communicable diseases*, such as tuberculosis or whooping cough; and *injuries*, such as road traffic accidents or self harm. To understand how this categorical variable (disease type) is distributed, you can count the number of rows for each category, then display those quantitative values, as in Figure 15.6.

15.2.2 Visualizing Multiple Variables

Once you have explored each variable independently, you will likely want to assess relationships between or across variables. The type of visual layout necessary for making these comparisons will (again) depend largely on the type of data you have for each variable.

For comparing relationships between two continuous variables, the best choice is a **scatterplot**. The visual processing system is quite good at estimating the linearity in a field of points created by a scatterplot, allowing you to describe how two variables are related. For example, using the disease burden data set, you can compare different metrics for measuring health loss. Figure 15.7 compares the disease burden as measured by the number of *deaths* due to each cause to the number of *years of life lost* (a metric that accounts for the age at death for each individual).

You can extend this approach to multiple continuous variables by creating a **scatterplot matrix** of all continuous features in the data set. Figure 15.8 compares all *pairs of metrics* of disease burden,

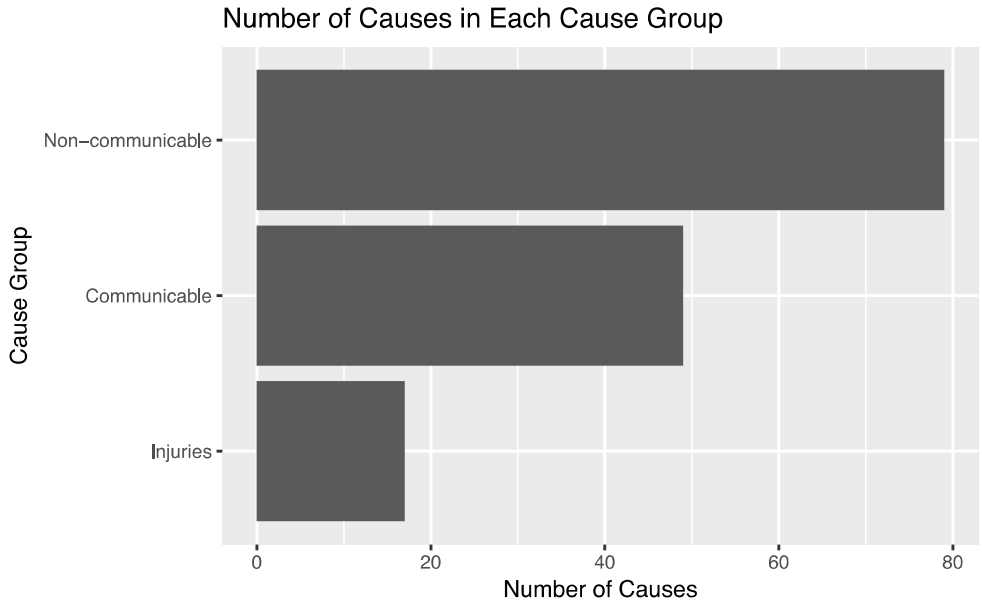

Figure 15.6 A visual representation of the number of causes in each disease category: non-communicable diseases, communicable diseases, and injuries.

Figure 15.7 Using a scatterplot to compare two continuous variables: the number of deaths versus the years of live lost for each disease in the United States.

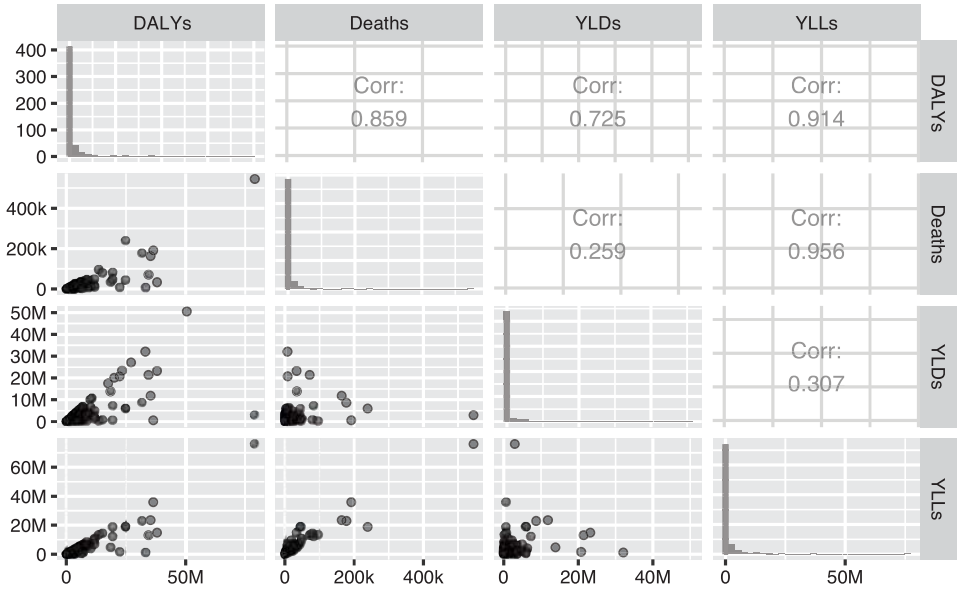

Figure 15.8 Comparing multiple continuous measurements of disease burden using a scatterplot matrix.

including number of deaths, years of life lost (*YLLs*), years lived with disability (*YLDs*, a measure of the disability experienced by the population), and disability-adjusted life years (*DALYs*, a combined measure of life lost and disability).

When comparing relationships between one continuous variable and one categorical variable, you can compute summary statistics for each group (see Figure 15.6), use a violin plot to display distributions for each category (see Figure 15.9), or use **faceting** to show the distribution for each category (see Figure 15.10).

For assessing relationships between two categorical variables, you need a layout that enables you to assess the *co-occurrences* of nominal values (that is, whether an observation contains both values). A great way to do this is to count the co-occurrences and show a **heatmap**. As an example, consider a broader set of population health data that evaluates the leading cause of death in each country (also from the Global Burden of Disease study). Figure 15.11 shows a subset of this data, including the disease type (communicable, non-communicable) for each disease, and the region where each country is found.

One question you may ask about this categorical data is:

> *"In each region, how often is the leading cause of death a communicable disease versus a non-communicable disease?"*

To answer this question, you can aggregate the data by region, and count the number of times each disease category (communicable, non-communicable) appears as the category for the leading cause of death. This aggregated data (shown in Figure 15.12) can then be displayed as a heatmap, as in Figure 15.13.

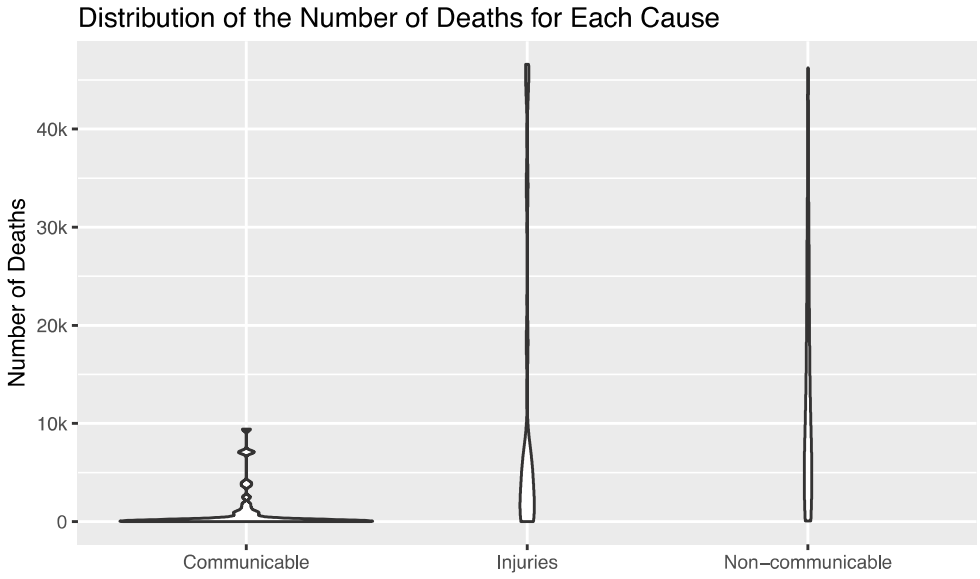

Figure 15.9 A violin plot showing the continuous distributions of the number of deaths for each cause (by category). Some outliers have been removed for demonstration.

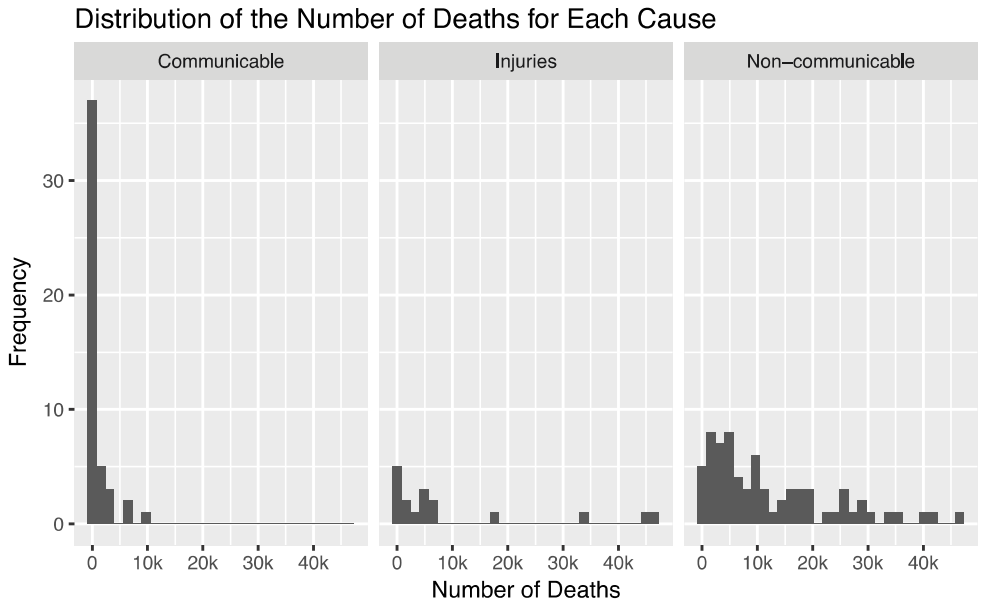

Figure 15.10 A faceted layout of histograms showing the continuous distributions of the number of deaths for each cause (by category). Some outliers have been removed for demonstration.

	country	region	leading_cause_of_death	category
24	Botswana	Southern Sub-Saharan Africa	HIV/AIDS	Communicable
25	Brazil	Tropical Latin America	Ischemic heart disease	Non-Communicable
26	Brunei	High-income Asia Pacific	Ischemic heart disease	Non-Communicable
27	Bulgaria	Central Europe	Ischemic heart disease	Non-Communicable
28	Burkina Faso	Western Sub-Saharan Africa	Malaria	Communicable
29	Burundi	Eastern Sub-Saharan Africa	Diarrheal diseases	Communicable
30	Cambodia	Southeast Asia	Lower respiratory infections	Communicable
31	Cameroon	Western Sub-Saharan Africa	HIV/AIDS	Communicable
32	Canada	High-income North America	Ischemic heart disease	Non-Communicable
33	Cape Verde	Western Sub-Saharan Africa	Ischemic heart disease	Non-Communicable

Figure 15.11 The leading cause of death in each country. The category of each disease (communicable, non-communicable) is shown, as is the region in which each country is found.

	region	category_of_leading_cause	number_of_countries
1	Andean Latin America	Communicable	1
2	Andean Latin America	Non-Communicable	2
3	Australasia	Non-Communicable	2
4	Caribbean	Non-Communicable	18
5	Central Asia	Non-Communicable	9
6	Central Europe	Non-Communicable	13
7	Central Latin America	Communicable	1
8	Central Latin America	Non-Communicable	8

Figure 15.12 Number of countries in each region in which the leading cause of death is communicable/non-communicable.

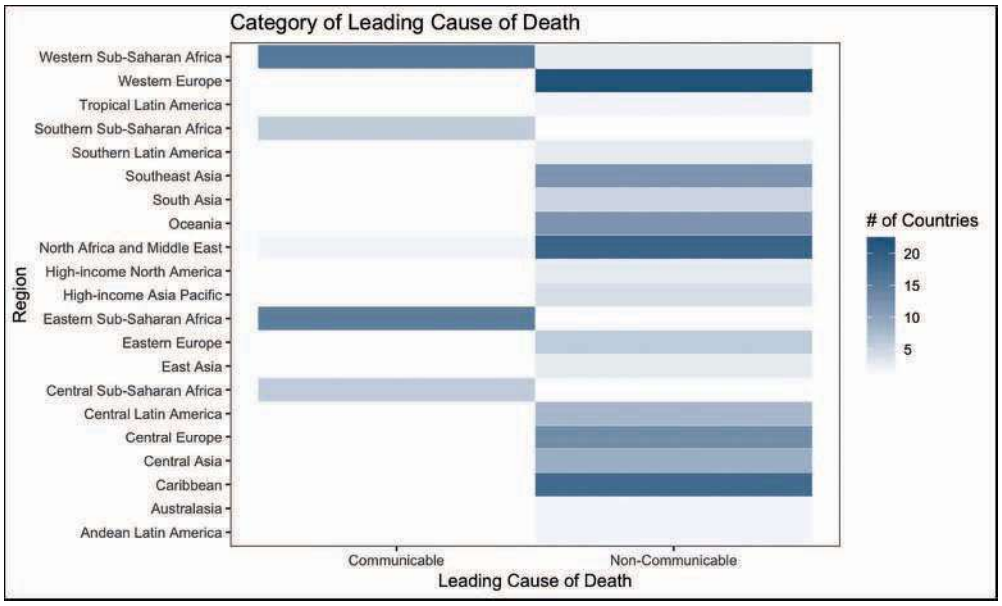

Figure 15.13 A heatmap of the number of countries in each region in which the leading cause of death is communicable/non-communicable.

