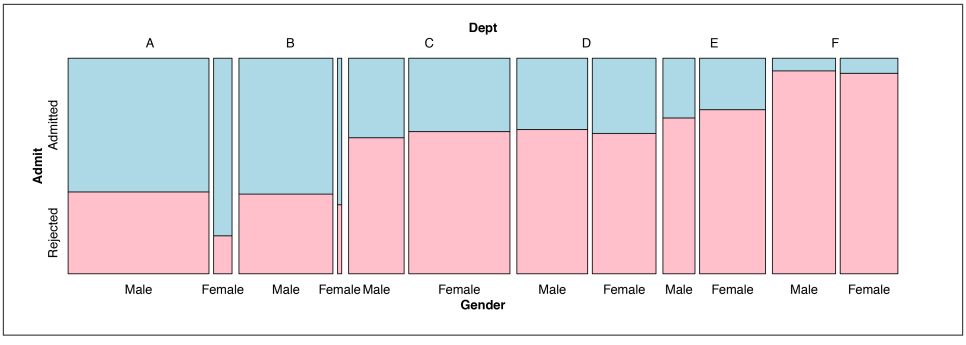

Figure 13-29. Splitting Dept vertically, Gender vertically, and Admit horizontally

The example here illustrates a classic case of Simpson's paradox, in which a relationship between variables within subgroups can change (or reverse!) when the groups are combined. The `UCBerkeley` table contains admissions data from the University of California-Berkeley in 1973. Overall, men were admitted at a higher rate than women, and because of this, the university was sued for gender bias. But when each department was examined separately, it was found that they each had approximately equal admission rates for men and women. The difference in overall admission rates was because women were more likely to apply to competitive departments with lower admission rates.

In Figures 13-28 and 13-29, you can see that within each department, admission rates were approximately equal between men and women. You can also see that departments with higher admission rates (A and B) were very imbalanced in the gender ratio of applicants: far more men applied to these departments than did women. As you can see, partitioning the data in different orders and directions can bring out different aspects of the data. In Figure 13-29, as in Figure 13-28, it's easy to compare male and female admission rates within each department and across departments. Splitting `Dept` vertically, `Gender` horizontally, and `Admit` horizontally, as in Figure 13-30, makes it difficult to compare male and female admission rates within each department, but it is easy to compare male and female application rates across departments.

See Also

See `?mosiacplot` for another function that can create mosaic plots.

P.J. Bickel, E.A. Hammel, and J.W. O'Connell, "Sex Bias in Graduate Admissions: Data from Berkeley," *Science* 187 (1975): 398–404.

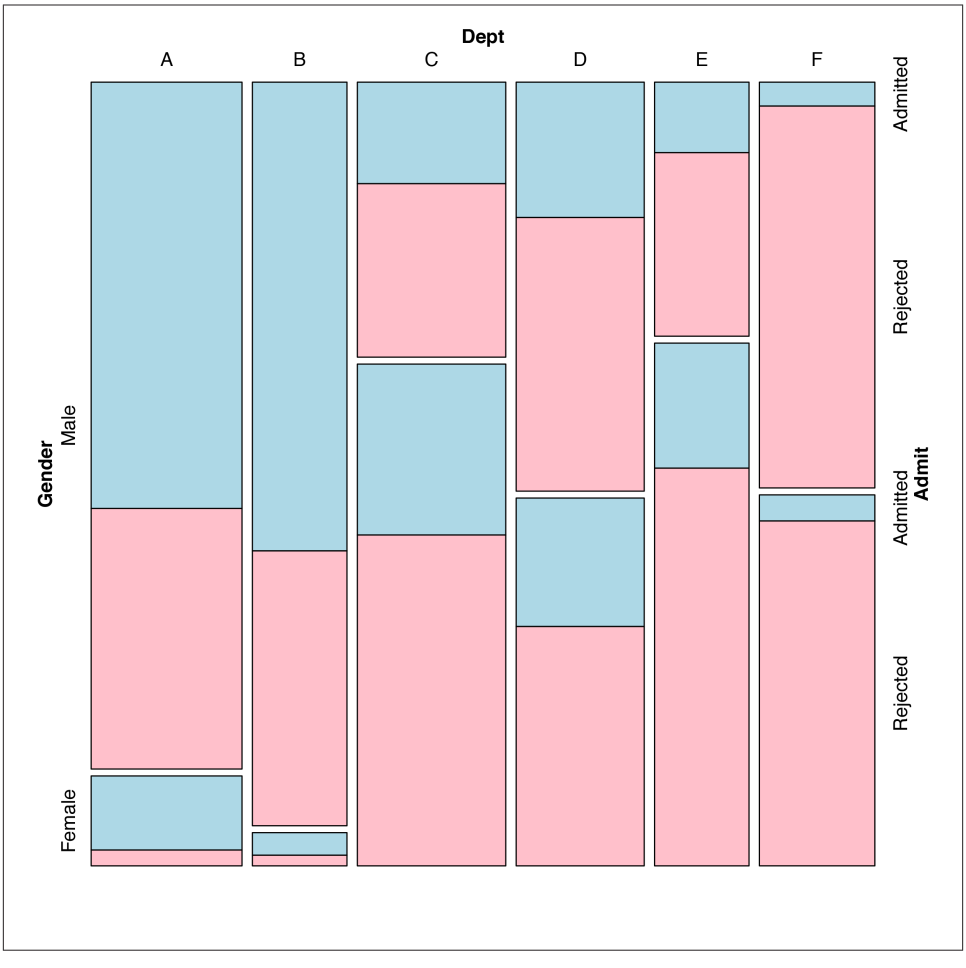

Figure 13-30. Splitting Dept vertically, Gender horizontally, and Admit horizontally

13.16. Creating a Pie Chart

Problem

You want to make a pie chart.

Solution

Use the `pie()` function. In this example (Figure 13-31), we'll use the `survey` data set from the MASS library:

```
library(MASS)  # For the data set

# Get a table of how many cases are in each level of fold
fold <- table(survey$Fold)
fold
```

```
 L on R Neither  R on L
     99      18     120
```

```
# Make the pie chart
pie(fold)
```

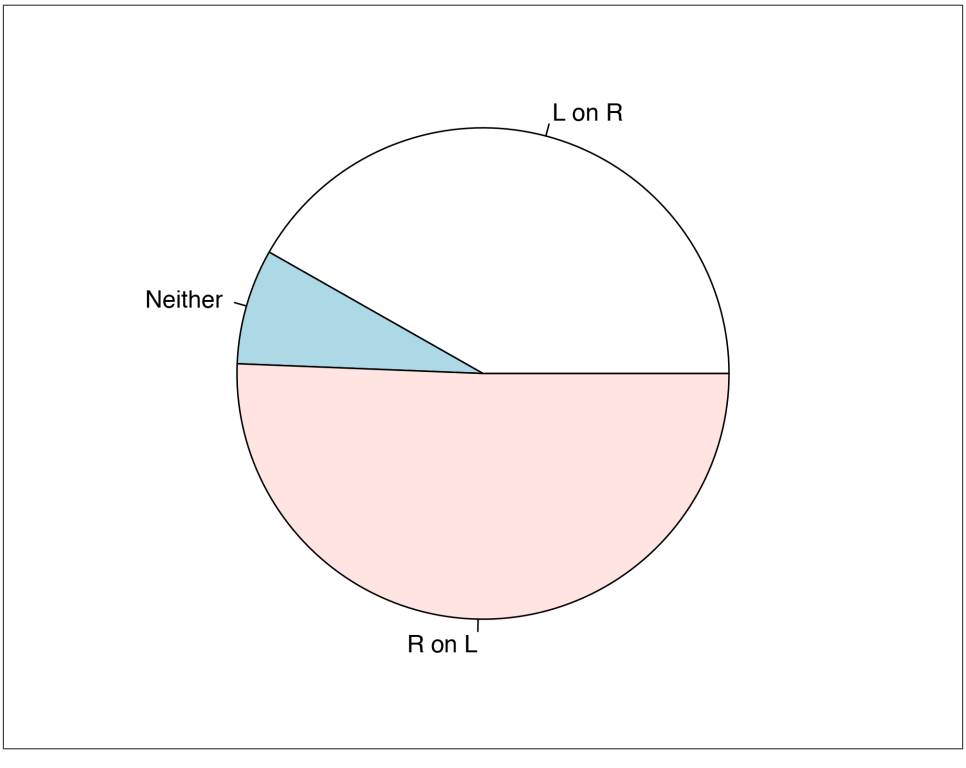

Figure 13-31. A pie chart

We passed `pie()` an object of class `table`. We could have instead given it a named vector, or a vector of values and a vector of labels, like this:

```
pie(c(99, 18, 120), labels=c("L on R", "Neither", "R on L"))
```

Discussion

The lowly pie chart is the subject of frequent abuse from data visualization experts. If you're thinking of using a pie chart, consider whether a bar graph (or stacked bar graph) would convey the information more effectively. Despite their faults, pie charts do have one important virtue: everyone knows how to read them.

13.17. Creating a Map

Problem

You want to create a geographical map.

Solution

Retrieve map data from the maps package and draw it with `geom_polygon()` (which can have a color fill) or `geom_path()` (which can't have a fill). By default, the latitude and longitude will be drawn on a Cartesian coordinate plane, but you can use `coord_map()` and specify a projection. The default projection is `"mercator"`, which, unlike the Cartesian plane, has a progressively changing spacing for latitude lines (Figure 13-32):

```
library(maps) # For map data
# Get map data for USA
states_map <- map_data("state")

ggplot(states_map, aes(x=long, y=lat, group=group)) +
    geom_polygon(fill="white", colour="black")

# geom_path (no fill) and Mercator projection
ggplot(states_map, aes(x=long, y=lat, group=group)) +
    geom_path() + coord_map("mercator")
```

Discussion

The `map_data()` function returns a data frame with the following columns:

`long`
: Longitude.

`lat`
: Latitude.

`group`
: This is a grouping variable for each polygon. A region or subregion might have multiple polygons, for example, if it includes islands.

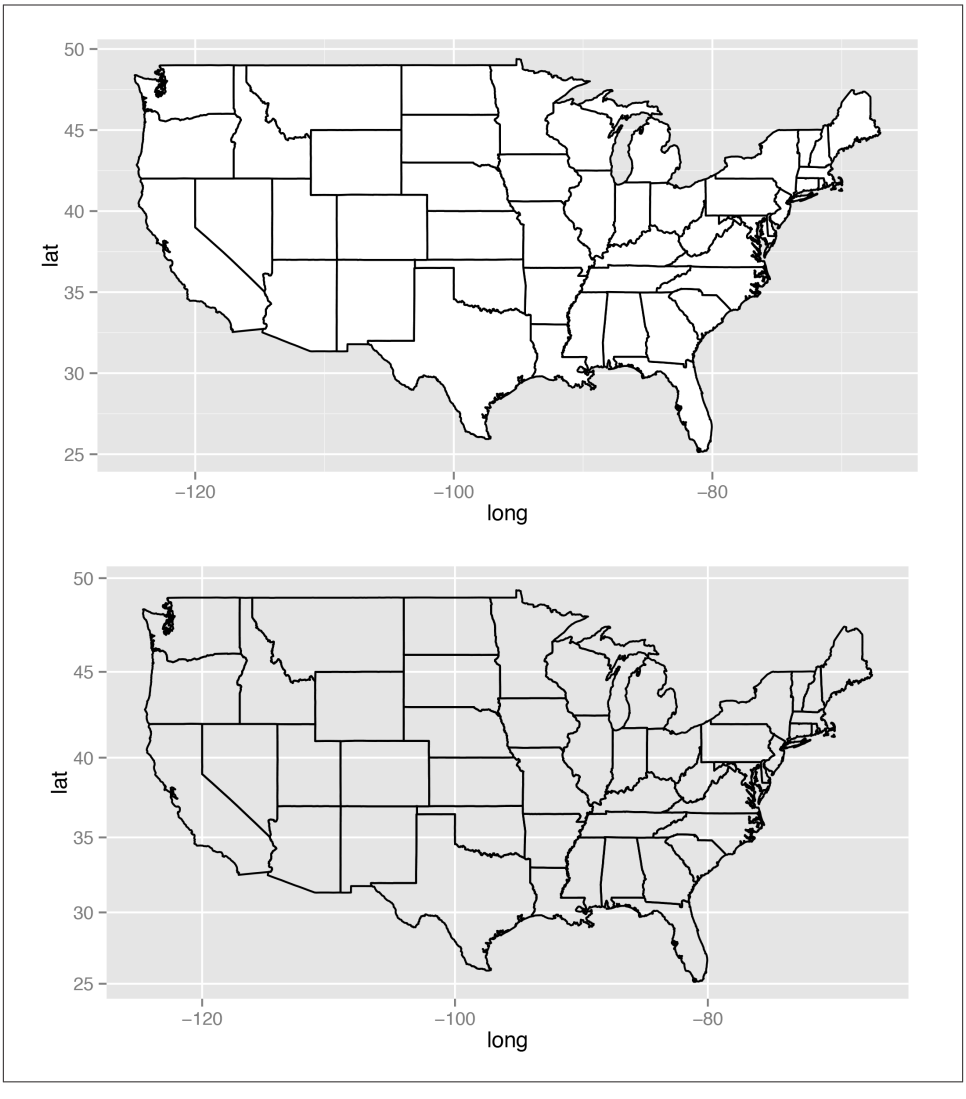

Figure 13-32. Top: a basic map with fill; bottom: with no fill, and Mercator projection

`order`
: The order to connect each point within a group.

`region`
: Roughly, the names of countries, although some other objects are present (such as some lakes).

`subregion`
: The names of subregions within a region, which can contain multiple groups. For example, the Alaska subregion includes many islands, each with its own group.

There are a number of different maps available, including `world`, `nz`, `france`, `italy`, `usa` (outline of the United States), `state` (each state in the USA), and `county` (each county in the USA). For example, to get map data for the world:

```
# Get map data for world
world_map <- map_data("world")
world_map
```

```
      long      lat group order      region subregion
 -133.3664 58.42416     1     1      Canada      <NA>
 -132.2681 57.16308     1     2      Canada      <NA>
 -132.0498 56.98610     1     3      Canada      <NA>
 ...
  124.7772 11.35419  2284 27634 Philippines     Leyte
  124.9697 11.30280  2284 27635 Philippines     Leyte
  125.0155 11.13887  2284 27636 Philippines     Leyte
```

If you want to draw a map of a region in the `world` map for which there isn't a separate map, you can first look for the region name, like so:

```
sort(unique(world_map$region))
```

```
 "Afghanistan"            "Albania"                "Algeria"
 "American Samoa"         "Andaman Islands"        "Andorra"
 "Angola"                 "Anguilla"               "Antarctica"
  ...
 "USA"                    "USSR"                   "Vanuatu"
 "Venezuela"              "Vietnam"                "Virgin Islands"
 "Vislinskiy Zaliv"       "Wales"                  "West Bank"
 "Western Sahara"         "Yemen"                  "Yugoslavia"
 "Zaire"                  "Zambia"                 "Zimbabwe"
```

```
# You might have noticed that it's a little out of date!
```

It's possible to get data for specific regions from a particular map (Figure 13-33):

```
east_asia <- map_data("world", region=c("Japan", "China", "North Korea",
                                        "South Korea"))
# Map region to fill color
ggplot(east_asia, aes(x=long, y=lat, group=group, fill=region)) +
    geom_polygon(colour="black") +
    scale_fill_brewer(palette="Set2")
```

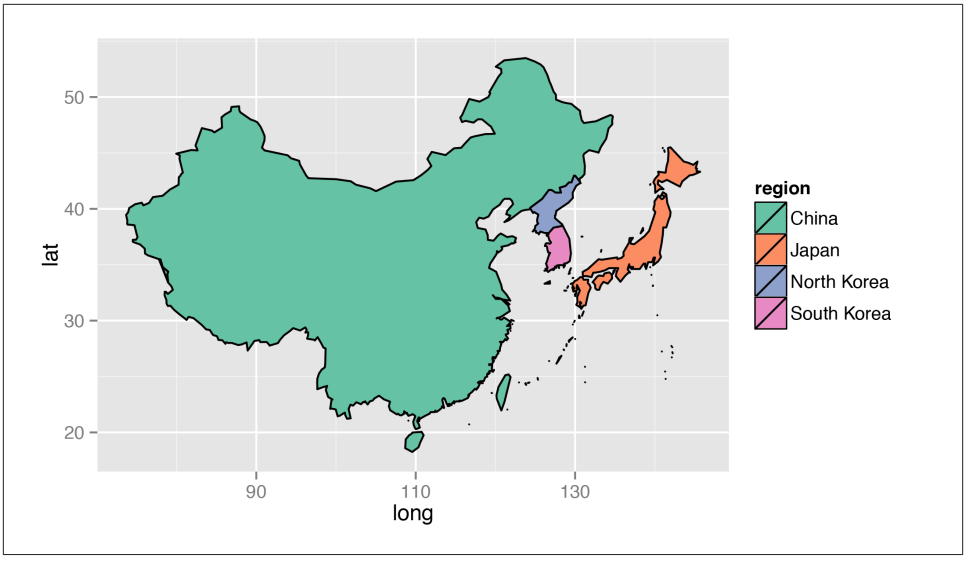

Figure 13-33. Specific regions from the world map

If there is a separate map available for a region, such as `nz` (New Zealand), that map data will be at a higher resolution than if you were to extract it from the `world` map, as shown in Figure 13-34:

```
# Get New Zealand data from world map
nz1 <- map_data("world", region="New Zealand")
nz1 <- subset(nz1, long > 0 & lat > -48)        # Trim off islands
ggplot(nz1, aes(x=long, y=lat, group=group)) + geom_path()

# Get New Zealand data from the nz map
nz2 <- map_data("nz")
ggplot(nz2, aes(x=long, y=lat, group=group)) + geom_path()
```

See Also

See the `mapdata` package for more map data sets. It includes maps of China and Japan, as well as a high-resolution world map, `worldHires`.

See the `map()` function, for quickly generating maps.

See `?mapproject` for a list of available map projections.