15.2.3 Visualizing Hierarchical Data

One distinct challenge is showing a hierarchy that exists in your data. If your data naturally has a **nested structure** in which each observation is a member of a group, visually expressing that hierarchy can be critical to your analysis. Note that there may be multiple levels of nesting for each observation (observations may be part of a group, and that group may be part of a larger group). For example, in the disease burden data set, each country is found within a particular region, which can be further categorized into larger groupings called *super-regions*. Similarly, each cause of death

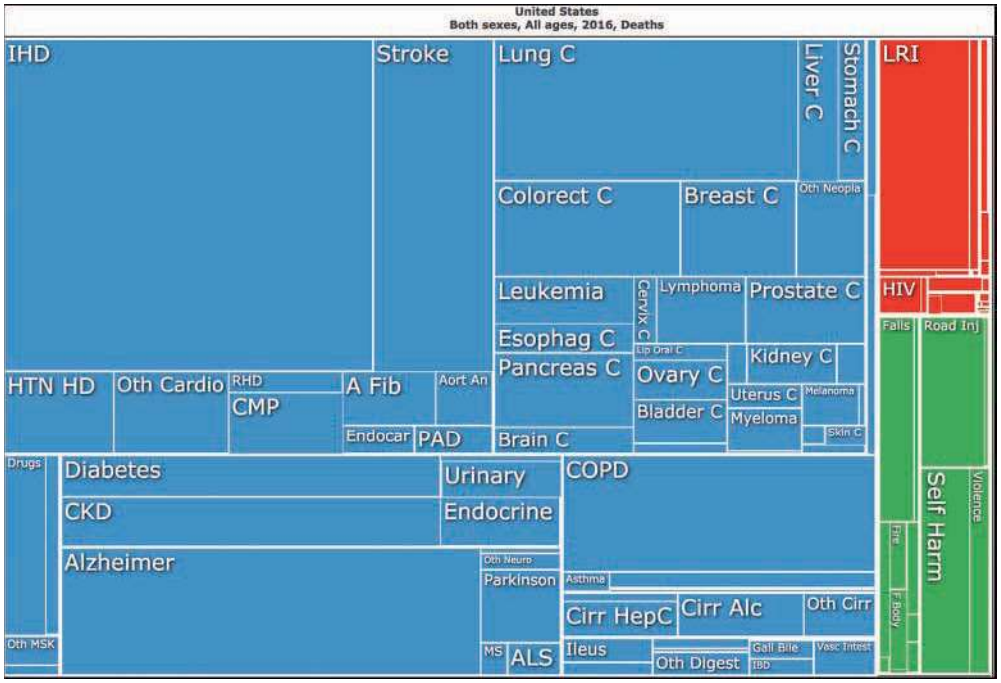

Figure 15.14 A treemap of the number of deaths in the United States from each cause. Screenshot from GBD Compare, a visualization tool for the global burden of disease (https://vizhub.healthdata.org/gbd-compare/).

(e.g., lung cancer) is a member of a family of causes (e.g., cancers), which can be further grouped into overarching categories (e.g., non-communicable diseases). Hierarchical data can be visualized using treemaps (Figure 15.14), **circle packing** (Figure 15.15), **sunburst diagrams** (Figure 15.16), or other layouts. Each of these visualizations uses an **area encoding** to represent a numeric value. These shapes (rectangles, circles, or arcs) are organized in a layout that clearly expresses the hierarchy of information.

The benefit of visualizing the hierarchy of a data set, however, is not without its costs. As described in Section 15.3, it is quite difficult to visually decipher and compare values encoded in a treemap (especially with rectangles of different aspect ratios). However, these displays provide a great summary overview of hierarchies, which is an important starting point for visually exploring data.

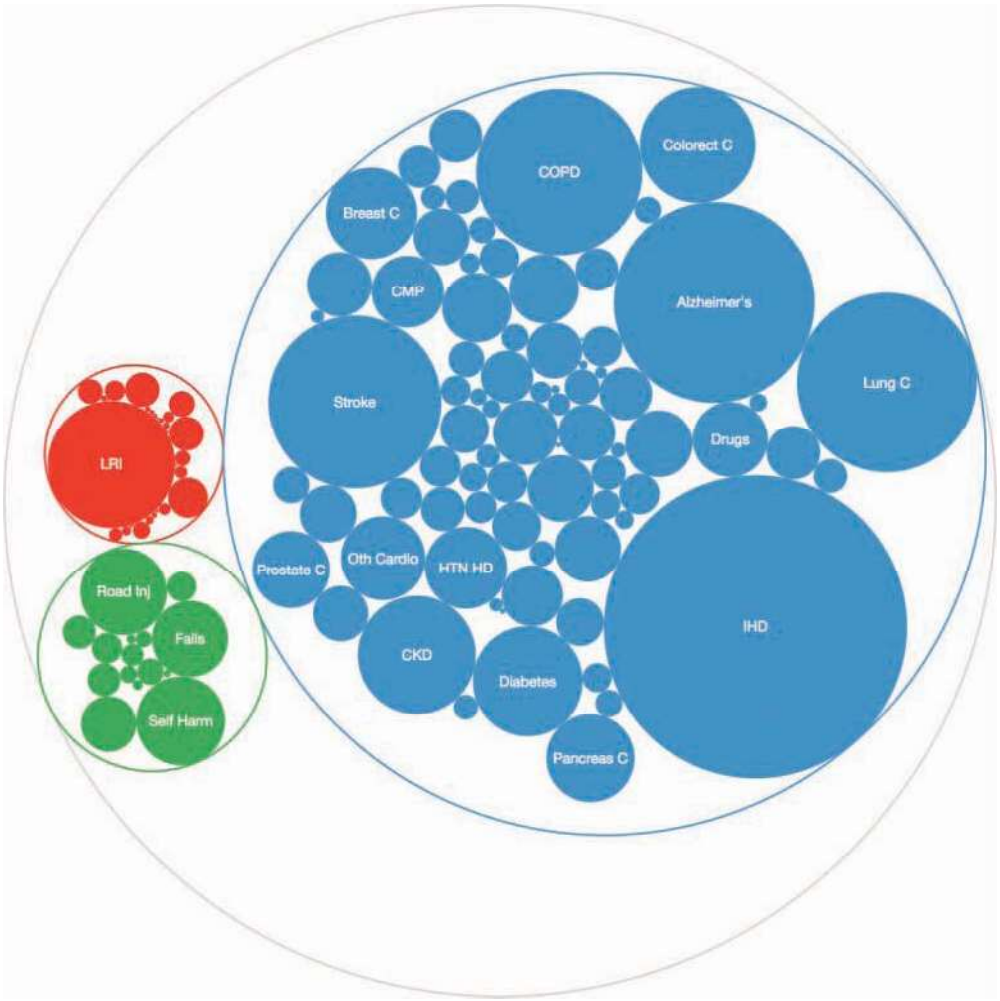

Figure 15.15 A re-creation of the treemap visualization (of disease burden in the United States) using a circle pack layout. Created using the d3.js library https://d3js.org.

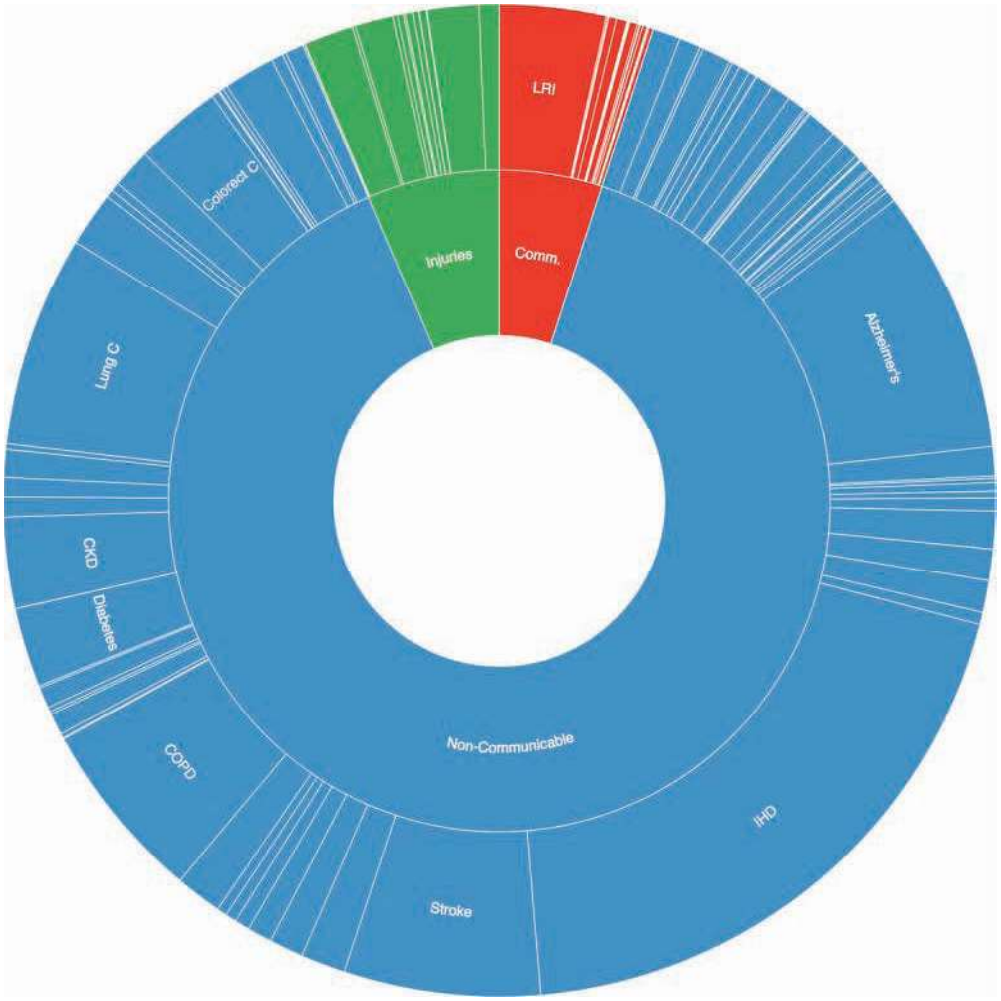

Figure 15.16 A re-creation of the treemap visualization (of disease burden in the United States) using a sunburst diagram. Created using the d3.js library https://d3js.org.

15.3 Choosing Effective Graphical Encodings

While the previously given guidelines for selecting visual layouts based on the data relationship to explore are a good place to start, there are often multiple ways to represent the same data set. Representing data in another format (e.g., visually) is called **encoding** that data. When you encode data, you use a particular "code" such as color or size to represent each value. These visual representations are then visually *de*coded by anyone trying to interpret the underlying values.

Your task is thus to select the encodings that are most accurately decoded by users, answering the question:

> *"What visual form best allows you to exploit the human visual system and available space to accurately display your data values?"*

In designing a visual layout, you should choose the **graphical encodings** that are most accurately visually decoded by your audience. This means that, for every value in your data, your user's interpretation of that value should be as accurate as possible. The accuracy of these perceptions is referred to as the **effectiveness** of a graphical encoding. Academic research[2] measuring the perceptiveness of different visual encodings has established a common set of possible encodings for quantitative information, listed here in order from most effective to least effective:

- **Position**: the horizontal or vertical position of an element along a common scale
- **Length**: the length of a segment, typically used in a stacked bar chart
- **Area**: the area of an element, such as a circle or a rectangle, typically used in a bubble chart (a scatterplot with differently sized markers) or a treemap
- **Angle**: the rotational angle of each marker, typically used in a circular layout like a pie chart
- **Color**: the color of each marker, usually along a continuous color scale
- **Volume**: the volume of a three-dimensional shape, typically used in a 3D bar chart

As an example, consider the very simple data set in Table 15.3. An effective visualization of this data set would enable you to easily distinguish between the values of each group (e.g., between the values 10 and 11). While this identification is simple for a *position* encoding, detecting this 10% difference is very difficult for other encodings. Comparisons between encodings of this data set are shown in Figure 15.17.

Thus when a visualization designer makes a blanket claim like "You should always use a bar chart rather than a pie chart," the designer is really saying, "A bar chart, which uses position encoding along a common scale, is more accurately visually decoded compared to a pie chart (which uses an angle encoding)."

Table 15.3 **A simple data set to demonstrate the perceptiveness of different graphical encodings (shown in Figure 15.17). Users should be able to visually distinguish between these values.**

group	value
a	1
b	10
c	11
d	7
e	8

[2] Most notably, Cleveland, W. S., & McGill, R. (1984). Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association, 79*(387), 531–554. https://doi.org/10.1080/01621459.1984.10478080

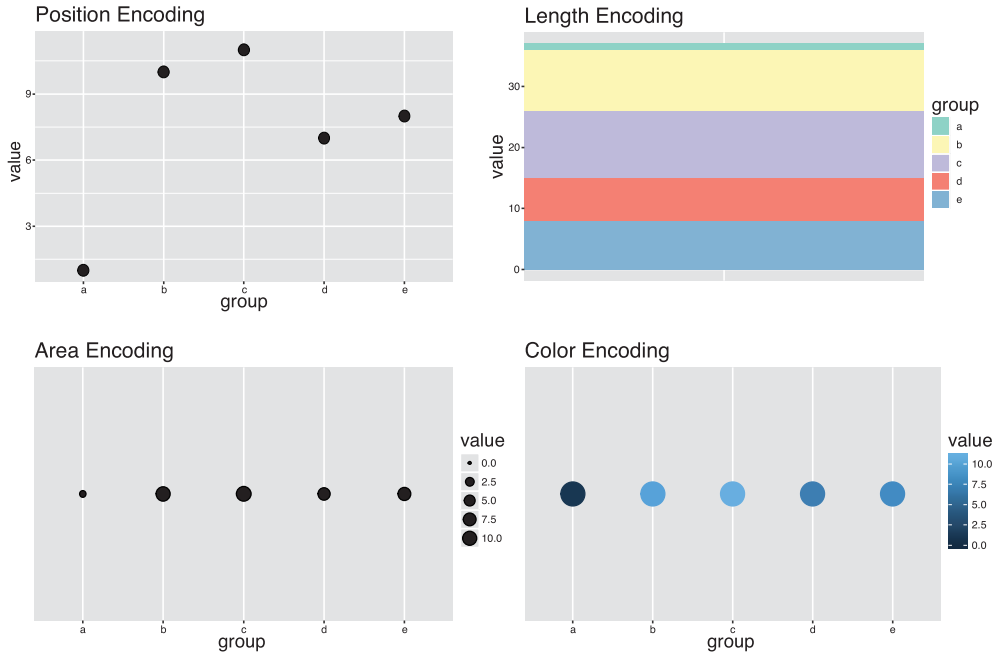

Figure 15.17 Different graphical encodings of the same data. Note the variation in perceptibility of differences between values!

To design your visualization, you should begin by encoding the most important data features with the most accurately decoded visual features (position, then length, then area, and so on). This will provide you with guidance as you compare different chart options and begin to explore more creative layouts.

While these guidelines may feel intuitive, the volume and distribution of your data often make this task more challenging. You may struggle to display all of your data, requiring you to also work to maximize the **expressiveness** of your visualizations (see Section 15.4).

15.3.1 Effective Colors

Color is one of the most prominent visual encodings, so it deserves special consideration. To describe how to use color effectively in visualizations, it is important to understand how color is measured. While there are many different conceptualizations of color spaces, a useful one for visualization is the **hue–saturation–lightness (HSL)** model, which defines a color using three attributes:

- The **hue** of a color, which is likely how you think of describing a color (e.g., "green" or "blue")
- The **saturation** or intensity of a color, which describes how "rich" the color is on a linear scale between gray (0%) and the full display of the hue (100%)

- The **lightness** of the color, which describes how "bright" the color is on a linear scale from black (0%) to white (100%)

This color model can be seen in Figure 15.18, which is an example of an interactive color selector[3] that allows you to manipulate each attribute independently to pick a color. The HSL model provides a good foundation for color selection in data visualization.

When selecting colors for visualization, the data type of your variable should drive your decisions. Depending on the data type (categorical or continuous), the purpose of your encoding will likely be different:

- For categorical variables, a color encoding is used to *distinguish between groups*. Therefore, you should select colors with different hues that are visually distinct and do not imply a rank ordering.
- For continuous variables, a color encoding is used to *estimate values*. Therefore, colors should be picked using a linear *interpolation* between color points (i.e., different lightness values).

Picking colors that most effectively satisfy these goals is trickier than it seems (and beyond the scope of this short section). But as with any other challenge in data science, you can build upon the open source work of other people. One of the most popular tools for picking colors (especially for maps) is Cynthia Brewer's **ColorBrewer**.[4] This tool provides a wonderful set of **color palettes** that differ in hue for categorical data (e.g., "Set3") and in lightness for continuous data (e.g., "Purples"); see Figure 15.19. Moreover, these palettes have been carefully designed to be viewable to people

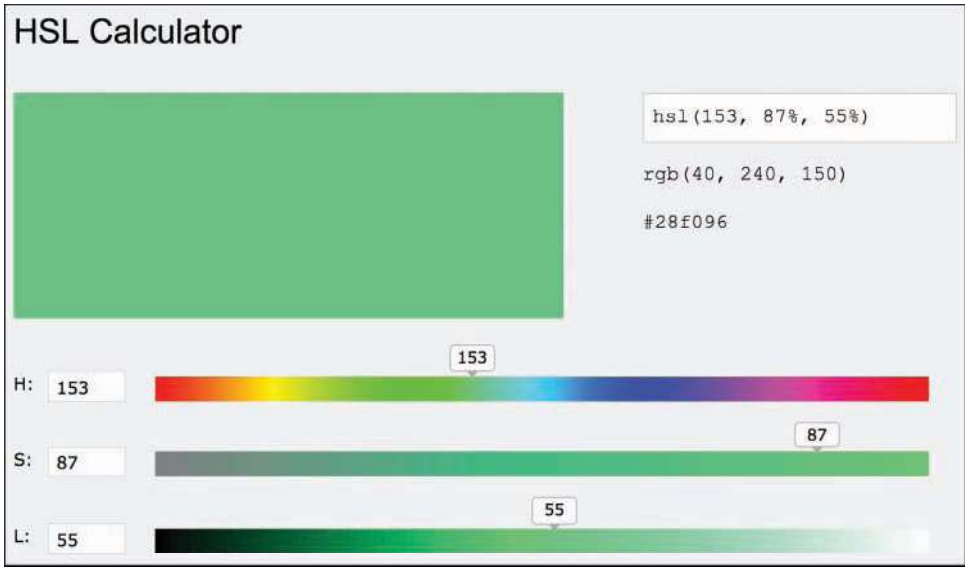

Figure 15.18 An interactive hue–staturation–lightness color picker, from w3schools.

[3] **HSL Calculator** by w3schools: https://www.w3schools.com/colors/colors_hsl.asp
[4] **ColorBrewer:** http://colorbrewer2.org

Figure 15.19 All palettes made available by the `colorbrewer` package in R. Run the `display.brewer.all()` function to see them in RStudio.

with certain forms of color blindness. These palettes are available in R through the `RColorBrewer` package; see Chapter 16 for details on how to use this package as part of your visualization process.

Selecting between different types of color palettes depends on the *semantic* meaning of the data. This choice is illustrated in Figure 15.20, which shows map visualizations of the population of each county in Washington state. The choice between different types of **continuous color scales** depends on the data:

- **Sequential** color scales are often best for displaying continuous values along a linear scale (e.g., for this population data).
- **Diverging** color scales are most appropriate when the divergence from a center value is meaningful (e.g., the midpoint is zero). For example, if you were showing changes in population over time, you could use a diverging scale to show increases in population using one hue, and decreases in population using another hue.
- **Multi-hue** color scales afford an increase in contrast between colors by providing a broader color range. While this allows for more precise interpretations than a (single hue) sequential color scale, the user may misinterpret or misjudge the differences in hue if the scale is not carefully chosen.

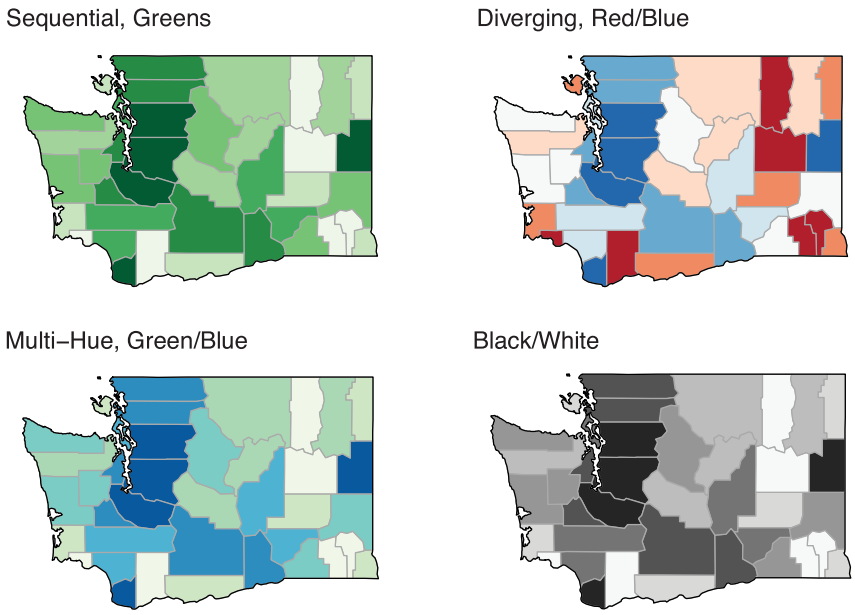

Figure 15.20 Population data in Washington represented with four ColorBrewer scales. The sequential and black/white scales accurately represent continuous data, while the diverging scale (inappropriately) implies divergence from a meaningful center point. Colors in the multi-hue scale may be misinterpreted as having different meanings.

- **Black and white** color scales are equivalent to sequential color scales (just with a hue of gray!) and may be required for your medium (e.g., when printing in a book or newspaper).

Overall, the choice of color will depend on the data. Your goal is to make sure that the color scale chosen enables the viewer to most effectively distinguish between the data's values and meanings.

15.3.2 Leveraging Preattentive Attributes

You often want to draw attention to particular observations in your visualizations. This can help you drive the viewer's focus toward specific instances that best convey the information or intended interpretation (to "tell a story" about the data). The most effective way to do this is to leverage the natural tendencies of the human visual processing system to direct a user's attention. This class of natural tendencies is referred to as **preattentive processing**: the cognitive work that your brain does without you deliberately paying attention to something. More specifically, these are the "[perceptual] tasks that can be performed on large multi-element displays in less than 200 to 250 milliseconds."[5] As detailed by Colin Ware,[6] the visual processing system will automatically process certain stimuli without any conscious effort. As a visualization designer, you want to take advantage of visual attributes that are processed preattentively, making your graphics as rapidly understood as possible.

As an example, consider Figure 15.21, in which you are able to count the occurrences of the number 3 at dramatically different speeds in each graphic. This is possible because your brain naturally identifies elements of the same color (more specifically, opacity) without having to put forth any effort. This technique can be used to drive focus in a visualization, thereby helping people quickly identify pertinent information.

Figure 15.21 Because opacity is processed preattentively, the visual processing system identifies elements of interest (the number 3) without effort in the right graphic, but not in the left graphic.

[5] Healey, C. G., & Enns, J. T. (2012). Attention and visual memory in visualization and computer graphics. *IEEE Transactions on Visualization and Computer Graphics, 18*(7), 1170–1188. https://doi.org/10.1109/TVCG.2011.127. Also at: https://www.csc2.ncsu.edu/faculty/healey/PP/

[6] Ware, C. (2012). *Information visualization: Perception for design*. Philadelphia, PA: Elsevier.

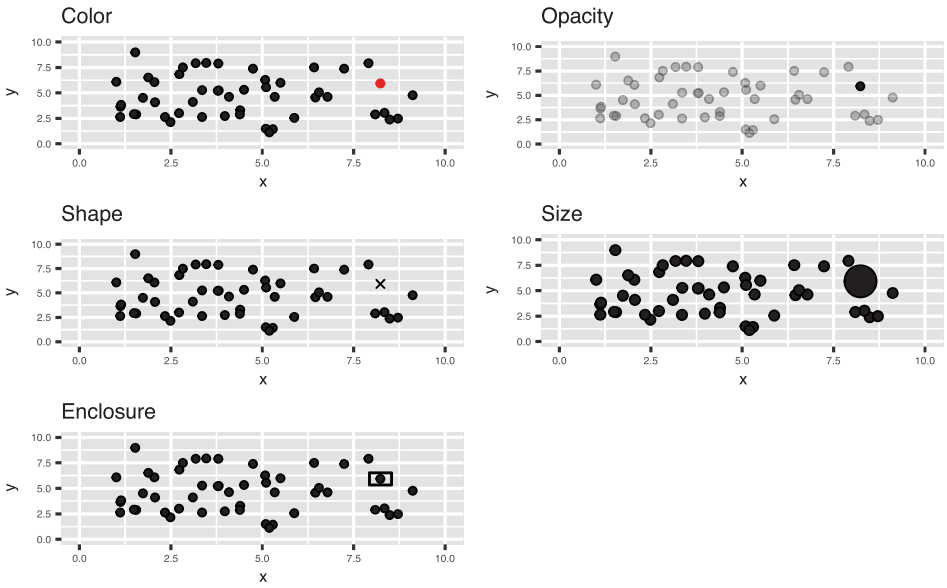

Figure 15.22 Driving focus with preattentive attributes. The selected point is clear in each graph, but especially easy to detect using color.

In addition to color, you can use other visual attributes that help viewers preattentively distinguish observations from those around them, as illustrated in Figure 15.22. Notice how quickly you can identify the "selected" point—though this identification happens more rapidly with some encodings (i.e., color) than with others!

As you can see, color and opacity are two of the most powerful ways to grab attention. However, you may find that you are already using color and opacity to encode a feature of your data, and thus can't also use these encodings to draw attention to particular observations. In that case, you can consider the remaining options (e.g., shape, size, enclosure) to direct attention to a specific set of observations.

15.4 Expressive Data Displays

The other principle you should use to guide your visualization design is to choose layouts that allow you to *express* as much data as possible. This goal was originally articulated as **Mackinlay's Expressiveness Criteria**[7] (clarifications added):

> A set of facts [data] is expressible in a language [visual layout] if that language contains a sentence [form] that
>
> 1. encodes all the facts in the set,
> 2. encodes only the facts in the set.

[7] Mackinlay, J. (1986). Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics, 5*(2), 110–141. https://doi.org/10.1145/22949.22950. Restatement by Jeffrey Heer.

The prompt of this **expressiveness** aim is to devise visualizations that express all of (and only) the data in your data set. The most common barrier to expressiveness is occlusion (overlapping data points). As an example, consider Figure 15.23, which visualizes the distribution of the number of deaths attributable to different causes in the United States. This chart uses the most visually perceptive visual encoding (position), but fails to express all of the data due to the overlap in values.

There are two common approaches to address the failure of expressiveness caused by overlapping data points:

1. Adjust the *opacity* of each marker to reveal overlapping data.
2. Break the data into different groupings or facets to alleviate the overlap (by showing only a subset of the data at a time).

These approaches are both implemented in combination in Figure 15.24.