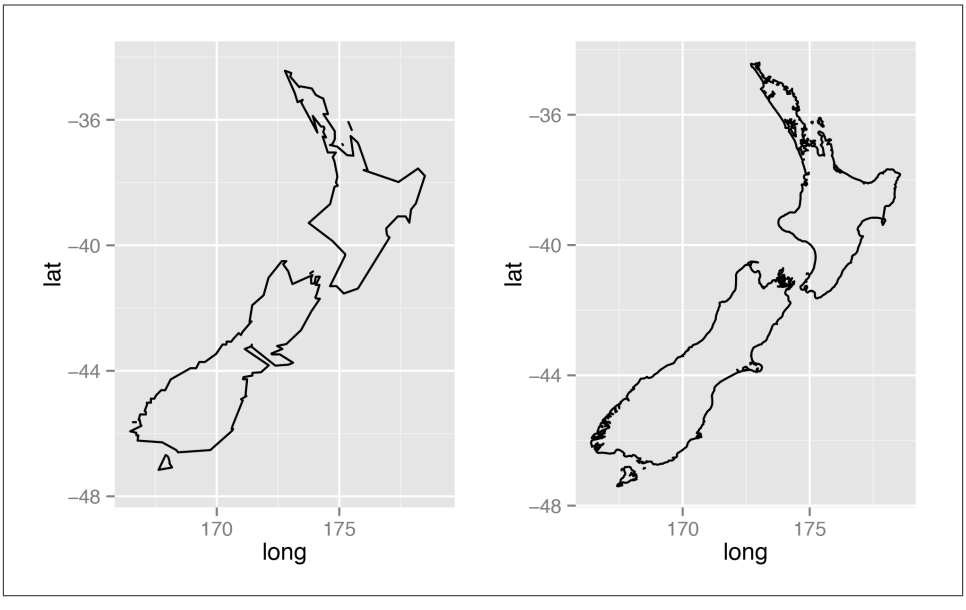

Figure 13-34. Left: New Zealand data taken from world map; right: data from nz map

13.18. Creating a Choropleth Map

Problem

You want to create a map with regions that are colored according to variable values.

Solution

Merge the value data with the map data, then map a variable to `fill`:

```
# Transform the USArrests data set to the correct format
crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
crimes
```

```
                      state Murder Assault UrbanPop Rape
Alabama             alabama   13.2     236       58 21.2
Alaska               alaska   10.0     263       48 44.5
Arizona             arizona    8.1     294       80 31.0
 ...
West Virginia   west virginia    5.7      81       39  9.3
Wisconsin           wisconsin    2.6      53       66 10.8
Wyoming               wyoming    6.8     161       60 15.6
```

```
library(maps) # For map data
states_map <- map_data("state")
```

```
# Merge the data sets together
crime_map <- merge(states_map, crimes, by.x="region", by.y="state")

# After merging, the order has changed, which would lead to polygons drawn in
# the incorrect order. So, we sort the data.
head(crime_map)
```

```
  region      long      lat group order subregion Murder Assault UrbanPop Rape
 alabama -87.46201 30.38968     1     1      <NA>   13.2     236       58 21.2
 alabama -87.48493 30.37249     1     2      <NA>   13.2     236       58 21.2
 alabama -87.95475 30.24644     1    13      <NA>   13.2     236       58 21.2
 alabama -88.00632 30.24071     1    14      <NA>   13.2     236       58 21.2
 alabama -88.01778 30.25217     1    15      <NA>   13.2     236       58 21.2
 alabama -87.52503 30.37249     1     3      <NA>   13.2     236       58 21.2
```

```
library(plyr)  # For arrange() function
# Sort by group, then order
crime_map <- arrange(crime_map, group, order)
head(crime_map)
```

```
  region      long      lat group order subregion Murder Assault UrbanPop Rape
 alabama -87.46201 30.38968     1     1      <NA>   13.2     236       58 21.2
 alabama -87.48493 30.37249     1     2      <NA>   13.2     236       58 21.2
 alabama -87.52503 30.37249     1     3      <NA>   13.2     236       58 21.2
 alabama -87.53076 30.33239     1     4      <NA>   13.2     236       58 21.2
 alabama -87.57087 30.32665     1     5      <NA>   13.2     236       58 21.2
 alabama -87.58806 30.32665     1     6      <NA>   13.2     236       58 21.2
```

Once the data is in the correct format, it can be plotted (Figure 13-35), mapping one of the columns with data values to `fill`:

```
ggplot(crime_map, aes(x=long, y=lat, group=group, fill=Assault)) +
    geom_polygon(colour="black") +
    coord_map("polyconic")
```

Discussion

The preceding example used the default color scale, which goes from dark to light blue. If you want to show how the values diverge from some middle value, you can use `scale_fill_gradient2()`, as shown in Figure 13-36:

```
ggplot(crimes, aes(map_id = state, fill=Assault)) +
    geom_map(map = states_map, colour="black") +
    scale_fill_gradient2(low="#559999", mid="grey90", high="#BB650B",
                         midpoint=median(crimes$Assault)) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    coord_map("polyconic")
```

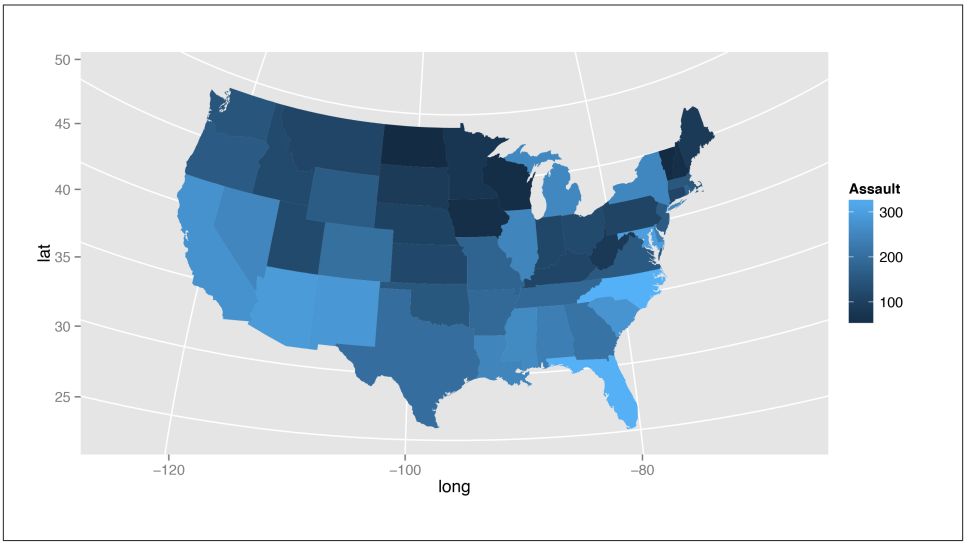

Figure 13-35. A map with a variable mapped to fill

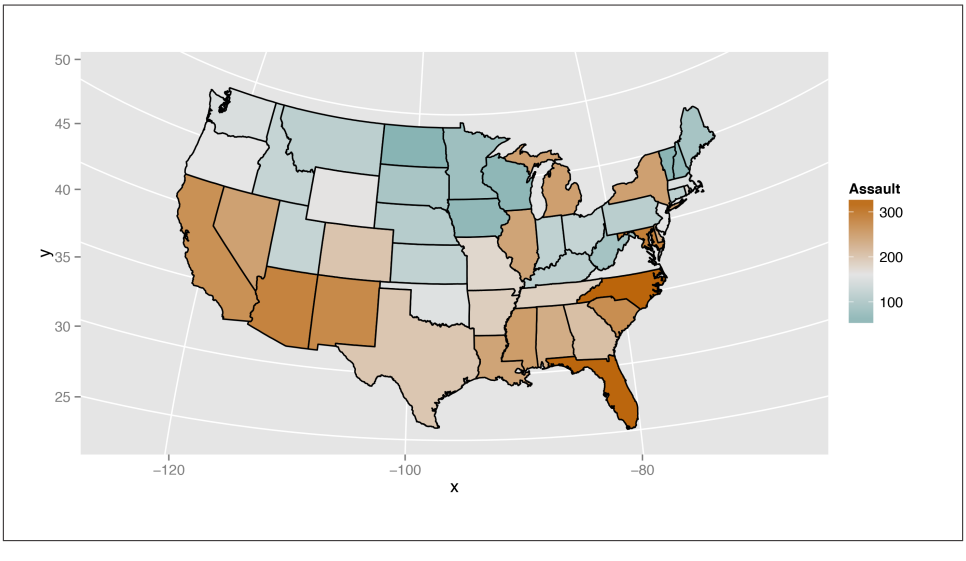

Figure 13-36. With a diverging color scale

The previous example mapped continuous values to `fill`, but we could just as well use discrete values. It's sometimes easier to interpret the data if the values are discretized. For example, we can categorize the values into quantiles and show those quantiles, as in Figure 13-37:

```
# Find the quantile bounds
qa <- quantile(crimes$Assault, c(0, 0.2, 0.4, 0.6, 0.8, 1.0))
qa
```

```
   0%   20%   40%   60%   80%  100%
 45.0  98.8 135.0 188.8 254.2 337.0
```

```
# Add a column of the quantile category
crimes$Assault_q <- cut(crimes$Assault, qa,
                        labels=c("0-20%", "20-40%", "40-60%", "60-80%", "80-100%"),
                        include.lowest=TRUE)
crimes
```

```
                      state Murder Assault UrbanPop Rape Assault_q
Alabama               alabama   13.2     236       58 21.2    60-80%
Alaska                 alaska   10.0     263       48 44.5   80-100%
 ...
Wisconsin           wisconsin    2.6      53       66 10.8     0-20%
Wyoming               wyoming    6.8     161       60 15.6    40-60%
```

```
# Generate a discrete color palette with 5 values
pal <- colorRampPalette(c("#559999", "grey80", "#BB650B"))(5)
pal
```

```
 "#559999" "#90B2B2" "#CCCCCC" "#C3986B" "#BB650B"
```

```
ggplot(crimes, aes(map_id = state, fill=Assault_q)) +
    geom_map(map = states_map, colour="black") +
    scale_fill_manual(values=pal) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    coord_map("polyconic") +
    labs(fill="Assault Rate\nPercentile")
```

Another way to make a choropleth, but without needing to merge the map data with the value data, is to use `geom_map()`. As of this writing, this will render maps faster than the method just described.

For this method, the map data frame must have columns named `lat`, `long`, and `region`. In the value data frame, there must be a column that is matched to the `region` column in the map data frame, and this column is specified by mapping it to the `map_id` aesthetic. For example, this code will have the same output as the first example (Figure 13-35):

```
# The 'state' column in the crimes data is to be matched to the 'region' column
# in the states_map data
ggplot(crimes, aes(map_id = state, fill=Assault)) +
    geom_map(map = states_map) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    coord_map("polyconic")
```

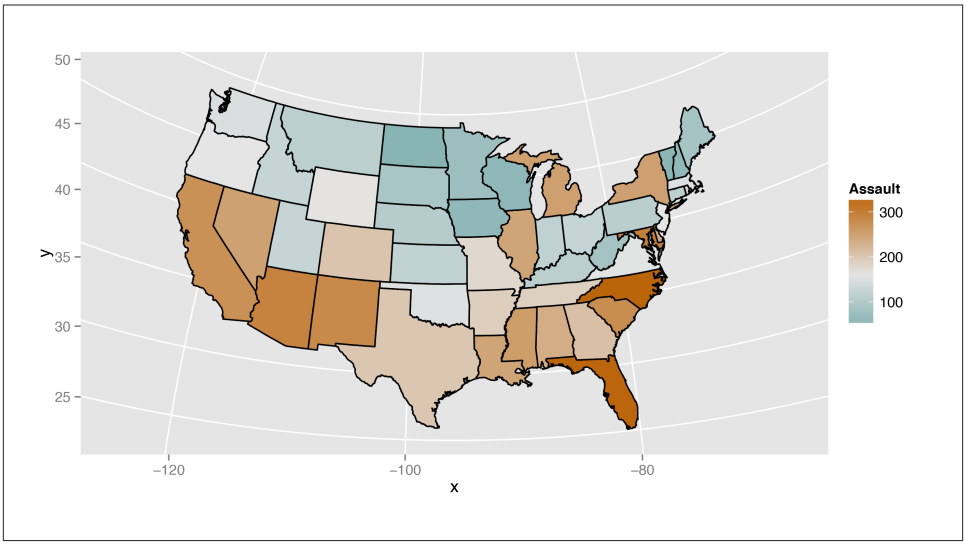

Figure 13-37. Choropleth map with discretized data

Notice that we also needed to use `expand_limits()`. This is because unlike most geoms, `geom_map()` doesn't automatically set the `x` and `y` limits; the use of `expand_limits()` makes it include those `x` and `y` values. (Another way to accomplish the same result is to use `ylim()` and `xlim()`.)

See Also

For an example of data overlaid on a map, see Recipe 13.12.

For more on using continuous colors, see Recipe 12.6.

13.19. Making a Map with a Clean Background

Problem

You want to remove background elements from a map.

Solution

First, save the following theme:

```
# Create a theme with many of the background elements removed
theme_clean <- function(base_size = 12) {
require(grid) # Needed for unit() function
  theme_grey(base_size) %+replace%
```

```
  theme(
    axis.title        = element_blank(),
    axis.text         = element_blank(),
    panel.background  = element_blank(),
    panel.grid        = element_blank(),
    axis.ticks.length = unit(0, "cm"),
    axis.ticks.margin = unit(0, "cm"),
    panel.margin      = unit(0, "lines"),
    plot.margin       = unit(c(0, 0, 0, 0), "lines"),
    complete = TRUE
  )
}
```

Then add it to the map (Figure 13-38). In this example, we'll add it to one of the choropleths we created in Recipe 13.18:

```
ggplot(crimes, aes(map_id = state, fill=Assault_q)) +
    geom_map(map = states_map, colour="black") +
    scale_fill_manual(values=pal) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    coord_map("polyconic") +
    labs(fill="Assault Rate\nPercentile") +
    theme_clean()
```

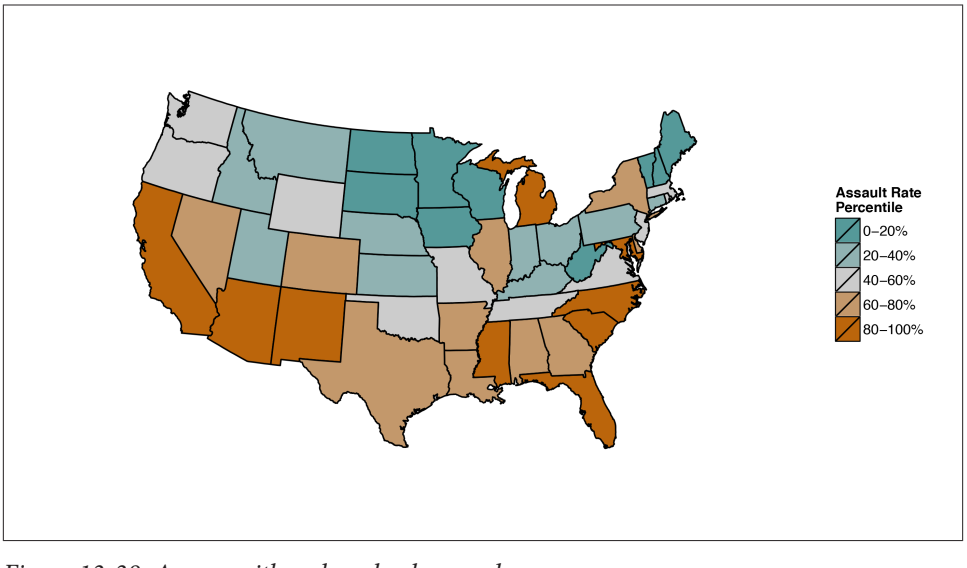

Figure 13-38. A map with a clean background

There's a bug in R versions 2.15.2 and earlier, which may throw an error that looks like this:

```
Error in grid.Call.graphics(L_setviewport, pvp, TRUE) :
  Non-finite location and/or size for viewport
```

This happens because some dimensions add up to having zero length, and the grid graphics engine has trouble handling this. This bug should be fixed in R 2.16. If you're using a version of R where this happens, you can work around it by changing the theme to use `axis.ticks.margin = unit(0.01, "cm")` instead of `axis.ticks.margin = unit(0, "cm")`.

Discussion

In some maps, it's important to include contextual information such as the latitude and longitude. In others, this information is unimportant and distracts from the information that's being conveyed. In Figure 13-38, it's unlikely that viewers will care about the latitude and longitude of the states. They can probably identify the states by shape and relative position, and even if they can't, having the latitude and longitude isn't really helpful.

13.20. Creating a Map from a Shapefile

Problem

You want to create a geographical map from an Esri shapefile.