Alternatively, you could consider changing the data that you are visualizing by **aggregating** it in an appropriate way. For example, you could group your data by values that have similar number of deaths (putting each into a "bin"), and then use a position encoding to show the number of observations per bin. The result of this is the commonly used layout known as a histogram, as shown in Figure 15.25. While this visualization does communicate summary information to your audience, it is unable to express each individual observation in the data (which would communicate more information through the chart).

At times, the expressiveness and effectiveness principles are at odds with one another. In an attempt to maximize expressiveness (and minimize the overlap of your symbols), you may have to choose a less effective encoding. While there are multiple strategies for this—for example, breaking

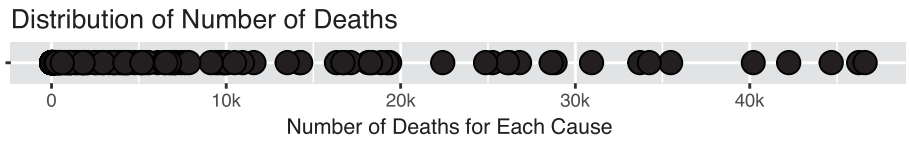

Figure 15.23 Position encoding of the number of deaths from each cause in the United States. Notice how the overlapping points (occlusion) prevent this layout from expressing all of the data. Some outliers have been removed for demonstration.

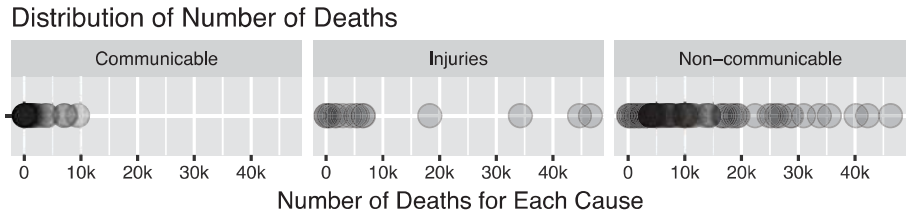

Figure 15.24 Position encoding of the number of deaths from each cause in the United States, faceted by the category of each cause. The use of a lower opacity in conjunction with the faceting enhances the expressiveness of the plots. Some outliers have been removed for demonstration.

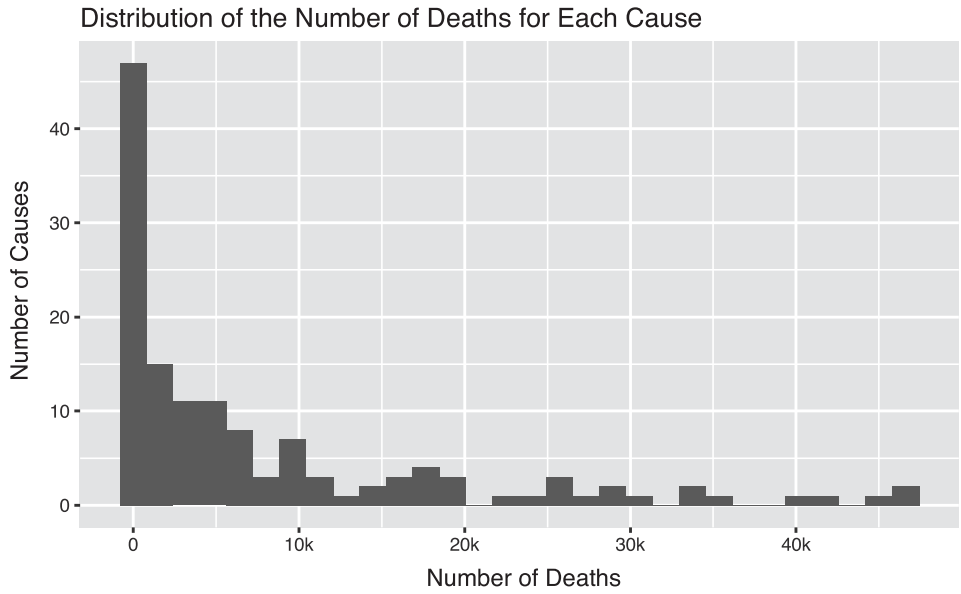

Figure 15.25 Histogram of the number of deaths attributable to each cause.

the data into multiple plots, aggregating the data, and changing the opacity of your symbols—the most appropriate choice will depend on the distribution and volume of your data, as well as the specific question you wish to answer.

15.5 Enhancing Aesthetics

Following the principles described in this chapter will go a long way in helping you devise informative visualizations. But to gain trust and attention from your potential audiences you will also want to spend time investing in the **aesthetics** (i.e., beauty) of your graphics.

> **Tip**: Making beautiful charts is a practice of removing clutter, *not* adding design.

One of the most renowned data visualization theorists, Edward Tufte, frames this idea in terms of the **data–ink ratio**.[8] Tufte argues that in every chart, you should maximize the ink dedicated to displaying the data (and in turn, minimize the non-data ink). This can translate to a number of actions:

- **Remove unnecessary encodings**. For example, if you have a bar chart, the bars should have different colors only if that information isn't otherwise expressed.

[8] Tufte, E. R. (1986). *The visual display of quantitative information. Cheshire*, CT: Graphics Press.

- **Avoid visual effects.** Any 3D effects, unnecessary shading, or other distracting formatting should be avoided. Tufte refers to this as "chart junk."
- **Include chart and axis labels**. Provide a title for your chart, as well as meaningful labels for your axes.
- **Lighten legends/labels**. Reduce the size or opacity of axis labels. Avoid using striking colors.

It's easy to look at a chart such as the chart on the left side of Figure 15.26 and claim that it *looks unpleasant*. However, describing *why* it looks distracting and *how* to improve it can be more challenging. If you follow the tips in this section and strive for simplicity, you can remove unnecessary elements and drive focus to the data (as shown on the right-hand side of Figure 15.26).

Luckily, many of these optimal choices are built into the default R packages for visualization, or are otherwise readily implemented. That being said, you may have to adhere to the aesthetics of your organization (or your own preferences!), so choosing an easily configurable visualization package (such as `ggplot2`, described in Chapter 16) is crucial.

As you begin to design and build visualizations, remember the following guidelines:

1. Dedicate each visualization to answering a **specific** question of interest.
2. Select a visual layout based on your data type.
3. Choose optimal graphical encodings based on how well they are visually decoded.
4. Ensure that your layout is able to express your data.
5. Enhance the aesthetics by removing visual effects, and by including clear labels.

These guidelines will be a helpful start, and don't forget that visualizations are about *insights*, not pictures.

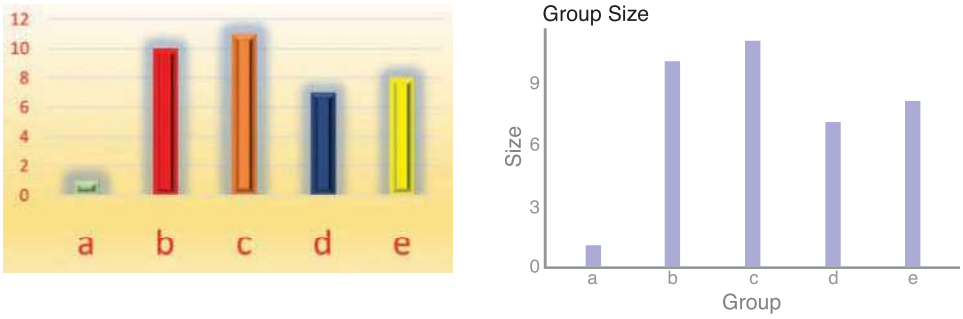

Figure 15.26 Removing distracting and uninformative visual features (left) and adding informative labels to create a cleaner chart (right).

16

Creating Visualizations with `ggplot2`

The ability to create visualizations (graphical representations) of data is a key step in being able to communicate information and findings to others. In this chapter, you will learn to use the **`ggplot2`**[1] package to declaratively make beautiful visual representations of your data.

Although `R` does provide built-in plotting functions, the `ggplot2` package is built on the premise of the *Grammar of Graphics* (similar to how `dplyr` implements a *Grammar of Data Manipulation*; indeed, both packages were originally developed by the same person). This makes the package particularly effective for describing how visualizations should represent data, and has turned it into the preeminent plotting package in `R`. Learning to use this package will allow you to make nearly any kind of (static) data visualization, customized to your exact specifications.

16.1 A Grammar of Graphics

Just as the grammar of language helps you construct meaningful sentences out of words, the **Grammar of Graphics** helps you construct graphical figures out of different visual elements. This grammar provides a way to talk about parts of a visual plot: all the circles, lines, arrows, and text that are combined into a diagram for visualizing data. Originally developed by Leland Wilkinson, the *Grammar of Graphics* was adapted by Hadley Wickham[2] to describe the *components* of a plot:

- The **data** being plotted
- The **geometric objects** (e.g., circles, lines) that appear on the plot
- The **aesthetics** (appearance) of the geometric objects, and the *mappings* from variables in the data to those aesthetics
- A **position adjustment** for placing elements on the plot so they don't overlap
- A **scale** (e.g., a range of values) for each aesthetic mapping used

[1] **`ggplot2`**: http://ggplot2.tidyverse.org

[2] Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics, 19*(1), 3–28. https://doi.org/10.1198/jcgs.2009.07098. Also at http://vita.had.co.nz/papers/layered-grammar.pdf

- A **coordinate system** used to organize the geometric objects
- The **facets** or groups of data shown in different plots

ggplot2 further organizes these components into **layers**, where each layer displays a single type of (highly configurable) *geometric object*. Following this grammar, you can think of each plot as a set of layers of images, where each image's appearance is based on some aspect of the data set.

Collectively, this grammar enables you to discuss what plots look like using a standard set of vocabulary. And like with dplyr and the *Grammar of Data Manipulation*, ggplot2 uses this grammar directly to declare plots, allowing you to more easily create specific visual images and tell stories[3] about your data.

16.2 Basic Plotting with ggplot2

The ggplot2 package provides a set of *functions* that mirror the *Grammar of Graphics*, enabling you to efficaciously specify what you want a plot to look like (e.g., what data, geometric objects, aesthetics, scales, and so on you want it to have).

ggplot2 is yet another external package (like dplyr, httr, etc.), so you will need to install and load the package to use it:

```
install.packages("ggplot2") # once per machine
library("ggplot2")          # in each relevant script
```

This will make all of the plotting functions you will need available. As a reminder, plots will be rendered in the lower-right quadrant of RStudio, as shown in Figure 16.1.

> **Fun Fact**: Similar to dplyr, the ggplot2 package also comes with a number of built-in data sets. This chapter will use the provided midwest data set as an example, described below.

This section uses the midwest data set that is included as part of the ggplot2 package—a subset of the data is shown in Figure 16.2. The data set contains information on each of 437 counties in 5 states in the midwestern United States (specifically, Illinois, Indiana, Michigan, Ohio, and Wisconsin). For each county, there are 28 features that describe the demographics of the county, including racial composition, poverty levels, and education rates. To learn more about the data, you can consult the documentation (`?midwest`).

To create a plot using the ggplot2 package, you call the **`ggplot()`** function, specifying as an argument the data that you wish to plot (i.e., `ggplot(data = SOME_DATA_FRAME)`). This will create a blank canvas upon which you can *layer* different visual markers. Each layer contains a specific *geometry*—think points, lines, and so on—that will be drawn on the canvas. For example, in Figure 16.3 (created using the following code), you can add a layer of points to assess the association between the percentage of people with a college education and the percentage of adults living in poverty in counties in the Midwest.

[3] Sander, L. (2016). Telling stories with data using the grammar of graphics. *Code Words*, 6. https://codewords.recurse.com/issues/six/telling-stories-with-data-using-the-grammar-of-graphics

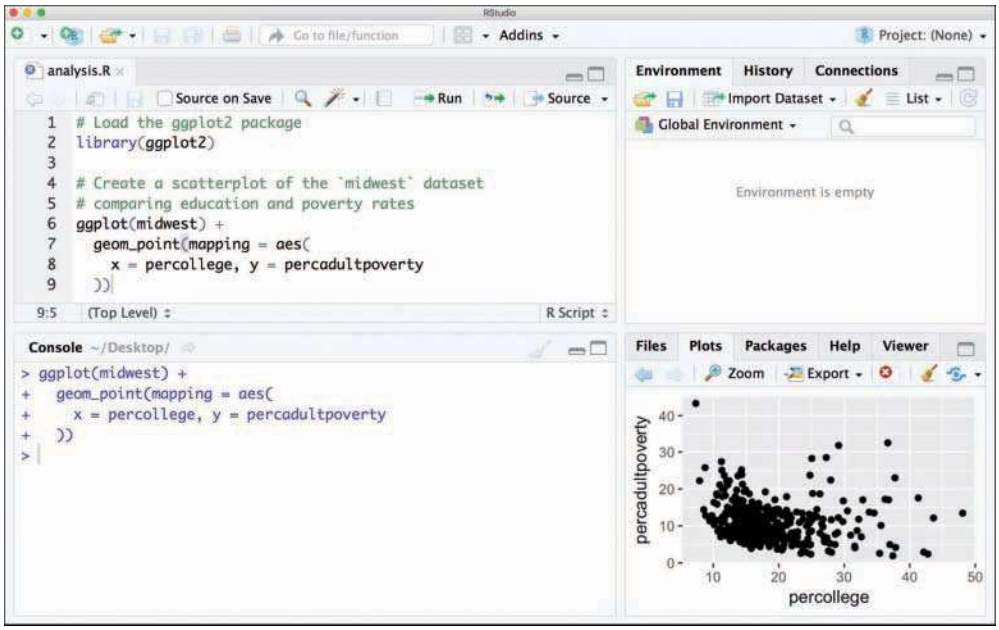

Figure 16.1 `ggplot2` graphics will render in the lower-right quadrant of the RStudio window.

	PID	county	state	area	poptotal	popdensity	popwhite	popblack
1	561	ADAMS	IL	0.052	66090	1270.9615	63917	1702
2	562	ALEXANDER	IL	0.014	10626	759.0000	7054	3496
3	563	BOND	IL	0.022	14991	681.4091	14477	429
4	564	BOONE	IL	0.017	30806	1812.1176	29344	127
5	565	BROWN	IL	0.018	5836	324.2222	5264	547
6	566	BUREAU	IL	0.050	35688	713.7600	35157	50
7	567	CALHOUN	IL	0.017	5322	313.0588	5298	1
8	568	CARROLL	IL	0.027	16805	622.4074	16519	111
9	569	CASS	IL	0.024	13437	559.8750	13384	16
10	570	CHAMPAIGN	IL	0.058	173025	2983.1897	146506	16559
11	571	CHRISTIAN	IL	0.042	34418	819.4762	34176	82
12	572	CLARK	IL	0.030	15921	530.7000	15842	10
13	573	CLAY	IL	0.028	14460	516.4286	14403	4

Figure 16.2 A subset of the `midwest` data set, which captures demographic information on 5 midwestern states. The data set is included as part of the `ggplot2` package and used throughout this chapter.

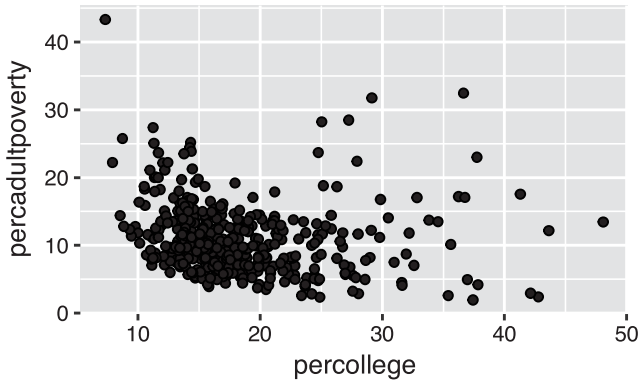

Figure 16.3 A basic use of `ggplot`: comparing the college education rates to adult poverty rates in Midwestern counties by adding a layer of points (thereby creating a scatterplot).

```
# Plot the `midwest` data set, with college education rate on the x-axis and
# percentage of adult poverty on the y-axis
ggplot(data = midwest) +
  geom_point(mapping = aes(x = percollege, y = percadultpoverty))
```

The code for creating a ggplot2 plot involves a few steps:

- The `ggplot()` function is passed the data frame to plot as the named `data` argument (it can also be passed as the first positional argument). Calling this function creates the blank canvas on which the visualization will be created.
- You specify the type of geometric object (sometimes referred to as a "geom") to draw by calling one of the many **`geom_` functions**[4]—in this case, `geom_point()`. Functions to render a layer of geometric objects all share a common prefix (`geom_`), followed by the name of the kind of geometry you wish to create. For example, `geom_point()` will create a layer with "point" (dot) elements as the geometry. There are a large number of these functions; more details are provided in Section 16.2.1.
- In each `geom_` function, you must specify the **aesthetic mappings**, which specify how data from the data frame will be mapped to the visual aspects of the geometry. These mappings are defined using the **`aes()`** (*aesthetic*) function. The **`aes()`** function takes a set of named arguments (like a list), where the argument name is the visual property to map *to*, and the argument value is the data feature (i.e., the column in the data frame) to map *from*. The value returned by the `aes()` function is passed to the named `mapping` argument (or passed as the first positional argument).

[4] **Layer: geoms** function reference: http://ggplot2.tidyverse.org/reference/index.html#section-layer-geoms

> **Caution**: The `aes()` function uses *non-standard evaluation* similar to `dplyr`, so you don't need to put the data frame column names in quotes. This can cause issues if the name of the column you wish to plot is stored as a string in a variable (e.g., `plot_var <- "COLUMN_NAME"`). To handle this situation, you can use the **`aes_string()`** function instead and specify the column names as string values or variables.

- You add layers of geometric objects to the plot by using the addition (**+**) operator.

Thus, you can create a basic plot by specifying a data set, an appropriate `geometry`, and a set of aesthetic mappings.

> **Tip**: The **`ggplot2`** package includes a **`qplot()`** function[a] for creating "quick plots." This function is a convenient shortcut for making simple, "default"-like plots. While this is a nice starting point, the strength of `ggplot2` lies in its *customizability*, so read on!
>
> [a] http://www.statmethods.net/advgraphs/ggplot2.html

16.2.1 Specifying Geometries

The most obvious distinction between plots is the geometric objects that they include. `ggplot2` supports the rendering of a variety of geometries, each created using the appropriate `geom_` function. These functions include, but are not limited to, the following:

- **`geom_point()`** for drawing individual points (e.g., for a scatterplot)
- **`geom_line()`** for drawing lines (e.g., for a line chart)
- **`geom_smooth()`** for drawing smoothed lines (e.g., for simple trends or approximations)
- **`geom_col()`** for drawing columns (e.g., for a bar chart)
- **`geom_polygon()`** for drawing arbitrary shapes (e.g., for drawing an area in a coordinate plane)

Each of these `geom_` functions requires as an argument a set of aesthetic mappings (defined using the `aes()` function, described in Section 16.2.2), though the specific *visual properties* that the data will map *to* will vary. For example, you can map a data feature to the `shape` of a `geom_point()` (e.g., if the points should be circles or squares), or you can map a feature to the `linetype` of a `geom_line()` (e.g., if it is solid or dotted), but not vice versa.

Since graphics are two-dimensional representations of data, almost all `geom_` functions *require* an `x` and `y` mapping. For example, in Figure 16.4, the bar chart of the number of counties per state (left) is built using the `geom_col()` geometry, while the hexagonal aggregation of the scatterplot from Figure 16.3 (right) is built using the `geom_hex()` function.

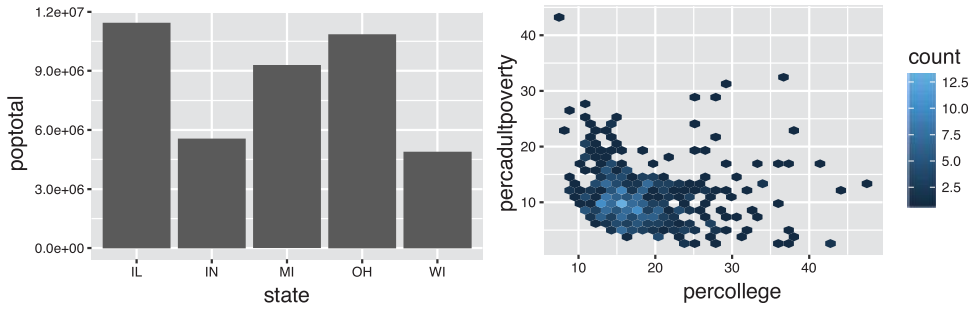

Figure 16.4 Plots with column geometry (left) and binned hexagons (right). The rectangles in the column geometry represent separate observations (counties) that have been automatically stacked on top of each other; see Section 16.3.1 for details.

```
# A bar chart of the total population of each state
# The `state` is mapped to the x-axis, and the `poptotal` is mapped
# to the y-axis
ggplot(data = midwest) +
  geom_col(mapping = aes(x = state, y = poptotal))

# A hexagonal aggregation that counts the co-occurrence of college
# education rate and percentage of adult poverty
ggplot(data = midwest) +
  geom_hex(mapping = aes(x = percollege, y = percadultpoverty))
```

What makes this really powerful is that you can add *multiple geometries* to a plot. This allows you to create complex graphics showing multiple aspects of your data, as in Figure 16.5.

```
# A plot with both points and a smoothed line
ggplot(data = midwest) +
  geom_point(mapping = aes(x = percollege, y = percadultpoverty)) +
  geom_smooth(mapping = aes(x = percollege, y = percadultpoverty))
```

While the `geom_point()` and `geom_smooth()` layers in this code both use the same aesthetic mappings, there's no reason you couldn't assign different aesthetic mappings to each geometry. Note that if the layers *do* share some aesthetic mappings, you can specify those as an argument to the `ggplot()` function as follows:

```
# A plot with both points and a smoothed line, sharing aesthetic mappings
ggplot(data = midwest, mapping = aes(x = percollege, y = percadultpoverty)) +
  geom_point() + # uses the default x and y mappings
  geom_smooth() + # uses the default x and y mappings
  geom_point(mapping = aes(y = percchildbelowpovert)) # uses own y mapping
```

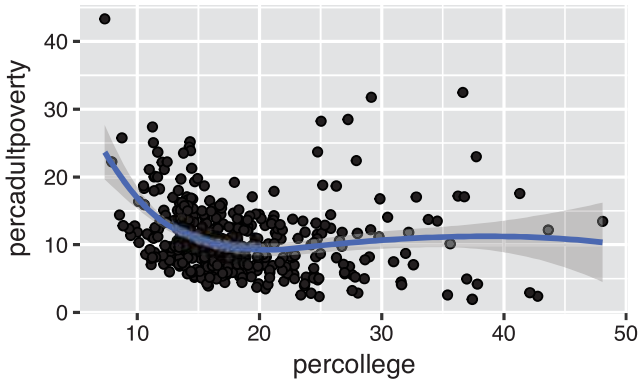

Figure 16.5 A plot comparing the adult poverty rate and the college education rate using multiple geometries. Each layer is added with a different `ggplot2` function: `geom_point()` for points, and `geom_smooth()` for the smoothed line.

Each geometry will use the data and individual aesthetics specified in the `ggplot()` function unless they are overridden by individual specifications.

> **Going Further**: Some `geom_` functions also perform a *statistical transformation* on the data, aggregating the data (e.g., counting the number of observations) before mapping that data to an aesthetic. While you can do many of these transformations using the `dplyr` functions `group_by()` and `summarize()`, a statistical transformation allows you to apply some aggregations purely to adjust the data's presentation, without needing to modify the data itself. You can find more information in the documentation.[a]
>
> [a] http://ggplot2.tidyverse.org/reference/index.html#section-layer-stats

16.2.2 Aesthetic Mappings

The aesthetic mappings take properties of the data and use them to influence **visual channels** (graphical encodings), such as position, color, size, or shape. Each visual channel therefore encodes a feature of the data and can be used to express that data. Aesthetic mappings are used for visual features that should be driven by data values, rather than set for all geometric elements. For example, if you want to use a color encoding to express the values in a column, you would use an aesthetic mapping. In contrast, if you want the color of all points to be the same (e.g., blue), you *would not* use an aesthetic mapping (because the color has nothing to do with your data).

The data-driven aesthetics for a plot are specified using the `aes()` function and passed into a particular `geom_` function layer. For example, if you want to know which *state* each county is in, you can add a mapping from the `state` feature of each row to the *color* channel. `ggplot2` will even create a legend for you automatically (as in Figure 16.6)! Note that using the `aes()` function will cause the visual channel to be based on the data specified in the argument.

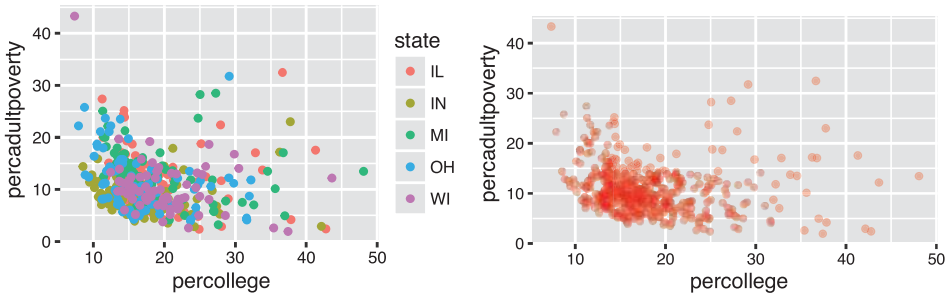

Figure 16.6 Different approaches for choosing color when comparing the adult poverty rate and the college education rate. The left uses a data-driven approach, in which each observation's `state` column is used to set the color (an aesthetic mapping), while the right sets a constant color for all observations. Code is below.

Conversely, if you wish to apply a visual property to an entire geometry, you can set that property on the geometry by passing it as an argument to the `geom_` function, outside of the `aes()` call, as shown in the following code. Figure 16.6 shows both approaches: driving color with the aesthetic (left) and choosing constant styles for each point (right).

```
# Change the color of each point based on the state it is in
ggplot(data = midwest) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = state)
  )

# Set a consistent color ("red") for all points -- not driven by data
ggplot(data = midwest) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty),
    color = "red",
    alpha = .3
  )
```

16.3 Complex Layouts and Customization

Building on these basics, you can use `ggplot2` to create almost any kind of plot you may want. In addition to specifying the geometry and aesthetics, you can further customize plots by using functions that follow from the *Grammar of Graphics*.

16.3.1 Position Adjustments

The plot using `geom_col()` in Figure 16.4 *stacked* all of the observations (rows) per state into a single column. This stacking is the default **position adjustment** for the geometry, which specifies a "rule" as to how different components should be positioned relative to each other to make sure they don't overlap. This positional adjustment can be made more apparent if you map a different

variable to the color encoding (using the `fill` aesthetic). In Figure 16.7 you can see the racial breakdown for the population in each state by adding a `fill` to the column geometry:

```
# Load the `dplyr` and `tidyr` libraries for data manipulation
library("dplyr")
library("tidyr")

# Wrangle the data using `tidyr` and `dplyr` -- a common step!
# Select the columns for racial population totals, then
# `gather()` those column values into `race` and `population` columns
state_race_long <- midwest %>%
  select(state, popwhite, popblack, popamerindian, popasian, popother) %>%
  gather(key = race, value = population, -state) # all columns except `state`

# Create a stacked bar chart of the number of people in each state
# Fill the bars using different colors to show racial composition
ggplot(state_race_long) +
  geom_col(mapping = aes(x = state, y = population, fill = race))
```

Remember: You will need to use your `dplyr` and `tidyr` skills to wrangle your data frames into the proper orientation for plotting. Being confident in those skills will make using the `ggplot2` library a relatively straightforward process; the hard part is getting your data in the desired shape.