Solution

Load the shapefile using `readShapePoly()` from the maptools package, convert it to a data frame with `fortify()`, then plot it (Figure 13-39):

```
library(maptools)

# Load the shapefile and convert to a data frame
taiwan_shp <- readShapePoly("TWN_adm/TWN_adm2.shp")
taiwan_map <- fortify(taiwan_shp)

ggplot(taiwan_map, aes(x = long, y = lat, group=group)) + geom_path()
```

Discussion

Esri shapefiles are a common format for map data. The `readShapePoly()` function reads a shape file and returns a `SpatialPolygonsDataFrame` object:

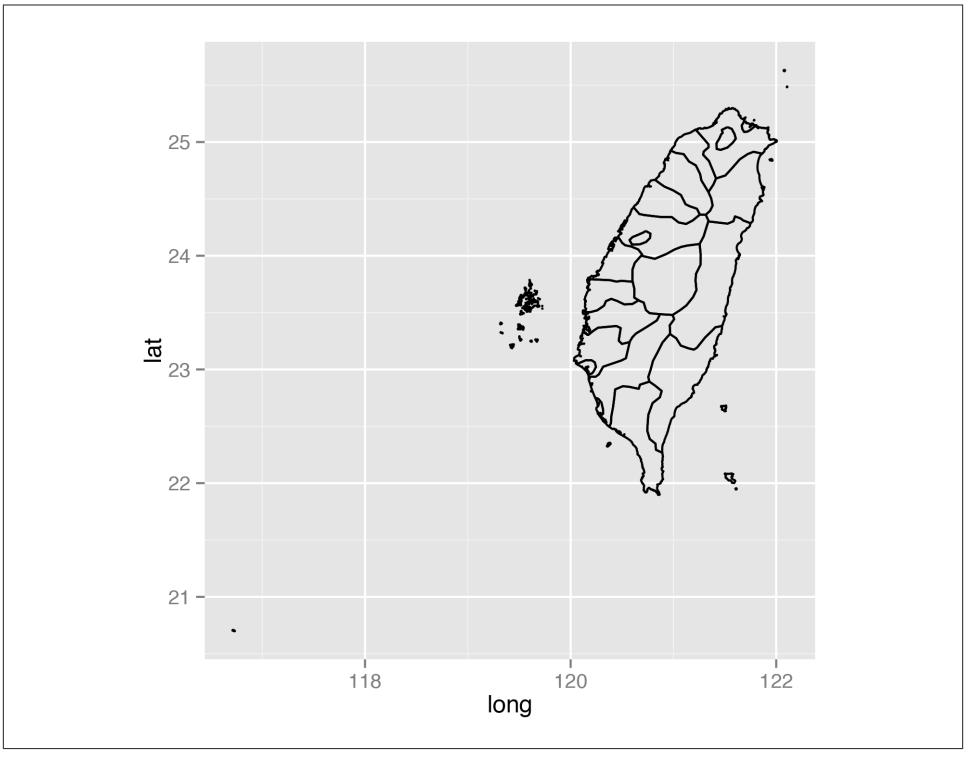

Figure 13-39. A map created from a shapefile

```
taiwan_shp <- readShapePoly("TWN_adm/TWN_adm2.shp")

# Look at the structure of the object
str(taiwan_shp)
```

```
Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame': 22 obs. of  11 variables:
  .. ..$ ID_0     : int [1:22] 223 223 223 223 223 223 223 223 223 223 ...
  .. ..$ ISO      : Factor w/ 1 level "TWN": 1 1 1 1 1 1 1 1 1 1 ...
  .. ..$ NAME_0   : Factor w/ 1 level "Taiwan": 1 1 1 1 1 1 1 1 1 1 ...
  .. ..$ ID_1     : int [1:22] 1 2 3 4 4 4 4 4 4 4 ...
 ... [lots more stuff]
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr NA
```

Converting it to a regular data frame gives the following:

```
taiwan_map <- fortify(taiwan_shp)
taiwan_map
```

```
     long      lat order  hole piece group id
```

```
120.2390 22.75155     1 FALSE     1   0.1  0
120.2701 22.74135     2 FALSE     1   0.1  0
120.2996 22.70920     3 FALSE     1   0.1  0
...
120.1340 23.61569  1236 FALSE     3  21.3 21
120.1340 23.61597  1237 FALSE     3  21.3 21
120.1365 23.61597  1238 FALSE     3  21.3 21
```

It's actually possible to pass the `SpatialPolygonsDataFrame` object directly to `ggplot()`, which will automatically `fortify()` it:

```
# Send the SpatialPolygonsDataFrame directly to ggplot()
ggplot(taiwan_shp, aes(x=long, y=lat, group=group)) + geom_path()
```

Even though this code is a bit simpler, you may still want to convert it yourself using `fortify()`. This will let you more easily inspect the data structure that is sent to `ggplot()`, or merge the data frame with another data set.

See Also

The shapefile used in this example is not included in the gcookbook package. It and many other shapefiles are available for download.

CHAPTER 14

Output for Presentation

Broadly speaking, visualizations of data serve two purposes: discovery and communication. In the discovery phase, you'll create exploratory graphics, and when you do this, it's important to be able try out different things quickly. In the communication phase, you'll present your graphics to others. When you do that, you'll need to tweak the appearance of the graphics (which I've written about in previous chapters), and you'll usually need to put them somewhere other than on your computer screen. This chapter is about that last part: *saving* your graphics so that they can be presented in documents.

14.1. Outputting to PDF Vector Files

Problem

You want to create a PDF of your plot.

Solution

There are two ways to output to PDF files. One method is to open the PDF graphics device with `pdf()`, make the plots, then close the device with `dev.off()`. This method works for most graphics in R, including base graphics and grid-based graphics like those created by ggplot2 and lattice:

```
# width and height are in inches
pdf("myplot.pdf", width=4, height=4)

# Make plots
plot(mtcars$wt, mtcars$mpg)
print(ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point())

dev.off()
```

If you make more than one plot, each one will go on a separate page in the PDF output. Notice that we called `print()` on the `ggplot` object to make sure that it will be output even when this code is in a script.

The `width` and `height` are in inches, so to specify the dimensions in centimeters, you must do the conversion manually:

```
# 8x8 cm
pdf("myplot.pdf", width=8/2.54, height=8/2.54)
```

If you are creating plots from a script and it throws an error while creating one, R might not reach the call to `dev.off()`, and could be left in a state where the PDF device is still open. When this happens, the PDF file won't open properly until you manually call `dev.off()`.

If you are creating a graph with ggplot2, using `ggsave()` can be a little simpler. It simply saves the last plot created with `ggplot()`:

```
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()

# Default is inches, but you can specify unit
ggsave("myplot.pdf", width=8, height=8, units="cm")
```

With `ggsave()`, you don't need to print the `ggplot` object, and if there is an error while creating or saving the plot, there's no need to manually close the graphic device. `ggsave()` can't be used to make multipage plots, though.

Discussion

PDF files are usually the best option when your goal is to output to printed documents. They work easily with LaTeX and can be used in presentations with Apple's Keynote, but Microsoft programs may have trouble importing them. (See Recipe 14.3 for details on creating vector images that can be imported into Microsoft programs.)

PDF files are also generally smaller than bitmap files such as portable network graphics (PNG) files, because they contain a set of instructions, such as "Draw a line from here to there," instead of information about the color of each pixel. However, there are cases where bitmap files are smaller. For example, if you have a scatter plot that is heavily overplotted, a PDF file can end up much larger than a PNG—even though most of the points are obscured, the PDF file will still contain instructions for drawing each and every point, whereas a bitmap file will not contain the redundant information. See Recipe 5.5 for an example.

See Also

If you want to manually edit the PDF or SVG file, see Recipe 14.4.

14.2. Outputting to SVG Vector Files

Problem

You want to create a scalable vector graphics (SVG) image of your plot.

Solution

SVG files can be created and used in much the same way as PDF files:

```
svg("myplot.svg", width=4, height=4)
plot(...)
dev.off()

# With ggsave()
ggsave("myplot.svg", width=8, height=8, units="cm")
```

Discussion

When it comes to importing images, some programs may handle SVG files better than PDFs, and vice versa. For example, web browsers tend to have better SVG support, while document-creation programs like LaTeX tend to have better PDF support.

14.3. Outputting to WMF Vector Files

Problem

You want to create a Windows metafile (WMF) image of your plot.

Solution

WMF files can be created and used in much the same way as PDF files—but they can only be created on Windows:

```
win.metafile("myplot.wmf", width=4, height=4)
plot(...)
dev.off()

# With ggsave()
ggsave("myplot.wmf", width=8, height=8, units="cm")
```

Discussion

Windows programs such as Microsoft Word and PowerPoint have poor support for importing PDF files, but they natively support WMF. One drawback is that WMF files do not support transparency (alpha).

14.4. Editing a Vector Output File

Problem

You want to open a vector output file for final editing.

Solution

Sometimes you need to make final tweaks to the appearance of a graph for presentation. You can open PDF and SVG files with the excellent free program Inkscape, or with the commercial program Adobe Illustrator.

Discussion

Font support can be a problem when you open a PDF file with Inkscape. Normally, point objects drawn with the PDF device will be written as symbols from the Zapf Dingbats font. This can be problematic if you want to open the file in an editor like Illustrator or Inkscape; for example, points may appear as the letter *q*, as in Figure 14-1, because that is the corresponding letter for a solid bullet in Zapf Dingbats.

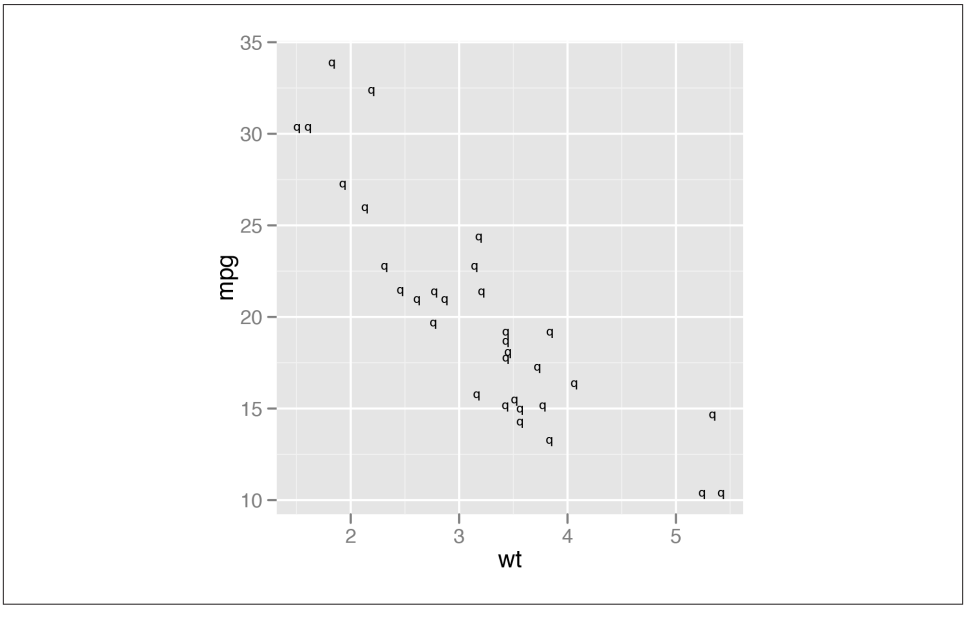

Figure 14-1. Bad conversion of point symbols after opening in Inkscape—also notice that the spacing of the fonts is slightly off

To avoid this problem, set `useDingbats=FALSE`. This will make the circles be drawn as circles instead of as font characters:

```
pdf("myplot.pdf", width=4, height=4, useDingbats=FALSE)

# or
ggsave("myplot.pdf", width=4, height=4, useDingbats=FALSE)
```

Inkscape might have some issues with fonts as well. You may have noticed that the fonts in Figure 14-1 don't look quite right. This is because Inkscape (version 0.48) couldn't find Helvetica, and substituted the font Bitstream Vera Sans instead. A workaround is to copy the Helvetica font file to your personal font library. For example, on Mac OS X, run `cp /System/Library/Fonts/Helvetica.dfont ~/Library/Fonts/` from a Terminal window to do this, then, when it says there is a font conflict, click "Ignore Conflict." After this, Inkscape should properly display the Helvetica font.

14.5. Outputting to Bitmap (PNG/TIFF) Files

Problem

You want to create a bitmap of your plot, writing to a PNG file.

Solution

There are two ways to output to PNG bitmap files. One method is to open the PDF graphics device with `png()`, make the plots, then close the device with `dev.off()`. This method works for most graphics in R, including base graphics and grid-based graphics like those created by ggplot2 and lattice:

```
# width and height are in pixels
png("myplot.png", width=400, height=400)

# Make plot
plot(mtcars$wt, mtcars$mpg)

dev.off()
```

For outputting multiple plots, put `%d` in the filename. This will be replaced with 1, 2, 3, and so on, for each subsequent plot:

```
# width and height are in pixels
png("myplot-%d.png", width=400, height=400)
```

```
plot(mtcars$wt, mtcars$mpg)
print(ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point())

dev.off()
```

Notice that we called `print()` on the `ggplot` object to make sure that it will be output even when this code is in a script.

The `width` and `height` are in pixels, and the default is to output at 72 pixels per inch (ppi). This resolution is suitable for displaying on a screen, but will look pixelated and jagged in print.

For high-quality print output, use at least 300 ppi. Figure 14-2 shows portions of the same plot at different resolutions. In this example, we'll use 300 ppi and create a 4×4-inch PNG file:

```
ppi <- 300
# Calculate the height and width (in pixels) for a 4x4-inch image at 300 ppi
png("myplot.png", width=4*ppi, height=4*ppi, res=ppi)
plot(mtcars$wt, mtcars$mpg)
dev.off()
```

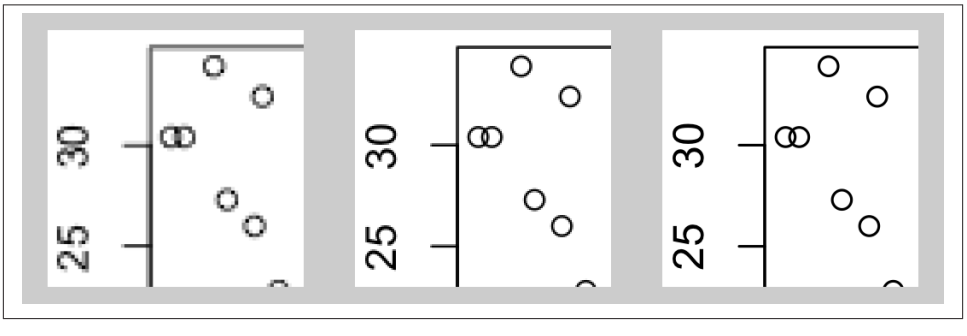

Figure 14-2. From left to right: PNG output at 72, 150, and 300 ppi (actual size)

If you are creating plots from a script and it throws an error while creating one, R might not reach the call to `dev.off()`, and could be left in a state where the PNG device is still open. When this happens, the PNG file won't open properly in a viewing program until you manually call `dev.off()`.

If you are creating a graph with ggplot2, using `ggsave()` can be a little simpler. It simply saves the last plot created with `ggplot()`. You specify the width and height in inches, not pixels, and tell it how many pixels per inch to use:

```
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point()

# Default dimensions are in inches, but you can specify the unit
ggsave("myplot.png", width=8, height=8, unit="cm", dpi=300)
```

With `ggsave()`, you don't need to print the `ggplot` object, and if there is an error while creating or saving the plot there's no need to manually close the graphic device.

Although the argument name is `dpi`, it really controls the *pixels* per inch (ppi), not the *dots* per inch. When a grey pixel is rendered in print, it is output with many smaller dots of black ink—and so print output has more dots per inch than pixels per inch.

Discussion

R supports other bitmap formats, like BMP, TIFF, and JPEG, but there's really not much reason to use them instead of PNG.

The exact appearance of the resulting bitmaps varies from platform to platform. Unlike R's PDF output device, which renders consistently across platforms, the bitmap output devices may render the same plot differently on Windows, Linux, and Mac OS X. There can even be variation within each of these operating systems.

Different platforms will render fonts differently, some platforms will antialias (smooth) lines while others will not, and some platforms support alpha (transparency) while others do not. If your platform lacks support for features like antialiasing and alpha, you can use the `CairoPNG()` device, from the Cairo package:

```
install.packages("Cairo")  # One-time installation
CairoPNG("myplot.png")
plot(...)
dev.off()
```

While `CairoPNG()` does not guarantee identical rendering across platforms (fonts may not be exactly the same), it does support features like antialiasing and alpha.

Changing the resolution affects the size (in pixels) of graphical objects like text, lines, and points. For example, a 6-by-6-inch image at 75 ppi has the same pixel dimensions as a 3-by-3-inch image at 150 ppi, but the appearance will be different, as shown in Figure 14-3. Both of these images are 450×450 pixels. When displayed on a computer screen, they may display at approximately the same size, as they do here.

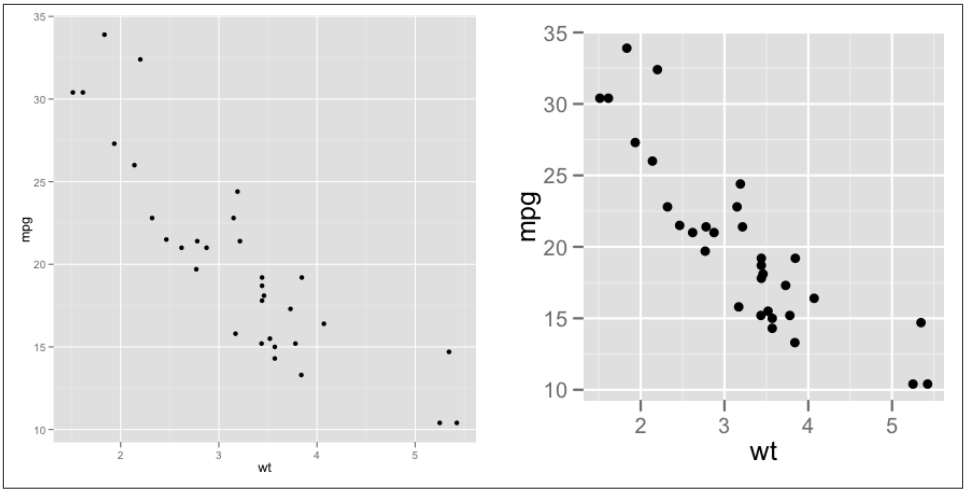

Figure 14-3. Left: 6×6 inch image at 75 ppi; right: 3×3 inch image at 150 ppi

14.6. Using Fonts in PDF Files

Problem

You want to use fonts other than the basic ones provided by R in a PDF file.

Solution

The extrafont package can be used to create PDF files with different fonts.

There are a number of steps involved, beginning with some one-time setup. Download and install Ghostscript, then run the following in R:

```
install.packages("extrafont")
library(extrafont)

# Find and save information about fonts installed on your system
font_import()

# List the fonts
fonts()
```

After the one-time setup is done, there are tasks you need to do in each R session:

```
library(extrafont)
# Register the fonts with R
loadfonts()
```

```
# On Windows, you may need to tell it where Ghostscript is installed
# (adjust the path to match your installation of Ghostscript)
Sys.setenv(R_GSCMD = "C:/Program Files/gs/gs9.05/bin/gswin32c.exe")
```

Finally, you can create a PDF file and embed fonts into it, as in Figure 14-4:

```
library(ggplot2)
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() +
    ggtitle("Title text goes here") +
    theme(text = element_text(size = 16, family="Impact"))

ggsave("myplot.pdf", width=4, height=4)

embed_fonts("myplot.pdf")
```

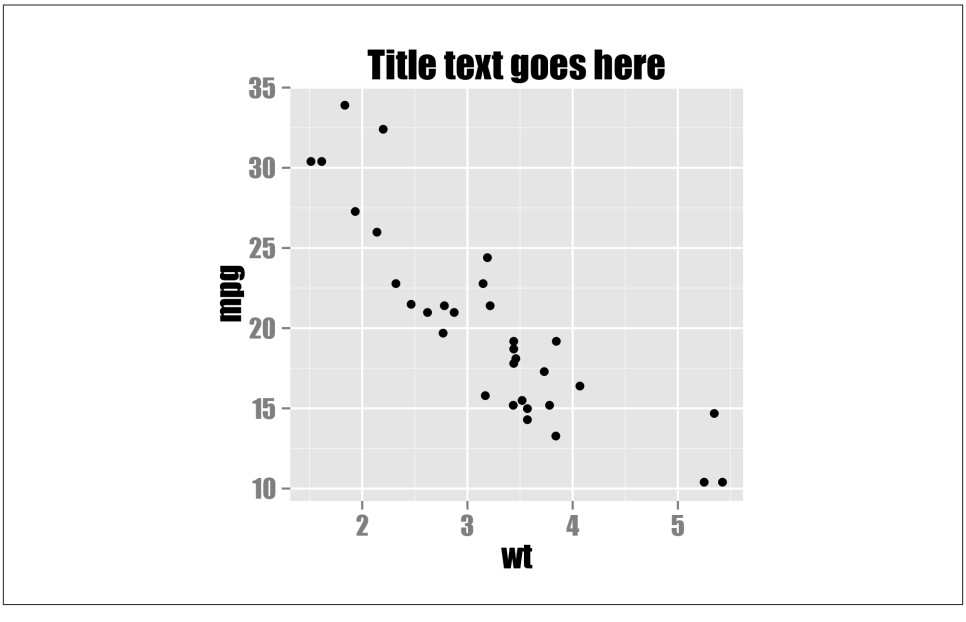

Figure 14-4. PDF output with embedded font Impact

Discussion

Fonts can be difficult to work with in R. Some output devices, such as the on-screen `quartz` device on Mac OS X, can display any font installed on the computer. Other output devices, such as the default `png` device on Windows, aren't able to display system fonts.

On top of this, PDF files have their own quirks when it comes to fonts. The PDF specification has 14 "core" fonts. These are fonts that every PDF renderer has, and they include standards such as Times, Helvetica, and Courier. If you create a PDF with these fonts, any PDF renderer should display it properly.

If you want to use a font that is *not* one of these core fonts, though, there's no guarantee that the PDF renderer on a given device will have that font, so you can't be sure that the font will display properly on another computer or printer. To solve this problem, non-core fonts can be *embedded* into the PDF; in other words, the PDF file can itself contain a copy of the font you want to use.

If you are putting multiple PDF figures in a PDF document, you may want to embed the fonts in the finished document instead of in each figure. This will make the final document smaller, since it will only have the font embedded once, instead of once for each figure.

Embedding fonts with R can be a tricky process, but the `extrafont` package handles many of the ugly details for you.

As of this writing, `extrafont` will only import TrueType (*.ttf*) fonts, but it may support other common formats, such as OpenType (*.otf*), in the future.

See Also

For more on controlling text appearance, see Recipe 9.2.

14.7. Using Fonts in Windows Bitmap or Screen Output

Problem

You are using Windows and want to use fonts other than the basic ones provided by R for bitmap or screen output.

Solution

The `extrafont` package can be used to create bitmap or screen output. The procedure is similar to using `extrafont` with PDF files (Recipe 14.6). The one-time setup is almost the same, except that Ghostscript is not required:

```
install.packages("extrafont")
library(extrafont)

# Find and save information about fonts installed on your system
font_import()

# List the fonts
fonts()
```

After the one-time setup is done, there are tasks you need to do in each R session:

```
library(extrafont)
# Register the fonts for Windows
loadfonts("win")
```

Finally, you can create each output file or display graphs on screen, as in Figure 14-5:

```
library(ggplot2)
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() +
    ggtitle("Title text goes here") +
    theme(text = element_text(size = 16, family="Georgia", face="italic"))

ggsave("myplot.png", width=4, height=4, dpi=300)
```

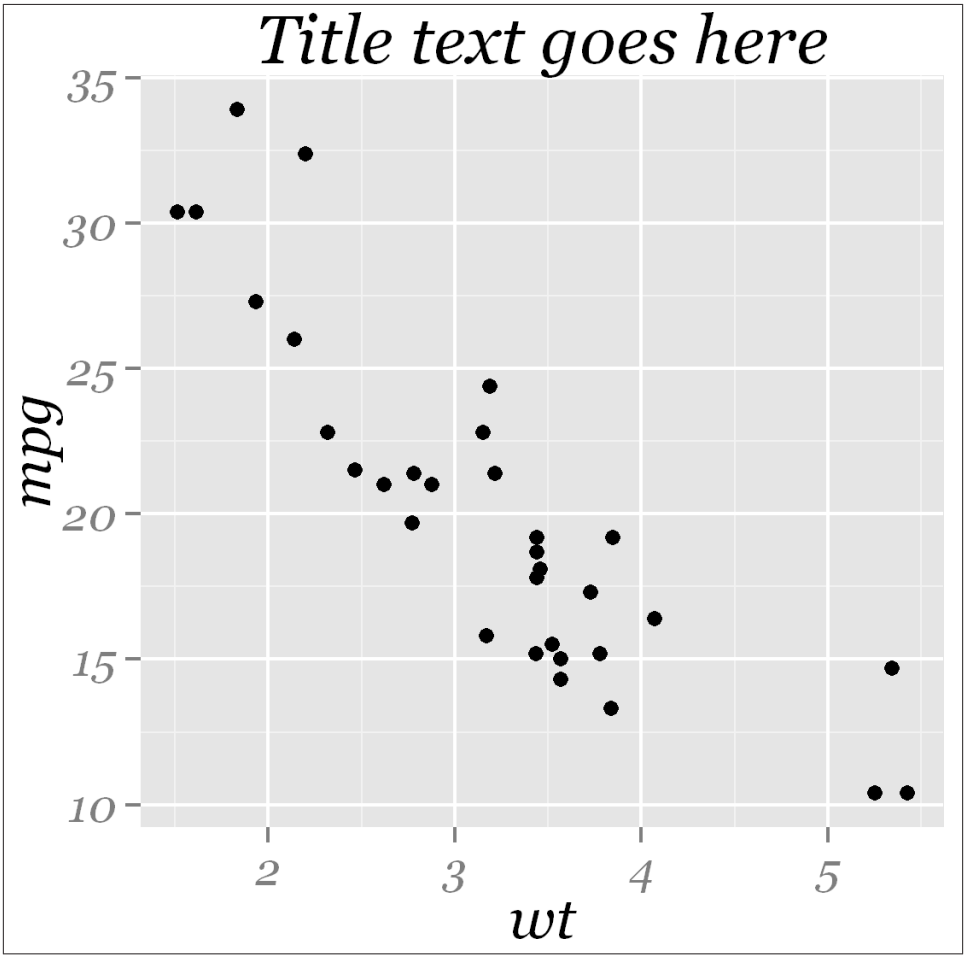

Figure 14-5. PNG output with font Georgia Italic

Discussion

Fonts are handled in a completely different way for bitmaps than they are for PDF files.

On Windows, for bitmap output it is necessary to register each font manually with R (`extrafont` makes this much easier). On Mac OS X and Linux, the fonts should already be available for bitmap output; it isn't necessary to register them manually.

CHAPTER 15
Getting Your Data into Shape

When it comes to making graphs, half the battle occurs before you call any graphing commands. Before you pass your data to the graphing functions, it must first be read in and given the correct structure. The data sets provided with R are ready to use, but when dealing with real-world data, this usually isn't the case: you'll have to clean up and restructure the data before you can visualize it.

Data sets in R are most often stored in data frames. They're typically used as two-dimensional data structures, with each row representing one case and each column representing one variable. Data frames are essentially lists of vectors and factors, all of the same length, where each vector or factor represents one column.

Here's the `heightweight` data set:

```
library(gcookbook) # For the data set
heightweight

sex ageYear ageMonth heightIn weightLb
  f   11.92      143     56.3     85.0
  f   12.92      155     62.3    105.0
 ...
  m   13.92      167     62.0    107.5
  m   12.58      151     59.3     87.0
```

It consists of five columns, with each row representing one case: a set of information about a single person. We can get a clearer idea of how it's structured by using the `str()` function:

```
str(heightweight)

'data.frame':   236 obs. of  5 variables:
 $ sex     : Factor w/ 2 levels "f","m": 1 1 1 1 1 1 1 1 1 1 ...
```

```
$ ageYear : num  11.9 12.9 12.8 13.4 15.9 ...
$ ageMonth: int  143 155 153 161 191 171 185 142 160 140 ...
$ heightIn: num  56.3 62.3 63.3 59 62.5 62.5 59 56.5 62 53.8 ...
$ weightLb: num  85 105 108 92 112 ...
```

The first column, `sex`, is a factor with two levels, `"f"` and `"m"`, and the other four columns are vectors of numbers (one of them, `ageMonth`, is specifically a vector of integers, but for the purposes here, it behaves the same as any other numeric vector).

Factors and character vectors behave similarly in ggplot2—the main difference is that with character vectors, items will be displayed in lexicographical order, but with factors, items will be displayed in the same order as the factor levels, which you can control.

15.1. Creating a Data Frame

Problem

You want to create a data frame from vectors.

Solution

You can put vectors together in a data frame with `data.frame()`:

```
# Two starting vectors
g <- c("A", "B", "C")
x <- 1:3

dat <- data.frame(g, x)
dat
```

```
 g x
 A 1
 B 2
 C 3
```

Discussion

A data frame is essentially a list of vectors and factors. Each vector or factor can be thought of as a column in the data frame.

If your vectors are in a list, you can convert the list to a data frame with the `as.data.frame()` function:

```
lst <- list(group = g, value = x)    # A list of vectors

dat <- as.data.frame(lst)
```

15.2. Getting Information About a Data Structure

Problem

You want to find out information about an object or data structure.

Solution

Use the `str()` function:

```
str(ToothGrowth)
```

```
'data.frame':   60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

This tells us that `ToothGrowth` is a data frame with three columns, `len`, `supp`, and `dose`. `len` and `dose` contain numeric values, while `supp` is a factor with two levels.

Discussion

The `str()` function is very useful for finding out more about data structures. One common source of problems is a data frame where one of the columns is a character vector instead of a factor, or vice versa. This can cause puzzling issues with analyses or graphs.

When you print out a data frame the normal way, by just typing the name at the prompt and pressing Enter, factor and character columns appear exactly the same. The difference will be revealed only when you run `str()` on the data frame, or print out the column by itself:

```
tg <- ToothGrowth
tg$supp <- as.character(tg$supp)

str(tg)
```

```
'data.frame':   60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: chr  "VC" "VC" "VC" "VC" ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

```
# Print out the columns by themselves

# From old data frame (factor)
ToothGrowth$supp
```

```
 [1] VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC
[26] VC VC VC VC VC OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
[51] OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
```

```
Levels: OJ VC

# From new data frame (character)
tg$supp

 [1] "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC"
[16] "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC" "VC"
[31] "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ"
[46] "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ" "OJ"
```

15.3. Adding a Column to a Data Frame

Problem

You want to add a column to a data frame.

Solution

Just assign some value to the new column.

If you assign a single value to the new column, the entire column will be filled with that value. This adds a column named `newcol`, filled with `NA`:

```
data$newcol <- NA
```

You can also assign a vector to the new column:

```
data$newcol <- vec
```

If the length of the vector is less than the number of rows in the data frame, then the vector is repeated to fill all the rows.

Discussion

Each "column" of a data frame is a vector or factor. R handles them slightly differently from standalone vectors, because all the columns in a data frame have the same length.

15.4. Deleting a Column from a Data Frame

Problem

You want to delete a column from a data frame.

Solution

Assign `NULL` to that column:

```
data$badcol <- NULL
```

Discussion

You can also use the `subset()` function and put a `-` (minus sign) in front of the column(s) to drop:

```
# Return data without badcol
data <- subset(data, select = -badcol)

# Exclude badcol and othercol
data <- subset(data, select = c(-badcol, -othercol))
```

See Also

Recipe 15.7 for more on getting a subset of a data frame.

15.5. Renaming Columns in a Data Frame

Problem

You want to rename the columns in a data frame.

Solution

Use the `names(dat) <-` function:

```
names(dat) <- c("name1", "name2", "name3")
```

Discussion

If you want to rename the columns by name:

```
library(gcookbook) # For the data set
names(anthoming)   # Print the names of the columns
```

```
 "angle" "expt"  "ctrl"
```

```
names(anthoming)[names(anthoming) == "ctrl"] <- c("Control")
names(anthoming)[names(anthoming) == "expt"] <- c("Experimental")
names(anthoming)
```

```
 "angle"        "Experimental" "Control"
```

They can also be renamed by numeric position:

```
names(anthoming)[1] <- "Angle"
names(anthoming)
```

```
 "Angle"        "Experimental" "Control"
```

15.6. Reordering Columns in a Data Frame

Problem

You want to change the order of columns in a data frame.

Solution

To reorder columns by their numeric position:

```
dat <- dat[c(1,3,2)]
```

To reorder by column name:

```
dat <- dat[c("col1", "col3", "col2")]
```

Discussion

The previous examples use list-style indexing. A data frame is essentially a list of vectors, and indexing into it as a list will return another data frame. You can get the same effect with matrix-style indexing:

```
library(gcookbook) # For the data set
anthoming
```

```
 angle expt ctrl
   -20    1    0
   -10    7    3
     0    2    3
    10    0    3
    20    0    1
```

```
anthoming[c(1,3,2)]     # List-style indexing
```

```
 angle ctrl expt
   -20    0    1
   -10    3    7
     0    3    2
    10    3    0
    20    1    0
```

```
# Putting nothing before the comma means to select all rows
anthoming[, c(1,3,2)]   # Matrix-style indexing
```

```
 angle ctrl expt
   -20    0    1
   -10    3    7
     0    3    2
    10    3    0
    20    1    0
```

In this case, both methods return the same result, a data frame. However, when retrieving a single column, list-style indexing will return a data frame, while matrix-style indexing will return a vector, unless you use `drop=FALSE`:

```
anthoming[3]     # List-style indexing
```

```
 ctrl
    0
    3
    3
    3
    1
```

```
anthoming[, 3]   # Matrix-style indexing
```

```
 0 3 3 3 1
```

```
anthoming[, 3, drop=FALSE]   # Matrix-style indexing with drop=FALSE
```

```
 ctrl
    0
    3
    3
    3
    1
```

15.7. Getting a Subset of a Data Frame

Problem

You want to get a subset of a data frame.

Solution

Use the `subset()` function. It can be used to pull out rows that satisfy a set of conditions and to select particular columns.

We'll use the `climate` data set for the examples here:

```
library(gcookbook) # For the data set
climate
```

```
   Source Year Anomaly1y Anomaly5y Anomaly10y Unc10y
Berkeley 1800        NA        NA     -0.435  0.505
Berkeley 1801        NA        NA     -0.453  0.493
Berkeley 1802        NA        NA     -0.460  0.486
 ...
 CRUTEM3 2009    0.7343        NA         NA     NA
 CRUTEM3 2010    0.8023        NA         NA     NA
 CRUTEM3 2011    0.6193        NA         NA     NA
```

The following will pull out only rows where `Source` is `"Berkeley"` and only the columns named `Year` and `Anomaly10y`:

```
subset(climate, Source == "Berkeley", select = c(Year, Anomaly10y))
```

```
Year Anomaly10y
1800     -0.435
1801     -0.453
1802     -0.460
 ...
2002      0.856
2003      0.869
2004      0.884
```

Discussion

It is possible to use multiple selection criteria, by using the `|` (OR) and `&` (AND) operators. For example, this will pull out only those rows where `source` is `"Berkeley"`, between the years 1900 and 2000:

```
subset(climate, Source == "Berkeley"  &  Year >= 1900  &  Year <= 2000,
       select = c(Year, Anomaly10y))
```

```
Year Anomaly10y
1900     -0.171
1901     -0.162
1902     -0.177
 ...
1998      0.680
1999      0.734
2000      0.748
```

You can also get a subset of data by indexing into the data frame with square brackets, although this approach is somewhat less elegant. The following code has the same effect as the code we just saw. The part before the comma picks out the rows, and the part after the comma picks out the columns:

```
climate[climate$Source=="Berkeley" & climate$Year >= 1900 & climate$Year <= 2000,
        c("Year", "Anomaly10y")]
```

If you grab just a single column this way, it will be returned as a vector instead of a data frame. To prevent this, use `drop=FALSE`, as in:

```
climate[climate$Source=="Berkeley" & climate$Year >= 1900 & climate$Year <= 2000,
        c("Year", "Anomaly10y"), drop=FALSE]
```

Finally, it's also possible to pick out rows and columns by their numeric position. This gets the second and fifth columns of the first 100 rows:

```
climate[1:100, c(2, 5)]
```

I generally recommend indexing using names rather than numbers when possible. It makes the code easier to understand when you're collaborating with others or when you come back to it months or years after writing it, and it makes the code less likely to break when there are changes to the data, such as when columns are added or removed.

15.8. Changing the Order of Factor Levels

Problem

You want to change the order of levels in a factor.

Solution

The level order can be specified explicitly by passing the factor to `factor()` and specifying `levels`. In this example, we'll create a factor that initially has the wrong ordering:

```
# By default, levels are ordered alphabetically
sizes <- factor(c("small", "large", "large", "small", "medium"))
sizes

small  large  large  small  medium
Levels: large medium small

# Change the order of levels
sizes <- factor(sizes, levels = c("small", "medium", "large"))
sizes

small  large  large  small  medium
Levels: small medium large
```

The order can also be specified with `levels` when the factor is first created.

Discussion

There are two kinds of factors in R: ordered factors and regular factors. In both types, the levels are arranged in *some* order; the difference is that the order is meaningful for an ordered factor, but it is arbitrary for a regular factor—it simply reflects how the data is stored. For graphing data, the distinction between ordered and regular factors is generally unimportant, and they can be treated the same.