Tip: Use the `fill` aesthetic when coloring in bars or other area shapes (that is, specifying what color to "fill" the area). The `color` aesthetic is instead used for the outline (stroke) of the shapes.

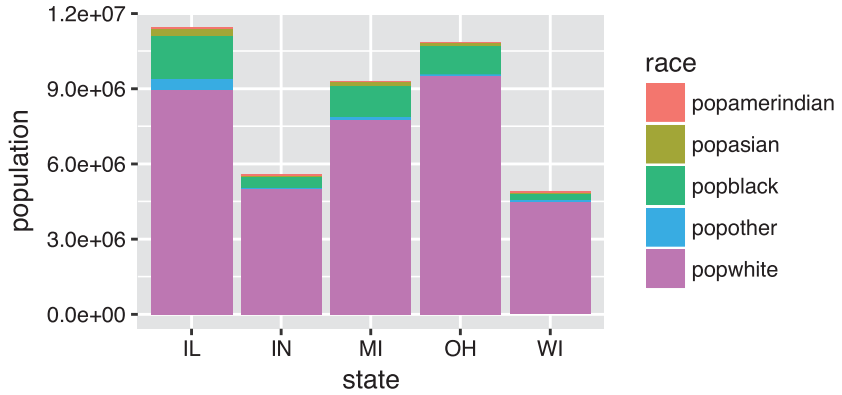

Figure 16.7 A stacked bar chart of the number of people in each state (by race). Colors are added by setting a `fill` aesthetic based on the `race` column.

By default, ggplot will adjust the position of each rectangle by stacking the "columns" for each county. The plot thus shows all of the elements instead of causing them to overlap. However, if you wish to specify a different position adjustment, you can use the **position** argument. For example, to see the relative composition (e.g., percentage) of people by race in each state, you can use a "fill" position (to *fill* each bar to 100%). To see the relative measures within each state side by side, you can use a "dodge" position. To explicitly achieve the default behavior, you can use the "identity" position. The first two options are shown in Figure 16.8.

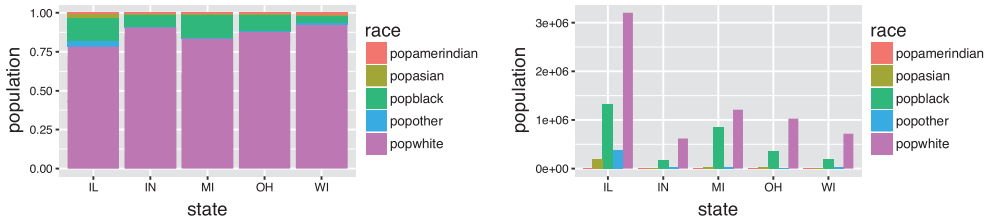

Figure 16.8 Bar charts of state population by race, shown with different position adjustments: filled (left) and dodged (right).

```
# Create a percentage (filled) column of the population (by race) in each state
ggplot(state_race_long) +
  geom_col(
    mapping = aes(x = state, y = population, fill = race), position = "fill"
  )

# Create a grouped (dodged) column of the number of people (by race) in each state
ggplot(state_race_long) +
  geom_col(
    mapping = aes(x = state, y = population, fill = race), position = "dodge"
  )
```

16.3.2 Styling with Scales

Whenever you specify an aesthetic mapping, ggplot2 uses a particular **scale** to determine the *range of values* that the data encoding should be mapped *to*. Thus, when you specify a plot such as:

```
# Plot the `midwest` data set, with college education rate on the x-axis and
# percentage of adult poverty on the y-axis. Color by state.
ggplot(data = midwest) +
  geom_point(mapping = aes(x = percollege, y = percadultpoverty, color = state))
```

ggplot2 automatically adds a scale for each mapping to the plot:

```
# Plot the `midwest` data set, with college education rate and
# percentage of adult poverty. Explicitly set the scales.
ggplot(data = midwest) +
  geom_point(mapping = aes(x = percollege, y = percadultpoverty, color = state)) +
  scale_x_continuous() + # explicitly set a continuous scale for the x-axis
  scale_y_continuous() + # explicitly set a continuous scale for the y-axis
  scale_color_discrete() # explicitly set a discrete scale for the color aesthetic
```

Each scale can be represented by a function named in the following format: `scale_`, followed by the name of the aesthetic property (e.g., `x` or `color`), followed by an `_` and the type of the scale (e.g., `continuous` or `discrete`). A `continuous` scale will handle values such as numeric data (where there is a *continuous set* of numbers), whereas a `discrete` scale will handle values such as colors (since there is a small *discrete* list of distinct colors). Notice also that scales are added to a plot using the `+` operator, similar to a `geom` layer.

While the default scales will often suffice for your plots, it is possible to explicitly add different scales to replace the defaults. For example, you can use a scale to change the direction of an axis (`scale_x_reverse()`), or plot the data on a *logarithmic scale* (`scale_x_log10()`). You can also use scales to specify the range of values on an axis by passing in a `limits` argument. Explicit limits are useful for making sure that multiple graphs share scales or formats, as well as for customizing the appearance of your visualizations. For example, the following code imposes the same scale across two plots, as shown in Figure 16.9:

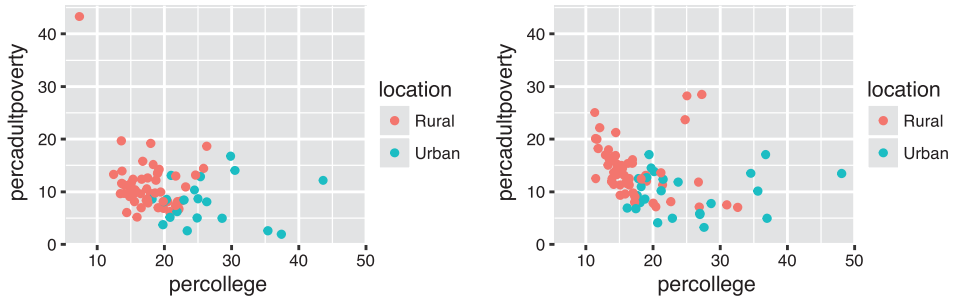

Figure 16.9 Plots of the percent college-educated population versus the percent adult poverty in Wisconsin (left) and Michigan (right). These plots share the same explicit scales (which are not based solely on the plotted data). Notice how it is easy to compare the two data sets to each other because the axes and colors match!

```
# Create a better label for the `inmetro` column
labeled <- midwest %>%
  mutate(location = if_else(inmetro == 0, "Rural", "Urban"))

# Subset data by state
wisconsin_data <- labeled %>% filter(state == "WI")
michigan_data <- labeled %>% filter(state == "MI")

# Define continuous scales based on the entire data set:
# range() produces a (min, max) vector to use as the limits
x_scale <- scale_x_continuous(limits = range(labeled$percollege))
y_scale <- scale_y_continuous(limits = range(labeled$percadultpoverty))

# Define a discrete color scale using the unique set of locations (urban/rural)
color_scale <- scale_color_discrete(limits = unique(labeled$location))
```

```
# Plot the Wisconsin data, explicitly setting the scales
ggplot(data = wisconsin_data) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = location)
  ) +
  x_scale +
  y_scale +
  color_scale

# Plot the Michigan data using the same scales
ggplot(data = michigan_data) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = location)
  ) +
  x_scale +
  y_scale +
  color_scale
```

These scales can also be used to specify the "tick" marks and labels; see the `ggplot2` documentation for details. For further ways of specifying where the data appears on the graph, see Section 16.3.3.

16.3.2.1 Color Scales

One of the most common scales to change is the color scale (i.e., the set of colors used in a plot). While you can use scale functions such as `scale_color_manual()` to specify a specific set of colors for your plot, a more common option is to use one of the predefined ColorBewer[5] palettes (described in Chapter 15, Figure 15.19). These palettes can be specified as a color scale with the `scale_color_brewer()` function, passing the `palette` as a named argument (see the rendered plot in Figure 16.10).

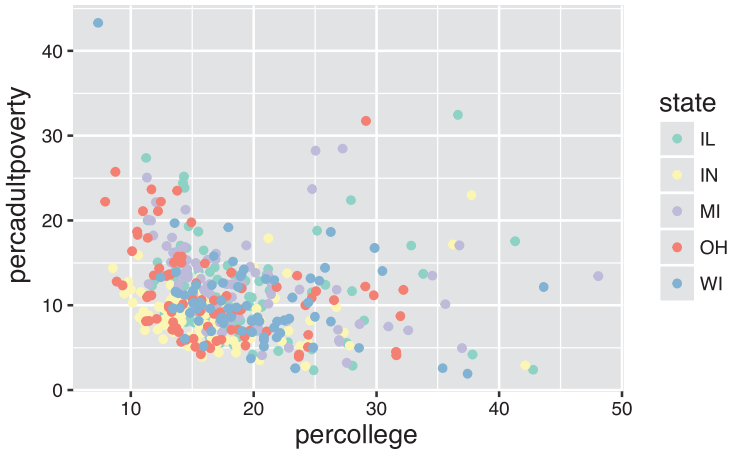

Figure 16.10 A comparison of each county's adult poverty rate and college education rate, using color to show the state each county is in. These colors come from the ColorBrewer `Set3` palette.

[5] ColorBrewer: http://colorbrewer2.org

```
# Change the color of each point based on the state it is in
ggplot(data = midwest) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = state)
  ) +
  scale_color_brewer(palette = "Set3") # use the "Set3" color palette
```

If you instead want to define your own color scheme, you can make use of a variety of `ggplot2` functions. For discrete color scales[6], you can specify a distinct set of colors to map to using a function such as `scale_color_manual()`. For continuous color scales[7], you can specify a range of colors to display using a function such as `scale_color_gradient()`.

16.3.3 Coordinate Systems

It is also possible to specify a plot's **coordinate system**, which is used to organize the geometric objects. As with scales, coordinate systems are specified with functions (whose names all start with `coord_`) and are added to a `ggplot`. You can use several different coordinate systems,[8] including but not limited to the following:

- **`coord_cartesian()`**: The default *Cartesian coordinate* system, where you specify `x` and `y` values—`x` values increase from left to right, and `y` values increase from bottom to top
- **`coord_flip()`**: A Cartesian system with the `x` and `y` flipped
- **`coord_fixed()`**: A Cartesian system with a "fixed" aspect ratio (e.g., 1.78 for "widescreen")
- **`coord_polar()`**: A plot using polar coordinates (i.e., a *pie chart*)
- **`coord_quickmap()`**: A coordinate system that approximates a good aspect ratio for maps. See the documentation for more details

The example in Figure 16.11 uses `coord_flip()` to create a horizontal bar chart (a useful layout for making labels more legible). In the `geom_col()` function's aesthetic mapping, you do not change what you assign to the `x` and `y` variables to make the bars horizontal; instead, you call the `coord_flip()` function to switch the orientation of the graph. The following code (which generates Figure 16.11) also creates a *factor variable* to sort the bars using the variable of interest:

```
# Create a horizontal bar chart of the most populous counties
# Thoughtful use of `tidyr` and `dplyr` is required for wrangling

# Filter down to top 10 most populous counties
top_10 <- midwest %>%
  top_n(10, wt = poptotal) %>%
  unite(county_state, county, state, sep = ", ") %>% # combine state + county
  arrange(poptotal) %>% # sort the data by population
  mutate(location = factor(county_state, county_state)) # set the row order
```

[6] **Gradient color scales** function reference: http://ggplot2.tidyverse.org/reference/scale_gradient.html
[7] **Create your own discrete scale** function reference:http://ggplot2.tidyverse.org/reference/scale_manual.html
[8] **Coordinate systems** function reference: http://ggplot2.tidyverse.org/reference/index.html#section-coordinate-systems

```
# Render a horizontal bar chart of population
ggplot(top_10) +
  geom_col(mapping = aes(x = location, y = poptotal)) +
  coord_flip() # switch the orientation of the x- and y-axes
```

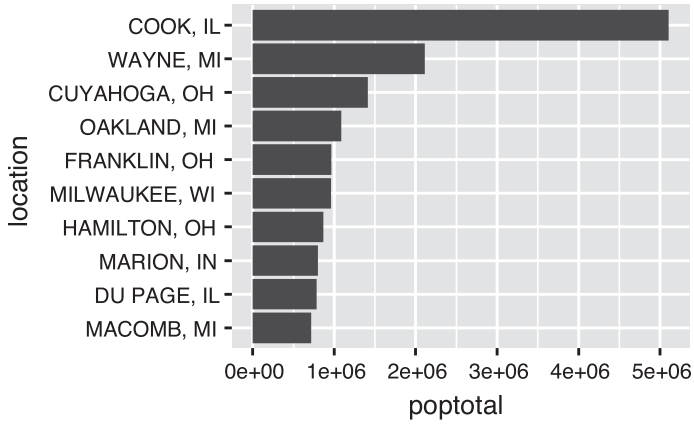

Figure 16.11 A horizontal bar chart of the population in the ten most populous counties. The orientation of the chart is "flipped" by calling the `coord_flip()` function.

In general, the coordinate system is used to specify where in the plot the `x` and `y` axes are placed, while scales are used to determine which values are shown on those axes.

16.3.4 Facets

Facets are ways of grouping a visualization into multiple different pieces (*subplots*). This allows you to view a separate plot for each unique value in a categorical variable. Conceptually, breaking a plot up into facets is similar to using the `group_by()` verb in `dplyr`: it creates the same visualization for each group separately (just as `summarize()` performs the same analysis for each group).

You can construct a plot with multiple facets by using a `facet_` function such as `facet_wrap()`. This function will produce a "row" of subplots, one for each categorical variable (the number of rows can be specified with an additional argument); subplots will "wrap" to the next line if there is not enough space to show them all in a single row. Figure 16.12 demonstrates faceting; as you can see in this plot, using facets is basically an "automatic" way of doing the same kind of grouping performed in Figure 16.9, which shows separate graphs for Wisconsin and Michigan.