The order of factor levels affects graphical output. When a factor variable is mapped to an aesthetic property in ggplot2, the aesthetic adopts the ordering of the factor levels. If a factor is mapped to the x-axis, the ticks on the axis will be in the order of the factor levels, and if a factor is mapped to color, the items in the legend will be in the order of the factor levels.

To reverse the level order, you can use `rev(levels())`:

```
factor(sizes, levels = rev(levels(sizes)))
```

```
small  large  large  small  medium
Levels: small medium large
```

See Also

To reorder a factor based on the value of another variable, see Recipe 15.9.

Reordering factor levels is useful for controlling the order of axes and legends. See Recipes 8.4 and 10.3 for more information.

15.9. Changing the Order of Factor Levels Based on Data Values

Problem

You want to change the order of levels in a factor based on values in the data.

Solution

Use `reorder()` with the factor that has levels to reorder, the values to base the reordering on, and a function that aggregates the values:

```
# Make a copy since we'll modify it
iss <- InsectSprays
iss$spray
```

```
 [1] A A A A A A A A A A A A B B B B B B B B B B B B C C C C C C C C C C C C D D
[39] D D D D D D D D D E E E E E E E E E E E E F F F F F F F F F F F F
Levels: A B C D E F
```

```
iss$spray <- reorder(iss$spray, iss$count, FUN=mean)
iss$spray
```

```
 [1] A A A A A A A A A A A A B B B B B B B B B B B B C C C C C C C C C C C C D D
[39] D D D D D D D D D E E E E E E E E E E E E F F F F F F F F F F F F
attr(,"scores")
       A        B        C        D        E        F
14.500000 15.333333  2.083333  4.916667  3.500000 16.666667
Levels: C E D A B F
```

Notice that the original levels were `ABCDEF`, while the reordered levels are `CEDABF`. The new order is determined by splitting `iss$count` into pieces according to the values in `iss$spray`, and then taking the mean of each group.

Discussion

The usefulness of `reorder()` might not be obvious from just looking at the raw output. Figure 15-1 shows three graphs made with `reorder()`. In these graphs, the order in which the items appear is determined by their values.

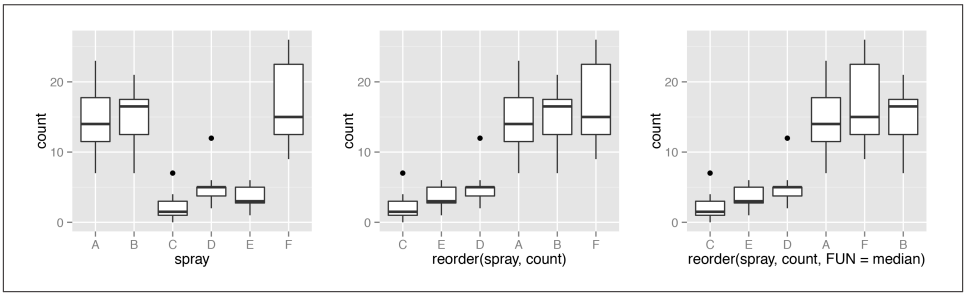

Figure 15-1. Left: original data; middle: reordered by the mean of each group; right: reordered by the median of each group

In the middle graph in Figure 15-1, the boxes are sorted by the mean. The horizontal line that runs across each box represents the *median* of the data. Notice that these values do not increase strictly from left to right. That's because with this particular data set, sorting by the mean gives a different order than sorting by the median. To make the median lines increase from left to right, as in the graph on the right in Figure 15-1, we used the `median()` function in `reorder()`.

See Also

Reordering factor levels is also useful for controlling the order of axes and legends. See Recipes 8.4 and 10.3 for more information.

15.10. Changing the Names of Factor Levels

Problem

You want to change the names of levels in a factor.

Solution

Use `revalue()` or `mapvalues()` from the plyr package:

```
sizes <- factor(c( "small", "large", "large", "small", "medium"))
sizes

small  large  large  small  medium
```

```
Levels: large medium small

levels(sizes)

"large"  "medium" "small"

# With revalue(), pass it a named vector with the mappings
sizes1 <- revalue(sizes, c(small="S", medium="M", large="L"))
sizes1

S L L S M
Levels: L M S

# Can also use quotes -- useful if there are spaces or other strange characters
revalue(sizes, c("small"="S", "medium"="M", "large"="L"))

# mapvalues() lets you use two separate vectors instead of a named vector
mapvalues(sizes, c("small", "medium", "large"), c("S", "M", "L"))
```

Discussion

The `revalue()` and `mapvalues()` functions are convenient, but for a more traditional (and clunky) R method for renaming factor levels, use the `levels()<-` function:

```
sizes <- factor(c( "small", "large", "large", "small", "medium"))

# Index into the levels and rename each one
levels(sizes)[levels(sizes)=="large"]  <- "L"
levels(sizes)[levels(sizes)=="medium"] <- "M"
levels(sizes)[levels(sizes)=="small"]  <- "S"
sizes

S L L S M
Levels: L M S
```

If you are renaming *all* your factor levels, there is a simpler method. You can pass a list to `levels()<-`:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))
levels(sizes) <- list(S="small", M="medium", L="large")
sizes

S L L S M
Levels: L M S
```

With this method, all factor levels must be specified in the list; if any are missing, they will be replaced with `NA`.

It's also possible to rename factor levels by position, but this is somewhat inelegant:

```
# By default, levels are ordered alphabetically
sizes <- factor(c("small", "large", "large", "small", "medium"))
```

```
small  large  large  small  medium
Levels: large medium small

levels(sizes)[1] <- "L"
sizes

small  L      L      small  medium
Levels: L medium small

# Rename all levels at once
levels(sizes) <- c("L", "M", "S")
sizes

[1] S L L S M
Levels: L M S
```

It's safer to rename factor levels by name rather than by position, since you will be less likely to make a mistake (and mistakes here may be hard to detect). Also, if your input data set changes to have more (or fewer) levels, the numeric positions of the existing levels could change, which could cause serious but nonobvious problems for your analysis.

See Also

If, instead of a factor, you have a character vector with items to rename, see Recipe 15.12.

15.11. Removing Unused Levels from a Factor

Problem

You want to remove unused levels from a factor.

Solution

Sometimes, after processing your data you will have a factor that contains levels that are no longer used. Here's an example:

```
sizes <- factor(c("small", "large", "large", "small", "medium"))
sizes <- sizes[1:3]
sizes

small  large  large
Levels: large medium small
```

To remove them, use `droplevels()`:

```
sizes <- droplevels(sizes)
sizes
```

```
small  large  large
Levels: large small
```

Discussion

The `droplevels()` function preserves the order of factor levels.

You can use the `except` argument to keep particular levels.

15.12. Changing the Names of Items in a Character Vector

Problem

You want to change the names of items in a character vector.

Solution

Use `revalue()` or `mapvalues()` from the plyr package:

```
sizes <- c("small", "large", "large", "small", "medium")
sizes
```

```
 "small"  "large"  "large"  "small"  "medium"
```

```
# With revalue(), pass it a named vector with the mappings
sizes1 <- revalue(sizes, c(small="S", medium="M", large="L"))
sizes1
```

```
 "S" "L" "L" "S" "M"
```

```
# Can also use quotes -- useful if there are spaces or other strange characters
revalue(sizes, c("small"="S", "medium"="M", "large"="L"))

# mapvalues() lets you use two separate vectors instead of a named vector
mapvalues(sizes, c("small", "medium", "large"), c("S", "M", "L"))
```

Discussion

A more traditional R method is to use square-bracket indexing to select the items and rename them:

```
sizes <- c("small", "large", "large", "small", "medium")
sizes
```

```
 "small"  "large"  "large"  "small"  "medium"
```

```
sizes[sizes=="small"]  <- "S"
```

```
sizes[sizes=="medium"] <- "M"
sizes[sizes=="large"]  <- "L"

sizes
```

```
 "S" "L" "L" "S" "M"
```

See Also

If, instead of a character vector, you have a factor with levels to rename, see Recipe 15.10.

15.13. Recoding a Categorical Variable to Another Categorical Variable

Problem

You want to recode a categorical variable to another variable.

Solution

For the examples here, we'll use a subset of the `PlantGrowth` data set:

```
# Work on a subset of the PlantGrowth data set
pg <- PlantGrowth[c(1,2,11,21,22), ]
pg
```

```
weight group
  4.17  ctrl
  5.58  ctrl
  4.81  trt1
  6.31  trt2
  5.12  trt2
```

In this example, we'll recode the categorical variable `group` into another categorical variable, `treatment`. If the old value was `"ctrl"`, the new value will be `"No"`, and if the old value was `"trt1"` or `"trt2"`, the new value will be `"Yes"`.

This can be done with the `match()` function:

```
pg <- PlantGrowth

oldvals <- c("ctrl", "trt1", "trt2")
newvals <- factor(c("No",   "Yes",  "Yes"))

pg$treatment <- newvals[ match(pg$group, oldvals) ]
```

It can also be done (more awkwardly) by indexing in the vectors:

```
pg$treatment[pg$group == "ctrl"] <- "no"
pg$treatment[pg$group == "trt1"] <- "yes"
```

```
pg$treatment[pg$group == "trt2"] <- "yes"

# Convert to a factor
pg$treatment <- factor(pg$treatment)
pg
```

```
weight group treatment
  4.17  ctrl        no
  5.58  ctrl        no
  4.81  trt1       yes
  6.31  trt2       yes
  5.12  trt2       yes
```

Here, we combined two of the factor levels and put the result into a new column. If you simply want to rename the levels of a factor, see Recipe 15.10.

Discussion

The coding criteria can also be based on values in multiple columns, by using the `&` and `|` operators:

```
pg$newcol[pg$group == "ctrl"  &  pg$weight <  5] <- "no_small"
pg$newcol[pg$group == "ctrl"  &  pg$weight >= 5] <- "no_large"
pg$newcol[pg$group == "trt1"] <- "yes"
pg$newcol[pg$group == "trt2"] <- "yes"

pg$newcol <- factor(pg$newcol)
pg
```

```
weight group weightcat treatment   newcol
  4.17  ctrl     small        no no_small
  5.58  ctrl     large        no no_large
  4.81  trt1     small       yes      yes
  4.17  trt1     small       yes      yes
  6.31  trt2     large       yes      yes
  5.12  trt2     large       yes      yes
```

It's also possible to combine two columns into one using the `interaction()` function, which appends the values with a `"."` in between. This combines the `weightcat` and `treatment` columns into a new column, `weighttrt`:

```
pg$weighttrt <- interaction(pg$weightcat, pg$treatment)
pg
```

```
weight group weightcat treatment   newcol weighttrt
  4.17  ctrl     small        no no_small  small.no
  5.58  ctrl     large        no no_large  large.no
  4.81  trt1     small       yes      yes small.yes
  4.17  trt1     small       yes      yes small.yes
  6.31  trt2     large       yes      yes large.yes
  5.12  trt2     large       yes      yes large.yes
```

See Also

For more on renaming factor levels, see Recipe 15.10.

See Recipe 15.14 for recoding continuous values to categorical values.

15.14. Recoding a Continuous Variable to a Categorical Variable

Problem

You want to recode a continuous variable to another variable.

Solution

For the examples here, we'll use a subset of the `PlantGrowth` data set.

```
# Work on a subset of the PlantGrowth data set
pg <- PlantGrowth[c(1,2,11,21,22), ]
pg
```

```
weight group
  4.17  ctrl
  5.58  ctrl
  4.81  trt1
  6.31  trt2
  5.12  trt2
```

In this example, we'll recode the continuous variable `weight` into a categorical variable, `wtclass`, using the `cut()` function:

```
pg$wtclass <- cut(pg$weight, breaks = c(0, 5, 6, Inf))
pg
```

```
 weight group wtclass
   4.17  ctrl   (0,5]
   5.58  ctrl   (5,6]
   4.81  trt1   (0,5]
   4.17  trt1   (0,5]
   6.31  trt2 (6,Inf]
   5.12  trt2   (5,6]
```

Discussion

For three categories we specify four bounds, which can include `Inf` and `-Inf`. If a data value falls outside of the specified bounds, it's categorized as `NA`. The result of `cut()` is a factor, and you can see from the example that the factor levels are named after the bounds.

To change the names of the levels, set the `labels`:

```
pg$wtclass <- cut(pg$weight, breaks = c(0, 5, 6, Inf),
                  labels = c("small", "medium", "large"))
pg
```

```
 weight group wtclass
   4.17  ctrl   small
   5.58  ctrl  medium
   4.81  trt1   small
   4.17  trt1   small
   6.31  trt2   large
   5.12  trt2  medium
```

As indicated by the factor levels, the bounds are by default *open* on the left and *closed* on the right. In other words, they don't include the lowest value, but they do include the highest value. For the smallest category, you can have it include both the lower and upper values by setting `include.lowest=TRUE`. In this example, this would result in 0 values going into the `small` category; otherwise, 0 would be coded as `NA`.

If you want the categories to be closed on the left and open on the right, set `right = FALSE`:

```
cut(pg$weight, breaks = c(0, 5, 6, Inf), right = FALSE)
```

See Also

To recode a categorical variable to another categorical variable, see Recipe 15.13.

15.15. Transforming Variables

Problem

You want to transform a variable in a data frame.

Solution

Reference the new column with the `$` operator, and assign some values to it. For this example, we'll use a copy of the `heightweight` data set:

```
library(gcookbook) # For the data set
# Make a copy of the data
hw <- heightweight
hw
```

```
sex ageYear ageMonth heightIn weightLb
  f   11.92      143     56.3     85.0
```

```
  f   12.92      155     62.3    105.0
 ...
  m   13.92      167     62.0    107.5
  m   12.58      151     59.3     87.0
```

This will convert `heightIn` to centimeters and store it in a new column, `heightCm`:

```
hw$heightCm <- hw$heightIn * 2.54
hw
```

```
sex ageYear ageMonth heightIn weightLb heightCm
  f   11.92      143     56.3     85.0  143.002
  f   12.92      155     62.3    105.0  158.242
 ...
  m   13.92      167     62.0    107.5  157.480
  m   12.58      151     59.3     87.0  150.622
```

Discussion

For slightly easier-to-read code, you can use `transform()` or `mutate()` from the plyr package. You only need to specify the data frame once, as the first argument to the function, meaning these provide a cleaner syntax, especially if you are transforming multiple variables:

```
hw <- transform(hw, heightCm = heightIn * 2.54, weightKg = weightLb / 2.204)
library(plyr)
hw <- mutate(hw, heightCm = heightIn * 2.54, weightKg = weightLb / 2.204)
hw
```

```
sex ageYear ageMonth heightIn weightLb heightCm weightKg
  f   11.92      143     56.3     85.0  143.002 38.56624
  f   12.92      155     62.3    105.0  158.242 47.64065
 ...
  m   13.92      167     62.0    107.5  157.480 48.77495
  m   12.58      151     59.3     87.0  150.622 39.47368
```

It is also possible to calculate a new variable based on multiple variables:

```
# These all have the same effect:
hw <- transform(hw, bmi = weightKg / (heightCm / 100)^2)
hw <- mutate(hw, bmi = weightKg / (heightCm / 100)^2)
hw$bmi <- hw$weightKg / (hw$heightCm/100)^2
hw
```

```
sex ageYear ageMonth heightIn weightLb heightCm weightKg      bmi
  f   11.92      143     56.3     85.0  143.002 38.56624 18.85919
  f   12.92      155     62.3    105.0  158.242 47.64065 19.02542
 ...
  m   13.92      167     62.0    107.5  157.480 48.77495 19.66736
  m   12.58      151     59.3     87.0  150.622 39.47368 17.39926
```

The main functional difference between `transform()` and `mutate()` is that `transform()` calculates the new columns simultaneously, while `mutate()` calculates the new columns sequentially, allowing you to base one new column on another new column. Since `bmi` is calculated from `heightCm` and `weightKg`, it is not possible to calculate all of them in a single call to `transform()`; `heightCm` and `weightKg` must be calculated first, and then `bmi`, as shown here.

With `mutate()`, however, we can calculate them all in one go. The following code has the same effect as the previous separate blocks:

```
hw <- heightweight
hw <- mutate(hw,
    heightCm = heightIn * 2.54,
    weightKg = weightLb / 2.204,
    bmi = weightKg / (heightCm / 100)^2)
```

See Also

See Recipe 15.16 for how to perform group-wise transformations on data.

15.16. Transforming Variables by Group

Problem

You want to transform variables by performing operations on groups of data, as specified by a grouping variable.

Solution

Use `ddply()` from the plyr package with the `transform()` function, and specify the operations:

```
library(MASS) # For the data set
library(plyr)
cb <- ddply(cabbages, "Cult", transform, DevWt = HeadWt - mean(HeadWt))
```

```
 Cult Date HeadWt VitC       DevWt
  c39  d16    2.5   51 -0.40666667
  c39  d16    2.2   55 -0.70666667
 ...
  c52  d21    1.5   66 -0.78000000
  c52  d21    1.6   72 -0.68000000
```

Discussion

Let's take a closer look at the `cabbages` data set. It has two grouping variables (factors): `Cult`, which has levels `c39` and `c52`, and `Date`, which has levels `d16`, `d20`, and `d21`. It also has two measured numeric variables, `HeadWt` and `VitC`:

```
cabbages
```

```
 Cult Date HeadWt VitC
  c39  d16    2.5   51
  c39  d16    2.2   55
...
  c52  d21    1.5   66
  c52  d21    1.6   72
```

Suppose we want to find, for each case, the deviation of `HeadWt` from the overall mean. All we have to do is take the overall mean and subtract it from the observed value for each case:

```
transform(cabbages, DevWt = HeadWt - mean(HeadWt))
```

```
 Cult Date HeadWt VitC        DevWt
  c39  d16    2.5   51 -0.093333333
  c39  d16    2.2   55 -0.393333333
 ...
  c52  d21    1.5   66 -1.093333333
  c52  d21    1.6   72 -0.993333333
```

You'll often want to do separate operations like this for each group, where the groups are specified by one or more grouping variables. Suppose, for example, we want to normalize the data within each group by finding the deviation of each case from the mean *within the group*, where the groups are specified by `Cult`. In these cases, we can use `ddply()` from the plyr package with the `transform()` function:

```
library(plyr)
cb <- ddply(cabbages, "Cult", transform, DevWt = HeadWt - mean(HeadWt))
cb
```

```
 Cult Date HeadWt VitC       DevWt
  c39  d16    2.5   51 -0.40666667
  c39  d16    2.2   55 -0.70666667
 ...
  c52  d21    1.5   66 -0.78000000
  c52  d21    1.6   72 -0.68000000
```

First it splits `cabbages` into separate data frames based on the value of `Cult`. There are two levels of `Cult`, `c39` and `c52`, so there are two data frames. It then applies the `transform()` function, with the remaining arguments, to each data frame.

Notice that the call to `ddply()` has all the same parts as the previous call to `transform()`. The only differences are that the parts are slightly rearranged and it adds the splitting variable, in this case, `Cult`.

The before and after results are shown in Figure 15-2:

```
# The data before normalizing
ggplot(cb, aes(x=Cult, y=HeadWt)) + geom_boxplot()

# After normalizing
ggplot(cb, aes(x=Cult, y=DevWt)) + geom_boxplot()
```

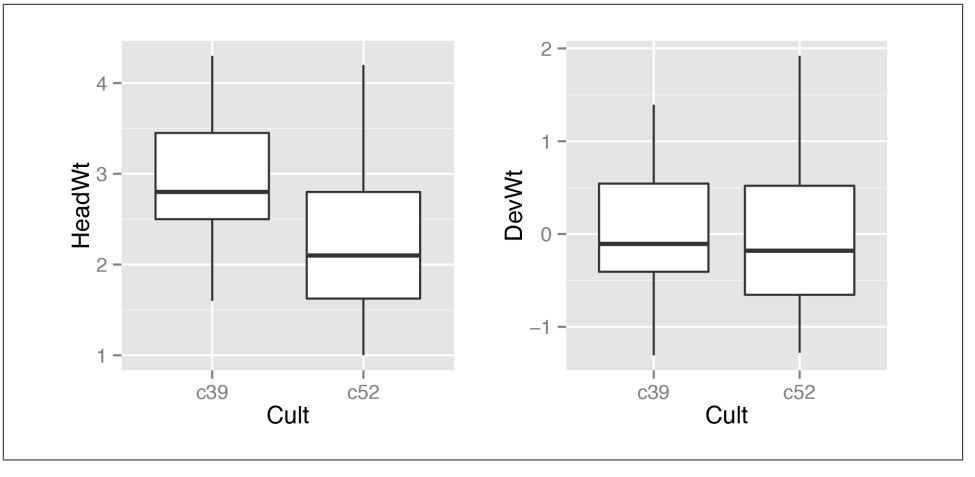

Figure 15-2. Left: before normalizing; right: after normalizing

You can also split the data frame on multiple variables and perform operations on multiple variables. This will split by `Cult` and `Date`, forming a group for each unique combination of the two variables, and then it will calculate the deviation from the mean of `HeadWt` and `VitC` within each group:

```
ddply(cabbages, c("Cult", "Date"), transform,
      DevWt = HeadWt - mean(HeadWt), DevVitC = VitC - mean(VitC))
```

```
 Cult Date HeadWt VitC DevWt DevVitC
  c39  d16    2.5   51 -0.68     0.7
  c39  d16    2.2   55 -0.98     4.7
...
  c52  d21    1.5   66  0.03    -5.8
  c52  d21    1.6   72  0.13     0.2
```

See Also

To summarize data by groups, see Recipe 15.17.

15.17. Summarizing Data by Groups

Problem

You want to summarize your data, based on one or more grouping variables.

Solution

Use `ddply()` from the plyr package with the `summarise()` function, and specify the operations to do:

```
library(MASS) # For the data set
library(plyr)

ddply(cabbages, c("Cult", "Date"), summarise, Weight = mean(HeadWt),
      VitC = mean(VitC))
```

```
 Cult Date Weight VitC
  c39  d16   3.18 50.3
  c39  d20   2.80 49.4
  c39  d21   2.74 54.8
  c52  d16   2.26 62.5
  c52  d20   3.11 58.9
  c52  d21   1.47 71.8
```

Discussion

Let's take a closer look at the `cabbages` data set. It has two factors that can be used as grouping variables: `Cult`, which has levels `c39` and `c52`, and `Date`, which has levels `d16`, `d20`, and `d21`. It also has two numeric variables, `HeadWt` and `VitC`:

```
cabbages
```

```
 Cult Date HeadWt VitC
  c39  d16    2.5   51
  c39  d16    2.2   55
 ...
  c52  d21    1.5   66
  c52  d21    1.6   72
```

Finding the overall mean of `HeadWt` is simple. We could just use the `mean()` function on that column, but for reasons that will soon become clear, we'll use the `summarise()` function instead:

```
library(plyr)
summarise(cabbages, Weight = mean(HeadWt))
```

```
   Weight
 2.593333
```

The result is a data frame with one row and one column, named `Weight`.

Often we want to find information about each subset of the data, as specified by a grouping variable. For example, suppose we want to find the mean of each `Cult` group. To do this, we can use `ddply()` with `summarise()`. Notice how the arguments get shifted around when we use them together:

```
library(plyr)
ddply(cabbages, "Cult", summarise, Weight = mean(HeadWt))
```

```
 Cult   Weight
 c39 2.906667
 c52 2.280000
```

The command first splits the data frame `cabbages` into separate data frames based on the value of `Cult`. There are two levels of `Cult`, `c39` and `c52`, so there are two data frames. It then applies the `summarise()` function to each of these data frames; it calculates `Weight` by taking the `mean()` of the `HeadWt` column in each of the data frames. The resulting summarized data frames each have one row, and `ddply()` puts them back together into one data frame, which is then returned.

Summarizing the data frame by splitting it up with more variables (or columns) is simple: just use a vector that names the additional variables. It's also possible to get more than one summary value by specifying more calculated columns. Here we'll summarize each `Cult` and `Date` group, getting the average of `HeadWt` and `VitC`:

```
ddply(cabbages, c("Cult", "Date"), summarise, Weight = mean(HeadWt),
      VitC = mean(VitC))
```

```
 Cult Date Weight VitC
  c39  d16   3.18 50.3
  c39  d20   2.80 49.4
  c39  d21   2.74 54.8
  c52  d16   2.26 62.5
  c52  d20   3.11 58.9
  c52  d21   1.47 71.8
```

It's possible to do more than take the mean. You may, for example, want to compute the standard deviation and count of each group. To get the standard deviation, use the `sd()` function, and to get a count, use the `length()` function:

```
ddply(cabbages, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt),
      sd = sd(HeadWt),
      n = length(HeadWt))
```

```
Cult Date Weight       sd  n
  c39  d16   3.18 0.9566144 10
  c39  d20   2.80 0.2788867 10
```

```
  c39  d21   2.74 0.9834181 10
  c52  d16   2.26 0.4452215 10
  c52  d20   3.11 0.7908505 10
  c52  d21   1.47 0.2110819 10
```

Other useful functions for generating summary statistics include `min()`, `max()`, and `median()`.

Dealing with NAs

One potential pitfall is that NAs in the data will lead to NAs in the output. Let's see what happens if we sprinkle a few NAs into `HeadWt`:

```
c1 <- cabbages                # Make a copy
c1$HeadWt[c(1,20,45)] <- NA   # Set some values to NA

ddply(c1, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt),
      sd = sd(HeadWt),
      n = length(HeadWt))
```

```
Cult Date Weight        sd  n
  c39  d16     NA        NA 10
  c39  d20     NA        NA 10
  c39  d21   2.74 0.9834181 10
  c52  d16   2.26 0.4452215 10
  c52  d20     NA        NA 10
  c52  d21   1.47 0.2110819 10
```

There are two problems here. The first problem is that `mean()` and `sd()` simply return NA if any of the input values are NA. Fortunately, these functions have an option to deal with this very issue: setting `na.rm=TRUE` will tell them to ignore the NAs.

The second problem is that `length()` counts NAs just like any other value, but since these values represent missing data, they should be excluded from the count. The `length()` function doesn't have an `na.rm` flag, but we can get the same effect by using `sum(!is.na(...))`. The `is.na()` function returns a logical vector: it has a TRUE for each NA item, and a FALSE for all other items. It is inverted by the `!`, and then `sum()` adds up the number of TRUEs. The end result is a count of non-NAs:

```
ddply(c1, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt, na.rm=TRUE),
      sd = sd(HeadWt, na.rm=TRUE),
      n = sum(!is.na(HeadWt)))
```

```
 Cult Date   Weight        sd  n
  c39  d16 3.255556 0.9824855  9
  c39  d20 2.722222 0.1394433  9
```

```
  c39  d21 2.740000 0.9834181 10
  c52  d16 2.260000 0.4452215 10
  c52  d20 3.044444 0.8094923  9
  c52  d21 1.470000 0.2110819 10
```

Missing combinations

If there are any empty combinations of the grouping variables, they will not appear in the summarized data frame. These missing combinations can cause problems when making graphs. To illustrate, we'll remove all entries that have levels `c52` and `d21`. The graph on the left in Figure 15-3 shows what happens when there's a missing combination in a bar graph:

```
# Copy cabbages and remove all rows with both c52 and d21
c2 <- subset(c1, !( Cult=="c52" & Date=="d21" ) )

c2a <- ddply(c2, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt, na.rm=TRUE),
      sd = sd(HeadWt, na.rm=TRUE),
      n = sum(!is.na(HeadWt)))
c2a
```

```
 Cult Date   Weight        sd  n
  c39  d16 3.255556 0.9824855  9
  c39  d20 2.722222 0.1394433  9
  c39  d21 2.740000 0.9834181 10
  c52  d16 2.260000 0.4452215 10
  c52  d20 3.044444 0.8094923  9
```

```
# Make the graph
ggplot(c2a, aes(x=Date, fill=Cult, y=Weight)) + geom_bar(position="dodge")
```

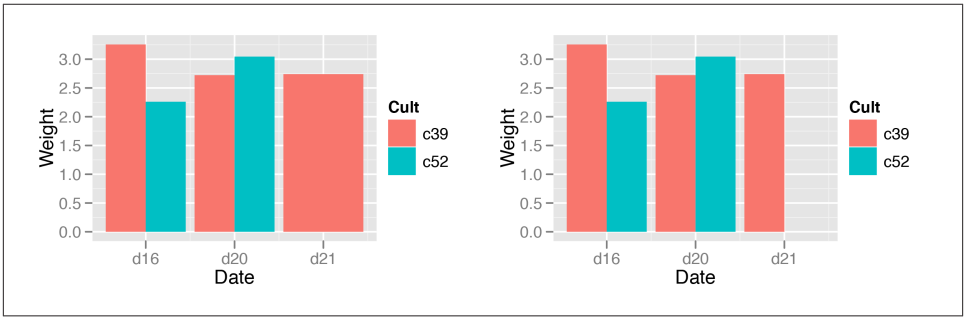

Figure 15-3. Left: bar graph with a missing combination; right: with missing combination filled

To fill in the missing combination (Figure 15-3, right), give `ddply()` the `.drop=FALSE` flag:

```
c2b <- ddply(c2, c("Cult", "Date"), .drop=FALSE, summarise,
      Weight = mean(HeadWt, na.rm=TRUE),
      sd = sd(HeadWt, na.rm=TRUE),
      n = sum(!is.na(HeadWt)))
c2b
```

```
 Cult Date   Weight        sd  n
  c39  d16 3.255556 0.9824855  9
  c39  d20 2.722222 0.1394433  9
  c39  d21 2.740000 0.9834181 10
  c52  d16 2.260000 0.4452215 10
  c52  d20 3.044444 0.8094923  9
  c52  d21      NaN        NA  0
```

```
# Make the graph
ggplot(c2b, aes(x=Date, fill=Cult, y=Weight)) + geom_bar(position="dodge")
```

See Also

If you want to calculate standard error and confidence intervals, see Recipe 15.18.

See Recipe 6.8 for an example of using `stat_summary()` to calculate means and overlay them on a graph.

To perform transformations on data by groups, see Recipe 15.16.

15.18. Summarizing Data with Standard Errors and Confidence Intervals

Problem

You want to summarize your data with the standard error of the mean and/or confidence intervals.

Solution

Getting the standard error of the mean involves two steps: first get the standard deviation and count for each group, then use those values to calculate the standard error. The standard error for each group is just the standard deviation divided by the square root of the sample size:

```
library(MASS) # For the data set
library(plyr)

ca <- ddply(cabbages, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt, na.rm=TRUE),
      sd = sd(HeadWt, na.rm=TRUE),
      n = sum(!is.na(HeadWt)),
```

```
      se = sd/sqrt(n))

ca

 Cult Date Weight        sd  n         se
  c39  d16   3.18 0.9566144 10 0.30250803
  c39  d20   2.80 0.2788867 10 0.08819171
  c39  d21   2.74 0.9834181 10 0.31098410
  c52  d16   2.26 0.4452215 10 0.14079141
  c52  d20   3.11 0.7908505 10 0.25008887
  c52  d21   1.47 0.2110819 10 0.06674995
```

In versions of plyr before 1.8, `summarise()` created all the new columns simultaneously, so you would have to create the `se` column separately, after creating the `sd` and `n` columns.

Discussion

Another method is to calculate the standard error in the call `ddply`. It's not possible to refer to the `sd` and `n` columns inside of the `ddply` call, so we'll have to recalculate them to get `se`. This will do the same thing as the two-step version shown previously:

```
ddply(cabbages, c("Cult", "Date"), summarise,
      Weight = mean(HeadWt, na.rm=TRUE),
      sd = sd(HeadWt, na.rm=TRUE),
      n = sum(!is.na(HeadWt)),
      se = sd / sqrtn) )
```

Confidence Intervals

Confidence intervals are calculated using the standard error of the mean and the degrees of freedom. To calculate a confidence interval, use the `qt()` function to get the quantile, then multiply that by the standard error. The `qt()` function will give quantiles of the *t*-distribution when given a probability level and degrees of freedom. For a 95% confidence interval, use a probability level of `.975`; for the bell-shaped *t*-distribution, this will in essence cut off 2.5% of the area under the curve at either end. The degrees of freedom equal the sample size minus one.

This will calculate the multiplier for each group. There are six groups and each has the same number of observations (10), so they will all have the same multiplier:

```
ciMult <- qt(.975, ca$n-1)
ciMult

# 2.262157 2.262157 2.262157 2.262157 2.262157 2.262157
```

Now we can multiply that vector by the standard error to get the 95% confidence interval:

```
ca$ci <- ca$se * ciMult
```

```
 Cult Date Weight        sd  n         se        ci
  c39  d16   3.18 0.9566144 10 0.30250803 0.6843207
  c39  d20   2.80 0.2788867 10 0.08819171 0.1995035
  c39  d21   2.74 0.9834181 10 0.31098410 0.7034949
  c52  d16   2.26 0.4452215 10 0.14079141 0.3184923
  c52  d20   3.11 0.7908505 10 0.25008887 0.5657403
  c52  d21   1.47 0.2110819 10 0.06674995 0.1509989
```

We could have done this all in one line, like this:

```
ca$ci95 <- ca$se * qt(.975, ca$n)
```

For a 99% confidence interval, use `.995`.

Error bars that represent the standard error of the mean and confidence intervals serve the same general purpose: to give the viewer an idea of how good the estimate of the population mean is. The standard error is the standard deviation of the sampling distribution. Confidence intervals are easier to interpret. Very roughly, a 95% confidence interval means that there's a 95% chance that the true population mean is within the interval (actually, it doesn't mean this at all, but this seemingly simple topic is way too complicated to cover here; if you want to know more, read up on Bayesian statistics).

This function will perform all the steps of calculating the standard deviation, count, standard error, and confidence intervals. It can also handle `NA`s and missing combinations, with the `na.rm` and `.drop` options. By default, it provides a 95% confidence interval, but this can be set with the `conf.interval` argument:

```
summarySE <- function(data=NULL, measurevar, groupvars=NULL,
                      conf.interval=.95, na.rm=FALSE, .drop=TRUE) {
    require(plyr)

    # New version of length that can handle NAs: if na.rm==T, don't count them
    length2 <- function (x, na.rm=FALSE) {
        if (na.rm) sum(!is.na(x))
        else       length(x)
    }

    # This does the summary
    datac <- ddply(data, groupvars, .drop=.drop,
                   .fun = function(xx, col, na.rm) {
                           c( n    = length2(xx[,col], na.rm=na.rm),
                              mean = mean   (xx[,col], na.rm=na.rm),
                              sd   = sd     (xx[,col], na.rm=na.rm)
                              )
                          },
                    measurevar,
                    na.rm
             )
```

```
    # Rename the "mean" column
    datac <- rename(datac, c("mean" = measurevar))

    datac$se <- datac$sd / sqrt(datac$n)  # Calculate standard error of the mean

    # Confidence interval multiplier for standard error
    # Calculate t-statistic for confidence interval:
    # e.g., if conf.interval is .95, use .975 (above/below), and use
    #  df=n-1, or if n==0, use df=0
    ciMult <- qt(conf.interval/2 + .5, datac$n-1)
    datac$ci <- datac$se * ciMult

    return(datac)
}
```

The following usage example has a 99% confidence interval and handles NAs and missing combinations:

```
# Remove all rows with both c52 and d21
c2 <- subset(cabbages, !( Cult=="c52" & Date=="d21" ) )

# Set some values to NA
c2$HeadWt[c(1,20,45)] <- NA


summarySE(c2, "HeadWt", c("Cult", "Date"), conf.interval=.99,
          na.rm=TRUE, .drop=FALSE)