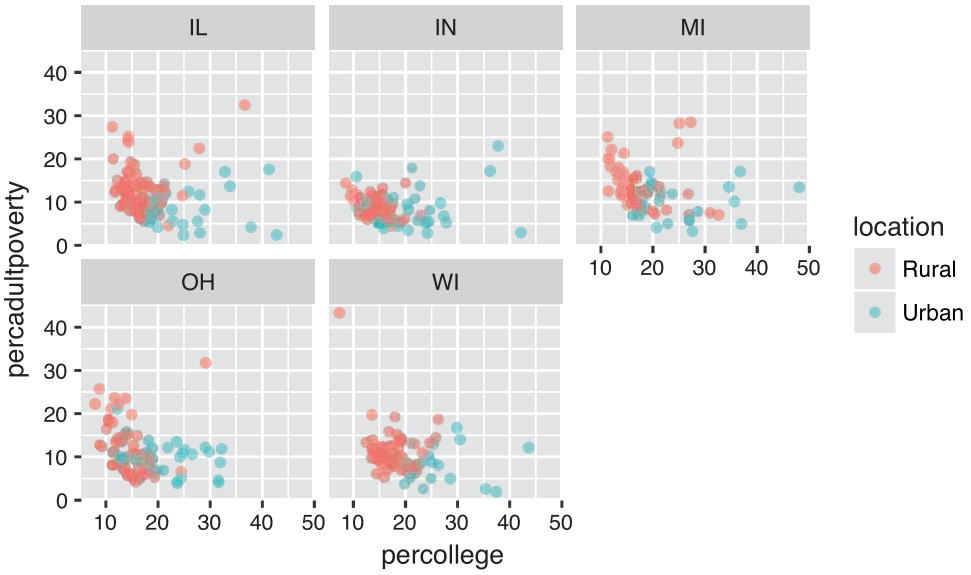

Figure 16.12 A comparison of each county's adult poverty rate and college education rate. A separate plot is created for each state using the `facet_wrap()` function.

```
# Create a better label for the `inmetro` column
labeled <- midwest %>%
  mutate(location = if_else(inmetro == 0, "Rural", "Urban"))

# Create the same chart as Figure 16.9, faceted by state
ggplot(data = labeled) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = location),
    alpha = .6
  ) +
 facet_wrap(~state) # pass the `state` column as a *fomula* to `facet_wrap()`
```

Note that the argument to the `facet_wrap()` function is the column to facet by, with the column name written with a tilde (**~**) in front of it, turning it into a **formula**.[9] A formula is a bit like an equation in mathematics; that is, it represents a set of operations to perform. The tilde can be read "as a function of." The `facet_` functions take formulas as arguments in order to determine how they should group and divide the subplots. In short, with `facet_wrap()` you need to put a ~ in front of the feature name you want to "group" by. See the official `ggplot2` documentation[10] for `facet_` functions for more details and examples.

[9] **Formula** documentation: https://www.rdocumentation.org/packages/stats/versions/3.4.3/topics/formula. See the *Details* in particular.

[10] **`ggplot2` facetting**: https://ggplot2.tidyverse.org/reference/#section-facetting

16.3.5 Labels and Annotations

Textual labels and annotations that more clearly express the meaning of axes, legends, and markers are an important part of making a plot understandable and communicating information. Although not an explicit part of the *Grammar of Graphics* (they would be considered a form of geometry), `ggplot2` provides functions for adding such annotations.

You can add titles and axis labels to a chart using the **`labs()`** function (*not* `labels()`, which is a different R function!), as in Figure 16.13. This function takes named arguments for each aspect to label—either `title` (or `subtitle` or `caption`), or the name of the aesthetic (e.g., `x`, `y`, `color`). Axis aesthetics such as `x` and `y` will have their label shown on the axis, while other aesthetics will use the provided label for the legend.

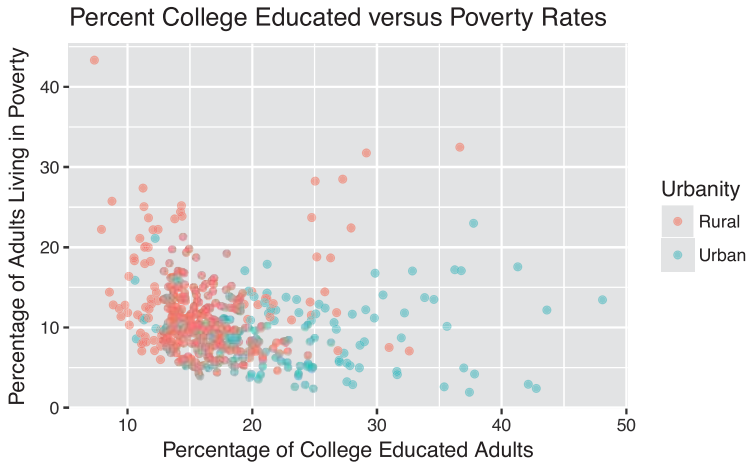

Figure 16.13 A comparison of each county's adult poverty rate and college education rate. The `labs()` function is used to add a title and labels for each aesthetic mapping.

```
# Adding better labels to the plot in Figure 16.10
ggplot(data = labeled) +
  geom_point(
    mapping = aes(x = percollege, y = percadultpoverty, color = location),
    alpha = .6
  ) +

  # Add title and axis labels
  labs(
    title = "Percent College Educated versus Poverty Rates", # plot title
    x = "Percentage of College Educated Adults", # x-axis label
    y = "Percentage of Adults Living in Poverty", # y-axis label
    color = "Urbanity" # legend label for the "color" property
  )
```

You can also add labels into the plot itself (e.g., to label each point or line) by adding a new `geom_text()` (for plain text) or `geom_label()` (for boxed text). In effect, you're plotting an extra set of data values that happen to be the value names. For example, in Figure 16.14, labels are used to identify the county with the highest level of poverty in each state. The background and border for each piece of text is created by using the `geom_label_repel()` function, which provides labels that don't overlap.

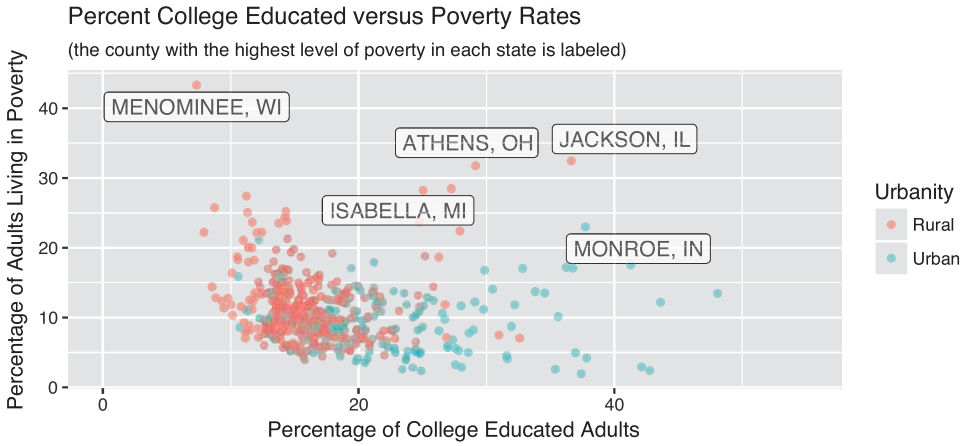

Figure 16.14 Using labels to identify the county in each state with the highest level of poverty. The `ggrepel` package is used to prevent labels from overlapping.

```
# Load the `ggrepel` package: functions that prevent labels from overlapping
library(ggrepel)

# Find the highest level of poverty in each state
most_poverty <- midwest %>%
  group_by(state) %>% # group by state
  top_n(1, wt = percadultpoverty) %>% # select the highest poverty county
  unite(county_state, county, state, sep = ", ") # for clear labeling

# Store the subtitle in a variable for cleaner graphing code
subtitle <- "(the county with the highest level of poverty
  in each state is labeled)"

# Plot the data with labels
ggplot(data = labeled, mapping = aes(x = percollege, y = percadultpoverty)) +

  # add the point geometry
  geom_point(mapping = aes(color = location), alpha = .6) +
```

```
# add the label geometry
geom_label_repel(
  data = most_poverty, # uses its own specified data set
  mapping = aes(label = county_state),
  alpha = 0.8
) +

# set the scale for the axis
scale_x_continuous(limits = c(0, 55)) +

# add title and axis labels
labs(
  title = "Percent College Educated versus Poverty Rates", # plot title
  subtitle = subtitle, # subtitle
  x = "Percentage of College Educated Adults", # x-axis label
  y = "Percentage of Adults Living in Poverty", # y-axis label
  color = "Urbanity" # legend label for the "color" property
)
```

16.4 Building Maps

In addition to building charts using `ggplot2`, you can use the package to draw geographic maps. Because two-dimensional maps already depend on a coordinate system (*latitude* and *longitude*), you can exploit the `ggplot2` Cartesian layout to create geographic visualizations. Generally speaking, there are two types of maps you will want to create:

- **Choropleth maps**: Maps in which different geographic areas are shaded based on data about each region (as in Figure 16.16). These maps can be used to visualize data that is aggregated to specified geographic areas. For example, you could show the eviction rate in each state using a choropleth map. Choropleth maps are also called *heatmaps*.
- **Dot distribution maps**: Maps in which markers are placed at specific coordinates, as in Figure 16.19. These plots can be used to visualize observations that occur at discrete (latitude/longitude) points. For example, you could show the specific address of each eviction notice filed in a given city.

This section details how to build such maps using `ggplot2` and complementary packages.

16.4.1 Choropleth Maps

To draw a choropleth map, you need to first draw the outline of each geographic unit (e.g., state, country). Because each geography will be an irregular closed shape, `ggplot2` can use the `geom_polygon()` function to draw the outlines. To do this, you will need to load a data file that describes the geometries (outlines) of your areas, appropriately called a **shapefile**. Many shapefiles

such as those made available by the U.S. Census Bureau[11] and OpenStreetMap[12] can be freely downloaded and used in R.

To help you get started with mapping, ggplot2 includes a handful of shapefiles (meaning you don't need to download one). You can load a given shapefile by providing the name of the shapefile you wish to load (e.g., `"usa"`, `"state"`, `"world"`) to the **`map_data()`** function. Once you have the desired shapefile in a usable format, you can render a map using the `geom_polygon()` function. This function plots a shape by drawing lines between each individual pair of `x`- and `y`- coordinates (in order), similar to a "connect-the-dots" puzzle. To maintain an appropriate aspect ratio for your map, use the `coord_map()` coordinate system. The map created by the following code is shown in Figure 16.15.

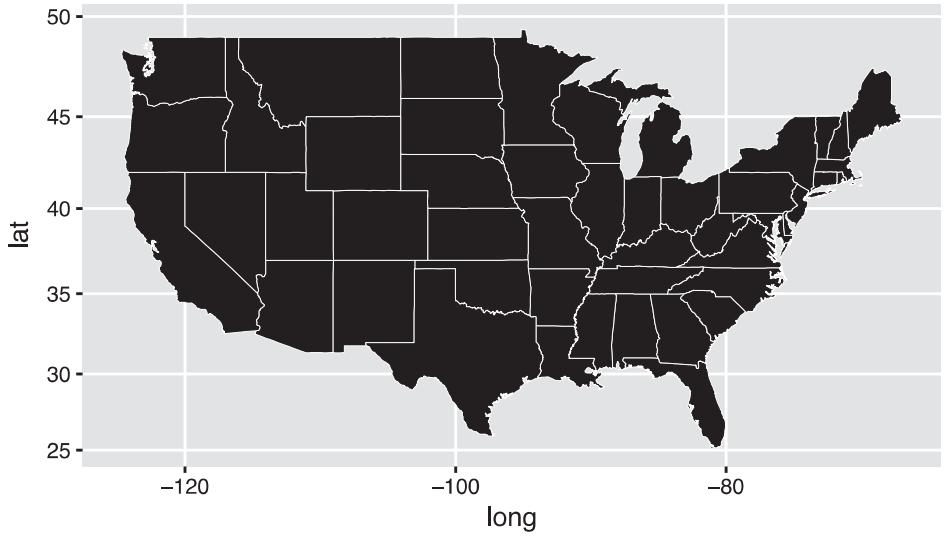

Figure 16.15 A U.S. state map, made with ggplot2.

```
# Load a shapefile of U.S. states using ggplot's `map_data()` function
state_shape <- map_data("state")

# Create a blank map of U.S. states
ggplot(state_shape) +
  geom_polygon(
    mapping = aes(x = long, y = lat, group = group),
    color = "white", # show state outlines
    size = .1        # thinly stroked
  ) +
  coord_map() # use a map-based coordinate system
```

[11] **U.S. Census: Cartographic Boundary Shapefiles**: https://www.census.gov/geo/maps-data/data/tiger-cart-boundary.html

[12] **OpenStreetMap: Shapefiles**: https://wiki.openstreetmap.org/wiki/Shapefiles

The data in the `state_shape` variable is just a data frame of longitude/latitude points that describe how to draw the outline of each state—the `group` variable indicates which state each point belongs to. If you want each geographic area (in this case, each U.S. state) to express different data through a visual channel such as color, you need to *load* the data, *join* it to the shapefile, and *map* the `fill` of each polygon. As is often the case, the biggest challenge is getting the data in the proper format for visualizing it (not using the visualization package). The map in Figure 16.16, which is built using the following code, shows the eviction rate in each U.S. state in 2016. The data was downloaded from the Eviction Lab at Princeton University.[13]

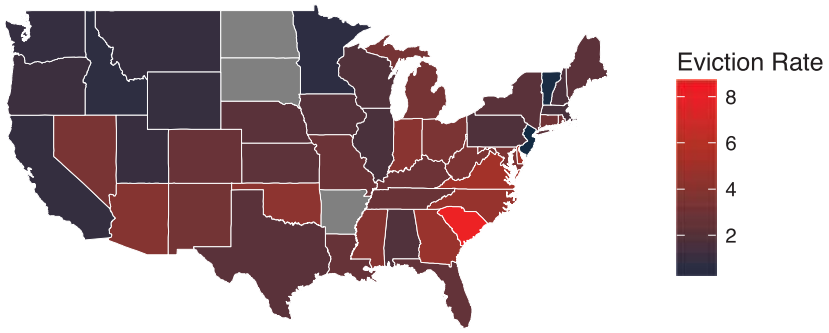

Figure 16.16 A choropleth map of eviction rates by state, made with `ggplot2`.

```
# Load evictions data
evictions <- read.csv("data/states.csv", stringsAsFactors = FALSE) %>%
  filter(year == 2016) %>% # keep only 2016 data
  mutate(state = tolower(state)) # replace with lowercase for joining

# Join eviction data to the U.S. shapefile
state_shape <- map_data("state") %>% # load state shapefile
  rename(state = region) %>% # rename for joining
  left_join(evictions, by="state") # join eviction data

# Draw the map setting the `fill` of each state using its eviction rate
ggplot(state_shape) +
  geom_polygon(
    mapping = aes(x = long, y = lat, group = group, fill = eviction.rate),
    color = "white", # show state outlines
    size = .1        # thinly stroked
  ) +
  coord_map() + # use a map-based coordinate system
  scale_fill_continuous(low = "#132B43", high = "Red") +
  labs(fill = "Eviction Rate") +
  blank_theme # variable containing map styles (defined in next code snippet)
```

[13] **Eviction Lab**: https://evictionlab.org. The Eviction Lab at Princeton University is a project directed by Matthew Desmond and designed by Ashley Gromis, Lavar Edmonds, James Hendrickson, Katie Krywokulski, Lillian Leung, and Adam Porton. The Eviction Lab is funded by the JPB, Gates, and Ford Foundations, as well as the Chan Zuckerberg Initiative.

The beauty and challenge of working with ggplot2 are that nearly every visual feature is configurable. These features can be adjusted using the **theme()** function for any plot (including maps!). Nearly every granular detail—minor grid lines, axis tick color, and more—is available for your manipulation. See the documentation[14] for details. The following is an example set of styles targeted to remove default visual features from maps:

```
# Define a minimalist theme for maps
blank_theme <- theme_bw() +
  theme(
    axis.line = element_blank(),         # remove axis lines
    axis.text = element_blank(),         # remove axis labels
    axis.ticks = element_blank(),        # remove axis ticks
    axis.title = element_blank(),        # remove axis titles
    plot.background = element_blank(),  # remove gray background
    panel.grid.major = element_blank(), # remove major grid lines
    panel.grid.minor = element_blank(), # remove minor grid lines
    panel.border = element_blank()       # remove border around plot
  )
```

16.4.2 Dot Distribution Maps

ggplot also allows you to plot data at discrete locations on a map. Because you are already using a geographic coordinate system, it is somewhat trivial to add discrete points to a map. The following code generates Figure 16.17:

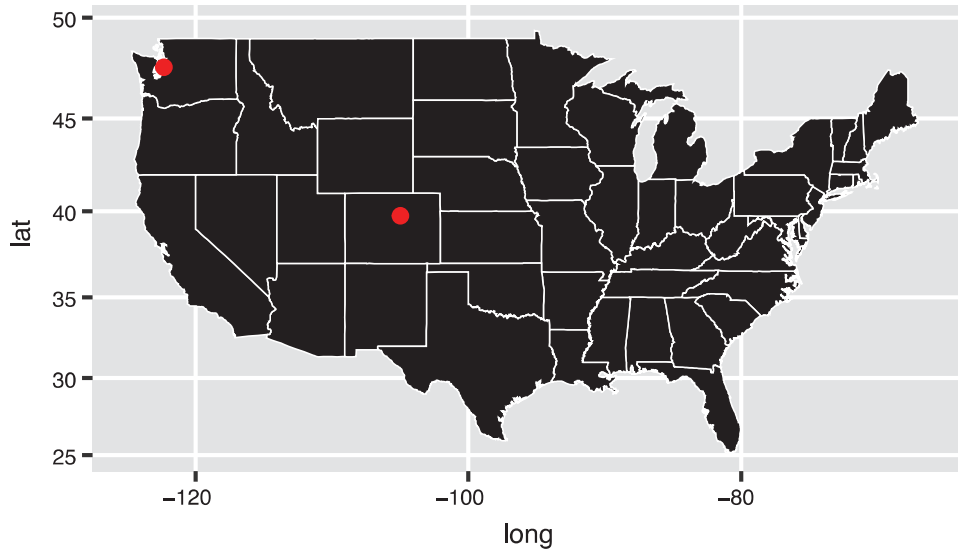

Figure 16.17 Adding discrete points to a map.

[14] **ggplot2 themes** reference: http://ggplot2.tidyverse.org/reference/index.html#section-themes

```
# Create a data frame of city coordinates to display
cities <- data.frame(
  city = c("Seattle", "Denver"),
  lat = c(47.6062, 39.7392),
  long = c(-122.3321, -104.9903)
)

# Draw the state outlines, then plot the city points on the map
ggplot(state_shape) +
  geom_polygon(mapping = aes(x = long, y = lat, group = group)) +
  geom_point(
    data = cities, # plots own data set
    mapping = aes(x = long, y = lat), # points are drawn at given coordinates
    color = "red"
  ) +
  coord_map() # use a map-based coordinate system
```

As you seek to increase the granularity of your map visualizations, it may be infeasible to describe every feature with a set of coordinates. This is why many visualizations use images (rather than polygons) to show geographic information such as streets, topography, buildings, and other geographic features. These images are called **map tiles**—they are pictures that can be stitched together to represent a geographic area. Map tiles are usually downloaded from a remote server, and then combined to display the complete map. The **ggmap**[15] package provides a nice extension to ggplot2 for both downloading map tiles and rendering them in R. Map tiles are also used with the Leaflet package, described in Chapter 17.

16.5 ggplot2 in Action: Mapping Evictions in San Francisco

To demonstrate the power of ggplot2 as a visualization tool for understanding pertinent social issues, this section visualizes eviction notices filed in San Francisco in 2017.[16] The complete code for this analysis is also available online in the book code repository.[17]

Before mapping this data, a minor amount of formatting needs to be done on the raw data set (shown in Figure 16.18):

```
# Load and format eviction notices data
# Data downloaded from https://catalog.data.gov/dataset/eviction-notices

# Load packages for data wrangling and visualization
library("dplyr")
library("tidyr")
```

[15] ggmap repository on GitHub: https://github.com/dkahle/ggmap
[16] **data.gov: Eviction Notices**: https://catalog.data.gov/dataset/eviction-notices
[17] **ggplot2 in Action**: https://github.com/programming-for-data-science/in-action/tree/master/ggplot2

```
# Load .csv file of notices
notices <- read.csv("data/Eviction_Notices.csv", stringsAsFactors = F)

# Data wrangling: format dates, filter to 2017 notices, extract lat/long data
notices <- notices %>%
  mutate(date = as.Date(File.Date, format="%m/%d/%y")) %>%
  filter(format(date, "%Y") == "2017") %>%
  separate(Location, c("lat", "long"), ", ") %>% # split column at the comma
  mutate(
    lat = as.numeric(gsub("\\(", "", lat)), # remove starting parentheses
    long = as.numeric(gsub("\\)", "", long)) # remove closing parentheses
  )
```

	Eviction.ID	Address	City	State	Eviction.Notice.Source.Zipcode	File.Date
1	M172475	3400 Block Of Cabrillo Street	San Francisco	CA	94121	10/6/17
2	M172687	200 Block Of Lincoln Way	San Francisco	CA	94122	10/23/17
3	M172665	100 Block Of San Jose Avenue	San Francisco	CA	94110	10/27/17
4	M172474	1500 Block Of Gough Street	San Francisco	CA	94109	10/6/17
5	M172571	900 Block Of Larkin Street	San Francisco	CA	94109	10/16/17
6	M172642	2300 Block Of Mission Street	San Francisco	CA	94110	10/19/17
7	M172623	100 Block Of Charles Street	San Francisco	CA	94131	10/19/17
8	M172560	1200 Block Of 40th Avenue	San Francisco	CA	94122	10/13/17
9	M172484	1300 Block Of Clement Street	San Francisco	CA	94118	10/10/17
10	M172684	200 Block Of Lincoln Way	San Francisco	CA	94122	10/23/17

Figure 16.18 A subset of the eviction notices data downloaded from *data.gov*.

To create a background map of San Francisco, you can use the `qmplot()` function from the *development* version of `ggmap` package (see below). Because the `ggmap` package is built to work with `ggplot2`, you can then display points on top of the map as you normally would (using `geom_point()`). Figure 16.19 shows the location of each eviction notice filed in 2017, created using the following code:

> **Tip**: Installing the *development* version of a package using `devtools::install_github("PACKAGE_NAME")` provides you access to the most recent version of a package, including bug fixes and new—though not always fully tested—features.

```
# Create a map of San Francisco, with a point at each eviction notice address
# Use `install_github()` to install the newer version of `ggmap` on GitHub
# devtools::install_github("dkhale/ggmap") # once per machine
library("ggmap")
library("ggplot2")
```

```
# Create the background of map tiles
base_plot <- qmplot(
    data = notices,               # name of the data frame
    x = long,                     # data feature for longitude
    y = lat,                      # data feature for latitude
    geom = "blank",               # don't display data points (yet)
    maptype = "toner-background", # map tiles to query
    darken = .7,                  # darken the map tiles
    legend = "topleft"            # location of legend on page
)

# Add the locations of evictions to the map
base_plot +
  geom_point(mapping = aes(x = long, y = lat), color = "red", alpha = .3) +
  labs(title = "Evictions in San Francisco, 2017") +
  theme(plot.margin = margin(.3, 0, 0, 0, "cm")) # adjust spacing around the map
```

Figure 16.19 Location of each eviction notice in San Francisco in 2017. The image is generated by layering points on top of map tiles using the ggplot2 package.

> **Tip**: You can store a plot returned by the `ggplot()` function in a variable (as in the preceding code)! This allows you to add different layers on top of a *base* plot, or to render the plot at chosen locations throughout a report (see Chapter 18).

While Figure 16.19 captures the gravity of the issue of evictions in the city, the overlapping nature of the points prevents ready identification of any patterns in the data. Using the `geom_polygon()` function, you can compute point density across two dimensions and display the computed values in *contours*, as shown in Figure 16.20.

```
# Draw a heatmap of eviction rates, computing the contours
base_plot +
  geom_polygon(
    stat = "density2d", # calculate two-dimensional density of points (contours)
    mapping = aes(fill = stat(level)), # use the computed density to set the fill
    alpha = .3 # Set the alpha (transparency)
  ) +
  scale_fill_gradient2(
    "# of Evictions",
    low = "white",
    mid = "yellow",
    high = "red"
  ) +
  labs(title="Number of Evictions in San Francisco, 2017") +
  theme(plot.margin = margin(.3, 0, 0, 0, "cm"))
```

This example of the `geom_polygon()` function uses the `stat` argument to automatically perform a **statistical transformation** (aggregation)—similar to what you could do using the `dplyr` functions `group_by()` and `summarize()`—that calculates the shape and color of each contour based on point density (a `"density2d"` aggregation). `ggplot2` stores the result of this aggregation in an internal data frame in a column labeled `level`, which can be accessed using the `stat()` helper function to set the `fill` (that is, `mapping = aes(fill = stat(level))`).

> **Tip**: For more examples of producing maps with `ggplot2`, see this tutorial.[a]

[a] http://eriqande.github.io/rep-res-web/lectures/making-maps-with-R.html

This chapter introduced the `ggplot2` package for constructing precise data visualizations. While the intricacies of this package can be difficult to master, the investment is well worth the effort, as it enables you to control the granular details of your visualizations.

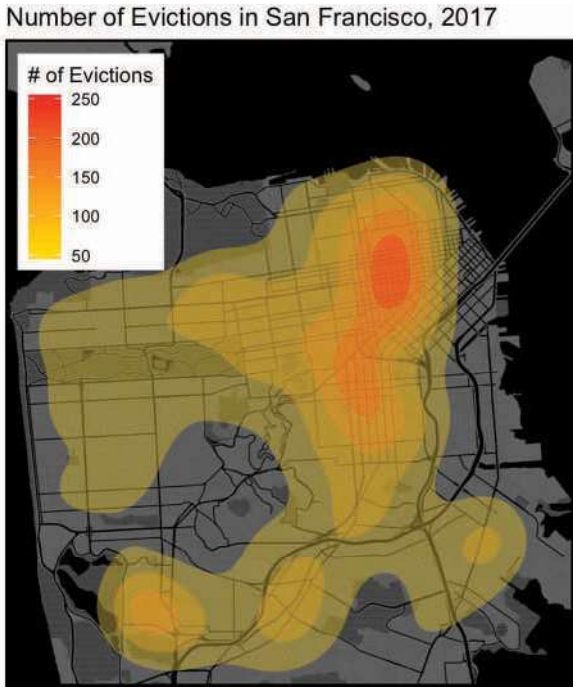

Figure 16.20 A *heatmap* of eviction notices in San Francisco. The image is created by aggregating eviction notices into 2D Contours with one of ggplot2's *statistical transformations*.

> **Tip**: Similar to dplyr and many other packages, ggplot2 has a large number of functions. A cheatsheet for the package is available through the RStudio menu: `Help > Cheatsheets`. In addition, this phenomenal cheatsheet[a] describes how to control the granular details of your ggplot2 visualizations.
>
> [a] http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/

For practice creating configurable visualizations with ggplot2, see the set of accompanying book exercises.[18]

[18] **ggplot2 exercises**: https://github.com/programming-for-data-science/chapter-16-exercises