 Cult Date  n   HeadWt        sd         se        ci
  c39  d16  9 3.255556 0.9824855 0.32749517 1.0988731
  c39  d20  9 2.722222 0.1394433 0.04648111 0.1559621
  c39  d21 10 2.740000 0.9834181 0.31098410 1.0106472
  c52  d16 10 2.260000 0.4452215 0.14079141 0.4575489
  c52  d20  9 3.044444 0.8094923 0.26983077 0.9053867
  c52  d21  0      NaN        NA         NA        NA
Warning message:
In qt(p, df, lower.tail, log.p) : NaNs produced
```

It will give this warning message when there are missing combinations. This isn't a problem; it just indicates that it couldn't calculate a quantile for a group with no observations.

See Also

See Recipe 7.7 to use the values calculated here to add error bars to a graph.

15.19. Converting Data from Wide to Long

Problem

You want to convert a data frame from "wide" format to "long" format.

Solution

Use `melt()` from the `reshape2` package. In the `anthoming` data set, for each angle, there are two measurements: one column contains measurements in the experimental condition and the other contains measurements in the control condition:

```
library(gcookbook) # For the data set
anthoming
```

```
 angle expt ctrl
   -20    1    0
   -10    7    3
     0    2    3
    10    0    3
    20    0    1
```

We can reshape the data so that all the measurements are in one column. This will put the values from `expt` and `ctrl` into one column, and put the names into a different column:

```
library(reshape2)
melt(anthoming, id.vars="angle", variable.name="condition", value.name="count")
```

```
 angle condition count
   -20      expt     1
   -10      expt     7
     0      expt     2
    10      expt     0
    20      expt     0
   -20      ctrl     0
   -10      ctrl     3
     0      ctrl     3
    10      ctrl     3
    20      ctrl     1
```

This data frame represents the same information as the original one, but it is structured in a way that is more conducive to some analyses.

Discussion

In the source data, there are *ID* variables and *measure* variables. The ID variables are those that specify which values go together. In the source data, the first row holds measurements for when `angle` is –20. In the output data frame, the two measurements, for `expt` and `ctrl`, are no longer in the same row, but we can still tell that they belong together because they have the same value of `angle`.

The measure variables are by default all the non-ID variables. The names of these variables are put into a new column specified by `variable.name`, and the values are put into a new column specified by `value.name`.

If you don't want to use all the non-ID columns as measure variables, you can specify `measure.vars`. For example, in the `drunk` data set, we can use just the `0-29` and `30-39` groups:

```
drunk
```

```
    sex 0-29 30-39 40-49 50-59 60+
   male  185   207   260   180  71
 female    4    13    10     7  10
```

```
melt(drunk, id.vars="sex", measure.vars=c("0-29", "30-39"),
     variable.name="age", value.name="count")
```

```
    sex   age count
   male  0-29   185
 female  0-29     4
   male 30-39   207
 female 30-39    13
```

It's also possible to use more than one column as the ID variables:

```
plum_wide
```

```
 length      time dead alive
   long   at_once   84   156
   long in_spring  156    84
  short   at_once  133   107
  short in_spring  209    31
```

```
melt(plum_wide, id.vars=c("length","time"), variable.name="survival",
                           value.name="count")
```

```
 length      time survival count
   long   at_once     dead    84
   long in_spring     dead   156
  short   at_once     dead   133
  short in_spring     dead   209
```

```
 long   at_once    alive   156
 long in_spring    alive    84
short   at_once    alive   107
short in_spring    alive    31
```

Some data sets don't come with a column with an ID variable. For example, in the `corneas` data set, each row represents one pair of measurements, but there is no ID variable. Without an ID variable, you won't be able to tell how the values are meant to be paired together. In these cases, you can add an ID variable before using `melt()`:

```
# Make a copy of the data
co <- corneas
co
```

```
 affected notaffected
      488         484
      478         478
      480         492
      426         444
      440         436
      410         398
      458         464
      460         476
```

```
# Add an ID column
co$id <- 1:nrow(co)

melt(co, id.vars="id", variable.name="eye", value.name="thickness")
```

```
 id         eye thickness
  1    affected       488
  2    affected       478
  3    affected       480
  4    affected       426
  5    affected       440
  6    affected       410
  7    affected       458
  8    affected       460
  1 notaffected       484
  2 notaffected       478
  3 notaffected       492
  4 notaffected       444
  5 notaffected       436
  6 notaffected       398
  7 notaffected       464
  8 notaffected       476
```

Having numeric values for the ID variable may be problematic for subsequent analyses, so you may want to convert `id` to a character vector with `as.character()`, or a factor with `factor()`.

See Also

See Recipe 15.20 to do conversions in the other direction, from long to wide.

See the `stack()` function for another way of converting from wide to long.

15.20. Converting Data from Long to Wide

Problem

You want to convert a data frame from "long" format to "wide" format.

Solution

Use the `dcast()` function from the `reshape2` package. In this example, we'll use the `plum` data set, which is in a long format:

```
library(gcookbook) # For the data set
plum
```

```
 length      time survival count
   long   at_once     dead    84
   long in_spring     dead   156
  short   at_once     dead   133
  short in_spring     dead   209
   long   at_once    alive   156
   long in_spring    alive    84
  short   at_once    alive   107
  short in_spring    alive    31
```

The conversion to wide format takes each unique value in one column and uses those values as headers for new columns, then uses another column for source values. For example, we can "move" values in the `survival` column to the top and fill them with values from `count`:

```
library(reshape2)
dcast(plum, length + time ~ survival, value.var="count")
```

```
 length      time dead alive
   long   at_once   84   156
   long in_spring  156    84
  short   at_once  133   107
  short in_spring  209    31
```

Discussion

The `dcast()` function requires you to specify the *ID* variables (those that remain in columns) and the *variable* variables (those that get "moved to the top"). This is done with a formula where the ID variables are before the tilde (`~`) and the variable variables are after it.

In the preceding example, there are two ID variables and one variable variable. In the next one, there is one ID variable and two variable variables. When there is more than one variable variable, the values are combined with an underscore:

```
dcast(plum, time ~ length + survival, value.var="count")
```

```
          time long_dead long_alive short_dead short_alive
     at_once        84        156        133         107
   in_spring       156         84        209          31
```

See Also

See Recipe 15.19 to do conversions in the other direction, from wide to long.

See the `unstack()` function for another way of converting from long to wide.

15.21. Converting a Time Series Object to Times and Values

Problem

You have a time series object that you wish to convert to numeric vectors representing the time and values at each time.

Solution

Use the `time()` function to get the time for each observation, then convert the times and values to numeric vectors with `as.numeric()`:

```
# Look at nhtemp Time Series object
nhtemp
```

```
Time Series:
Start = 1912
End = 1971
Frequency = 1
 [1] 49.9 52.3 49.4 51.1 49.4 47.9 49.8 50.9 49.3 51.9 50.8 49.6 49.3 50.6 48.4
[16] 50.7 50.9 50.6 51.5 52.8 51.8 51.1 49.8 50.2 50.4 51.6 51.8 50.9 48.8 51.7
[31] 51.0 50.6 51.7 51.5 52.1 51.3 51.0 54.0 51.4 52.7 53.1 54.6 52.0 52.0 50.9
[46] 52.6 50.2 52.6 51.6 51.9 50.5 50.9 51.7 51.4 51.7 50.8 51.9 51.8 51.9 53.0
```

```
# Get times for each observation
as.numeric(time(nhtemp))
```

```
 [1] 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926
[16] 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941
[31] 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956
[46] 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971
```

```
# Get value of each observation
as.numeric(nhtemp)
```

```
 [1] 49.9 52.3 49.4 51.1 49.4 47.9 49.8 50.9 49.3 51.9 50.8 49.6 49.3 50.6 48.4
[16] 50.7 50.9 50.6 51.5 52.8 51.8 51.1 49.8 50.2 50.4 51.6 51.8 50.9 48.8 51.7
[31] 51.0 50.6 51.7 51.5 52.1 51.3 51.0 54.0 51.4 52.7 53.1 54.6 52.0 52.0 50.9
[46] 52.6 50.2 52.6 51.6 51.9 50.5 50.9 51.7 51.4 51.7 50.8 51.9 51.8 51.9 53.0
```

```
# Put them in a data frame
nht <- data.frame(year=as.numeric(time(nhtemp)), temp=as.numeric(nhtemp))
nht
```

```
 year temp
 1912 49.9
 1913 52.3
 ...
 1970 51.9
 1971 53.0
```

Discussion

Time series objects efficiently store information when there are observations at regular time intervals, but for use with ggplot2, they need to be converted to a format that separately represents times and values for each observation.

Some time series objects are cyclical. The `presidents` data set, for example, contains four observations per year, one for each quarter:

```
presidents
```

```
     Qtr1 Qtr2 Qtr3 Qtr4
1945   NA   87   82   75
1946   63   50   43   32
1947   35   60   54   55
 ...
1972   49   61   NA   NA
1973   68   44   40   27
1974   28   25   24   24
```

To convert it to a two-column data frame with one column representing the year with fractional values, we can do the same as before:

```
pres_rating <- data.frame(
    year     = as.numeric(time(presidents)),
```

```
    rating  = as.numeric(presidents)
)
pres_rating
```

```
    year rating
1945.00     NA
1945.25     87
1945.50     82
...
1974.25     25
1974.50     24
1974.75     24
```

It is also possible to store the year and quarter in separate columns, which may be useful in some visualizations:

```
pres_rating2 <- data.frame(
    year    = as.numeric(floor(time(presidents))),
    quarter = as.numeric(cycle(presidents)),
    rating  = as.numeric(presidents)
)
pres_rating2
```

```
 year quarter rating
 1945       1     NA
 1945       2     87
 1945       3     82
 ...
 1974       2     25
 1974       3     24
 1974       4     24
```

See Also

The zoo package is also useful for working with time series objects.

APPENDIX A

Introduction to ggplot2

Most of the recipes in this book involve the ggplot2 package, written by Hadley Wickham. ggplot2 has only been around for a few years, but in that short time it has attracted many users in the R community because of its versatility, clear and consistent interface, and beautiful output.

ggplot2 takes a different approach to graphics than other graphing packages in R. It gets its name from Leland Wilkinson's *grammar of graphics*, which provides a formal, structured perspective on how to describe data graphics.

Even though this book deals largely with ggplot2, I don't mean to say that it's the be-all and end-all of graphics in R. For example, I sometimes find it faster and easier to inspect and explore data with R's base graphics, especially when the data isn't already structured properly for use with ggplot2. There are some things that ggplot2 can't do, or can't do as well as other graphing packages. There are other things that ggplot2 can do, but that specialized packages are better suited to handling. For most purposes, though, I believe that ggplot2 gives the best return on time invested, and it provides beautiful, publication-ready results.

Another excellent package for general-purpose graphs is lattice, by Deepyan Sarkar, which is an implementation of *trellis* graphics. It is included as part of the base installation of R.

If you want a deeper understanding of ggplot2, read on!

Background

In a data graphic, there is a mapping (or correspondence) from properties of the data to visual properties in the graphic. The data properties are typically numerical or categorical values, while the visual properties include the *x* and *y* positions of points, colors of lines, heights of bars, and so on. A data visualization that didn't map the data to visual

properties wouldn't be a data visualization. On the surface, representing a number with an *x* coordinate may seem very different from representing a number with a color of a point, but at an abstract level, they are the same. Everyone who has made data graphics has at least an implicit understanding of this. For most of us, that's where our understanding remains.

In the grammar of graphics, this deep similarity is not just recognized, but made central. In R's base graphics functions, each mapping of data properties to visual properties is its own special case, and changing the mappings may require restructuring your data, issuing completely different graphing commands, or both.

To illustrate, I'll show a graph made from the `simpledat` data set from the gcookbook package:

```
library(gcookbook) # For the data set
simpledat
```

```
   A1 A2 A3
B1 10  7 12
B2  9 11  6
```

This will make a simple grouped bar graph, with the As going along the x-axis and the bars grouped by the Bs (Figure A-1):

```
barplot(simpledat, beside=TRUE)
```

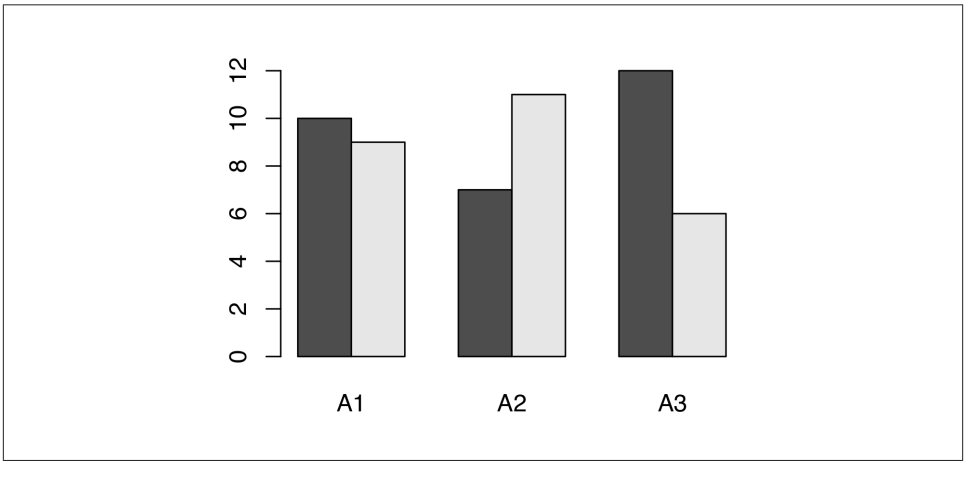

Figure A-1. A bar graph made with barplot()

One thing we might want to do is switch things up so the Bs go along the x-axis and the As are used for grouping. To do this, we need to restructure the data by transposing the matrix:

```
t(simpledat)
```

```
   B1 B2
A1 10  9
A2  7 11
A3 12  6
```

With the restructured data, we can create the graph the same way as before (Figure A-2):

```
barplot(t(simpledat), beside=TRUE)
```

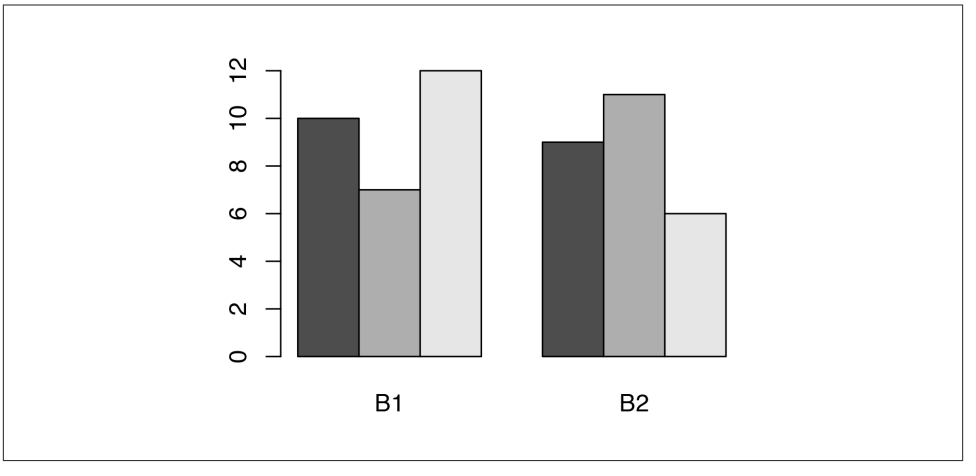

Figure A-2. A bar graph with transposed data

Another thing we might want to do is to represent the data with lines instead of bars, as shown in Figure A-3. To do this with base graphics, we need to use a completely different set of commands. First we call `plot()`, which tells R to create a new graph and draw a line for one row of data. Then we tell it to draw a second row with `lines()`:

```
plot(simpledat[1,], type="l")
lines(simpledat[2,], type="l", col="blue")
```

The resulting graph has a few quirks. The second (blue) line runs below the visible range, because the *y* range was set only for the first line, when the `plot()` function was called. Additionally, the x-axis is numbered instead of categorical.

Now let's take a look at the corresponding code and graphs with ggplot2. With ggplot2, the structure of the data is always the same: it requires a data frame in "long" format, as opposed to the "wide" format used previously. When the data is in long format, each row represents one item. Instead of having their groups determined by their *positions* in the matrix, the items have their groups specified in a separate column. Here is `simpledat`, converted to long format:

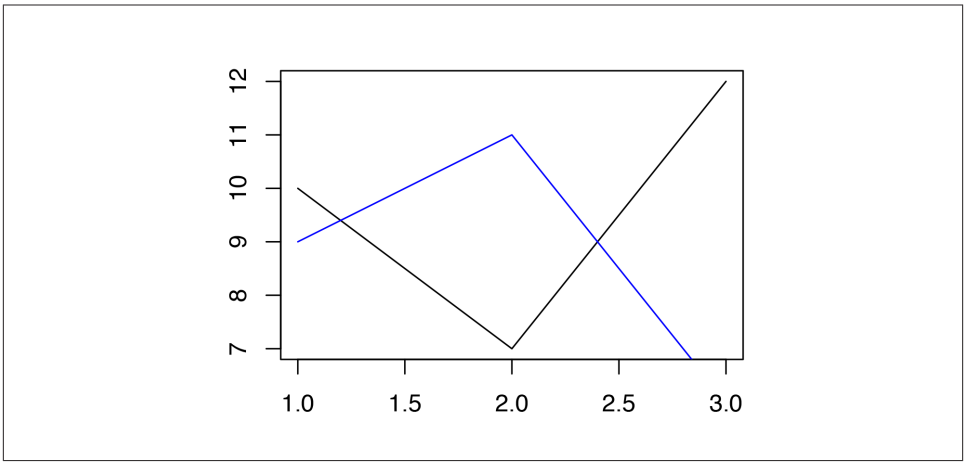

Figure A-3. A line graph made with plot() and lines()

```
simpledat_long
```

```
 Aval Bval value
   A1   B1    10
   A1   B2     9
   A2   B1     7
   A2   B2    11
   A3   B1    12
   A3   B2     6
```

This represents the same information, but with a different structure. There are advantages and disadvantages to the long format, but on the whole, I find that it makes things simpler when dealing with complicated data sets. See Recipes 15.19 and 15.20 for information about converting between wide and long data formats.

To make the first grouped bar graph (Figure A-4), we first have to load the ggplot2 library. Then we tell it to map `Aval` to the *x* position with `x=Aval`, and `Bval` to the fill color with `fill=Bval`. This will make the As run along the x-axis and the Bs determine the grouping. We also tell it to map `value` to the *y* position, or height, of the bars, with `y=value`. Finally, we tell it to draw bars with `geom_bar()` (don't worry about the other details yet; we'll get to those later):

```
library(ggplot2)
ggplot(simpledat_long, aes(x=Aval, y=value, fill=Bval)) +
    geom_bar(stat="identity", position="dodge")
```

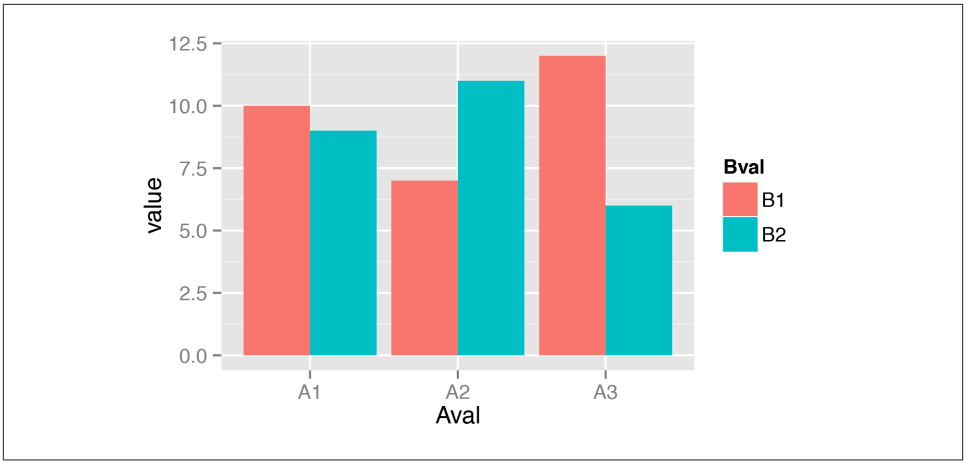

Figure A-4. A bar graph made with ggplot() and geom_bar()

To switch things so that the Bs go along the x-axis and the As determine the grouping (Figure A-5), we simply swap the mapping specification, with `x=Bval` and `fill=Aval`. Unlike with base graphics, we don't have to change the data; we just change the commands for making the graph:

```
ggplot(simpledat_long, aes(x=Bval, y=value, fill=Aval)) +
    geom_bar(stat="identity", position="dodge")
```

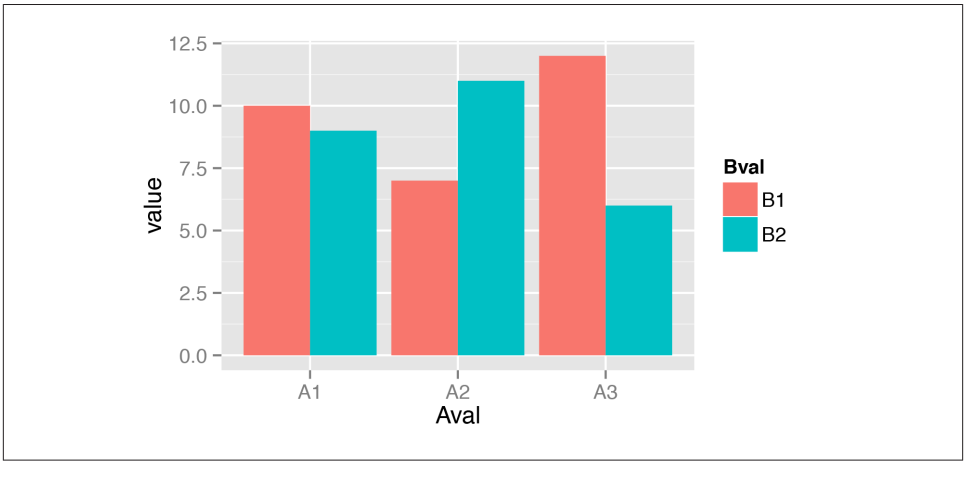

Figure A-5. Bar graph of the same data, but with x and fill mappings switched

You may have noticed that with ggplot2, components of the plot are combined with the + operator. You can gradually build up a ggplot object by adding components to it, then, when you're all done, you can tell it to print.

To change it to a line graph (Figure A-6), we change `geom_bar()` to `geom_line()`. We'll also map `Bval` to the *line* color, with `colour`, instead of the *fill* colour (note the British spelling—the author of ggplot2 is a Kiwi). Again, don't worry about the other details yet:

```
ggplot(simpledat_long, aes(x=Aval, y=value, colour=Bval, group=Bval)) +
    geom_line()
```

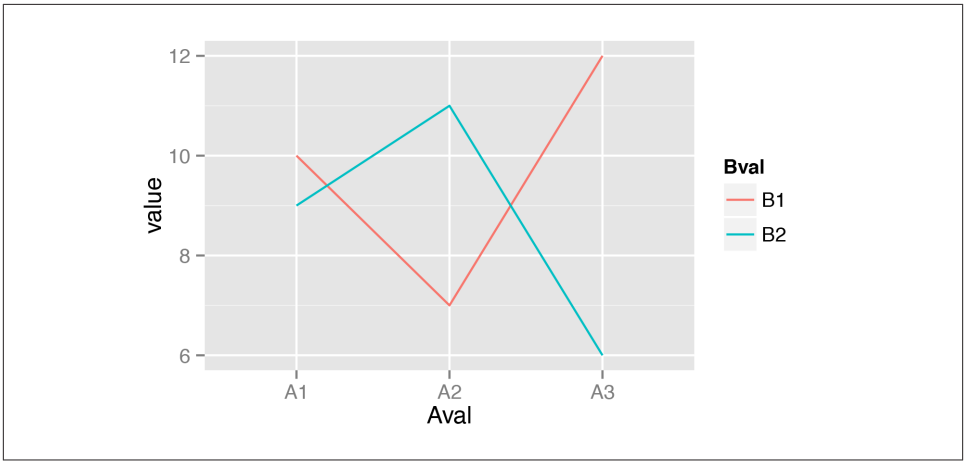

Figure A-6. A line graph made with ggplot() and geom_line()

With base graphics, we had to use completely different commands to make a line graph instead of a bar graph. With ggplot2, we just changed the *geom* from bars to lines. The resulting graph also has important differences from the base graphics version: the *y* range is automatically adjusted to fit all the data because all the lines are drawn together instead of one at a time, and the x-axis remains categorical instead of being converted to a numeric axis. The ggplot2 graphs also have automatically generated legends.

Some Terminology and Theory

Before we go any further, it'll be helpful to define some of the terminology used in ggplot2:

- The *data* is what we want to visualize. It consists of *variables*, which are stored as columns in a data frame.
- *Geoms* are the geometric objects that are drawn to represent the data, such as bars, lines, and points.
- Aesthetic attributes, or *aesthetics*, are visual properties of geoms, such as *x* and *y* position, line color, point shapes, etc.
- There are *mappings* from data values to aesthetics.
- *Scales* control the mapping from the values in the data space to values in the aesthetic space. A continuous *y* scale maps larger numerical values to vertically higher positions in space.
- *Guides* show the viewer how to map the visual properties back to the data space. The most commonly used guides are the tick marks and labels on an axis.

Here's an example of how a typical mapping works. You have *data*, which is a set of numerical or categorical values. You have *geoms* to represent each observation. You have an *aesthetic*, such as *y* (vertical) position. And you have a *scale*, which defines the mapping from the data space (numeric values) to the aesthetic space (vertical position). A typical linear *y*-scale might map the value 0 to the baseline of the graph, 5 to the middle, and 10 to the top. A logarithmic *y* scale would place them differently.

These aren't the only kinds of data and aesthetic spaces possible. In the abstract grammar of graphics, the data and aesthetics could be anything; in the ggplot2 implementation, there are some predetermined types of data and aesthetics. Commonly used data types include numeric values, categorical values, and text strings. Some commonly used aesthetics include horizontal and vertical position, color, size, and shape.

To interpret the graph, viewers refer to the *guides*. An example of a guide is the y-axis, including the tick marks and labels. The viewer refers to this guide to interpret what it means when a point is in the middle of the scale. A *legend* is another type of scale. A legend might show people what it means for a point to be a circle or a triangle, or what it means for a line to be blue or red.

Some aesthetics can only work with categorical variables, such as the shape of a point: triangles, circles, squares, etc. Some aesthetics work with categorical or continuous variables, such as *x* (horizontal) position. For a bar graph, the variable must be categorical—it would make no sense for there to be a continuous variable on the x-axis. For a scatter plot, the variable must be numeric. Both of these types of data (categorical and numeric) can be mapped to the aesthetic space of *x* position, but they require different types of scales.

In ggplot2 terminology, categorical variables are called *discrete*, and numeric variables are called *continuous*. These terms may not always correspond to how they're used elsewhere. Sometimes a variable that is continuous in the ggplot2 sense is discrete in the ordinary sense. For example, the number of visible sunspots must be an integer, so it's numeric (*continuous* to ggplot2) and discrete (in ordinary language).

Building a Simple Graph

Ggplot2 has a simple requirement for data structures: they must be stored in data frames, and each type of variable that is mapped to an aesthetic must be stored in its own column. In the `simpledat` examples we looked at earlier, we first mapped one variable to the `x` aesthetic and another to the `fill` aesthetic; then we changed the mapping specification to change which variable was mapped to which aesthetic.

We'll walk through a simple example here. First, we'll make a data frame of some sample data:

```
dat <- data.frame(xval=1:4, yval=c(3,5,6,9), group=c("A","B","A","B"))
dat
```

```
 xval yval group
    1    3     A
    2    5     B
    3    6     A
    4    9     B
```

A basic `ggplot()` specification looks like this:

```
ggplot(dat, aes(x=xval, y=yval))
```

This creates a `ggplot` object using the data frame `dat`. It also specifies default *aesthetic mappings* within `aes()`:

- `x=xval` maps the column `xval` to the *x* position.
- `y=yval` maps the column `yval` to the *y* position.

After we've given `ggplot()` the data frame and the aesthetic mappings, there's one more critical component: we need to tell it what *geometric objects* to put there. At this point, ggplot2 doesn't know if we want bars, lines, points, or something else to be drawn on the graph. We'll add `geom_point()` to draw points, resulting in a scatter plot:

```
ggplot(dat, aes(x=xval, y=yval)) + geom_point()
```

If you're going to reuse some of these components, you can store them in variables. We can save the `ggplot` object in `p`, and then add `geom_point()` to it. This has the same effect as the preceding code:

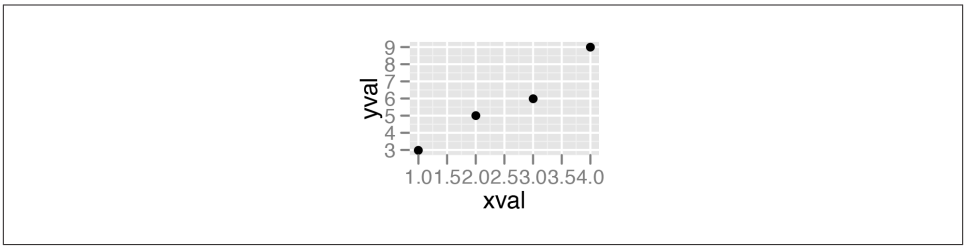

Figure A-7. A basic scatter plot

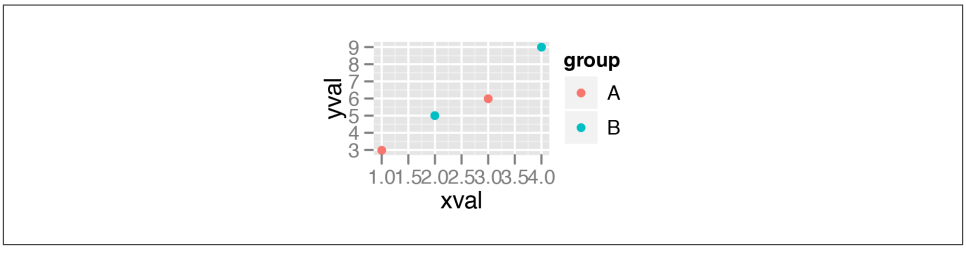

Figure A-8. A scatter plot with a variable mapped to colour

```
p <- ggplot(dat, aes(x=xval, y=yval))
p + geom_point()
```

We can also map the variable `group` to the color of the points, by putting `aes()` inside the call to `geom_point()`, and specifying `colour=group`:

```
p + geom_point(aes(colour=group))
```

This doesn't alter the *default* aesthetic mappings that we defined previously, inside of `ggplot(...)`. What it does is add an aesthetic mapping for this particular geom, `geom_point()`. If we added other geoms, this mapping would not apply to them.

Contrast this aesthetic *mapping* with aesthetic *setting*. This time, we won't use `aes()`; we'll just set the value of `colour` directly:

```
p + geom_point(colour="blue")
```

We can also modify the *scales*; that is, the mappings from data to visual attributes. Here, we'll change the *x* scale so that it has a larger range:

```
p + geom_point() + scale_x_continuous(limits=c(0,8))
```

If we go back to the example with the `colour=group` mapping, we can also modify the color scale:

```
p + geom_point() +
    scale_colour_manual(values=c("orange","forestgreen"))
```

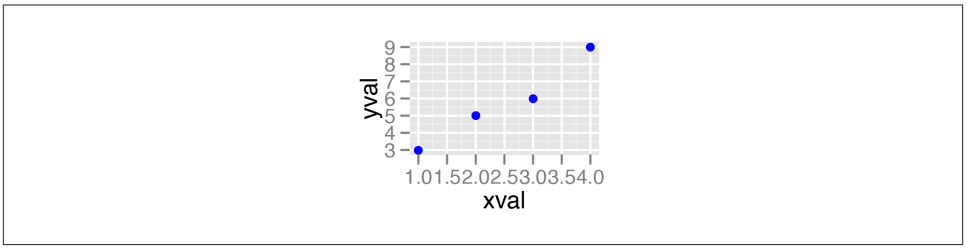

Figure A-9. A scatter plot with colors set

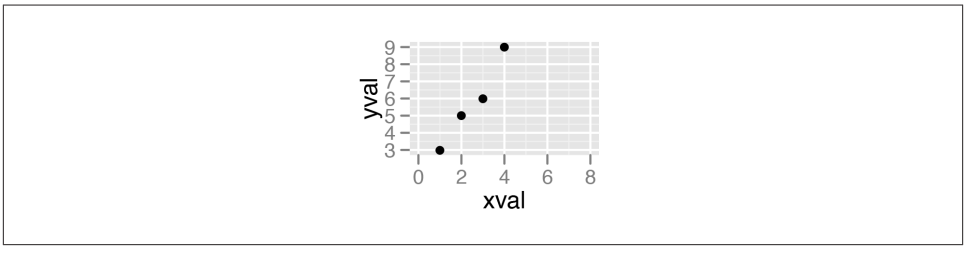

Figure A-10. A scatter plot with increased x-range

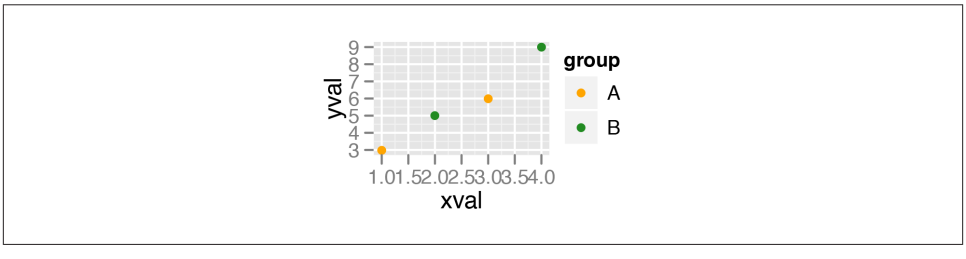

Figure A-11. A scatter plot with modified colors and a different palette

Both times when we modified the scale, the *guide* also changed. With the *x* scale, the guide was the markings along the x-axis. With the color scale, the guide was the legend.

Notice that we've used + to join together the pieces. In this last example, we ended a line with +, then added more on the next line. If you are going to have multiple lines, you have to put the + at the end of each line, instead of at the beginning of the next line. Otherwise, R's parser won't know that there's more stuff coming; it'll think you've finished the expression and evaluate it.

Printing

In R's base graphics, the graphing functions tell R to draw graphs to the output device (the screen or a file). Ggplot2 is a little different. The commands don't directly draw to the output device. Instead, the functions build plot *objects*, and the graphs aren't drawn until you use the `print()` function, as in `print(object)`. You might be thinking, "But wait, I haven't told R to print anything, yet it's made these graphs!" Well, that's not exactly true. In R, when you issue a command at the prompt, it really does two things: first it runs the command, then it runs `print()` with the returned result of that command.

The behavior at the interactive R prompt is different from when you run a script or function. In scripts, commands aren't automatically printed. The same is true for functions, but with a slight catch: the result of the last command in a function is returned, so if you call the function from the R prompt, the result of that last command will be printed because it's the result of the function.

Some introductions to ggplot2 make use of a function called `qplot()`, which is intended as a convenient interface for making graphs. It does require a little less typing than using `ggplot()` plus a geom, but I've found it a bit confusing to use because it has a slightly different way of specifying certain graphing parameters. I think it's simpler and easier to just use `ggplot()`.

Stats

Sometimes your data must be transformed or summarized before it is mapped to an aesthetic. This is true, for example, with a histogram, where the samples are grouped into bins and counted. The counts for each bin are then used to specify the height of a bar. Some geoms, like `geom_histogram()`, automatically do this for you, but sometimes you'll want to do this yourself, using various `stat_`*`xx`* functions.

Themes

Some aspects of a graph's appearance fall outside the scope of the grammar of graphics. These include the color of the background and grid lines in the graphing area, the fonts used in the axis labels, and the text in the graph title. These are controlled with the `theme()` function, explored in Chapter 9.

End

Hopefully you now have an understanding of the concepts behind ggplot2. The rest of this book shows you how to use it!

Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

B

C

D

E

F

G

H

N

O

P

Q

R

S

Y

Z

About the Author

Winston Chang is a software engineer at RStudio, where he works on data visualization and software development tools for R. He holds a Ph.D. in Psychology from Northwestern University. During his time as a graduate student, he created a website called "Cookbook for R," which contains recipes for handling common tasks in R. In previous lives, he was a philosophy graduate student and a computer programmer.

Colophon

The animal on the cover of *R Graphics Cookbook* is a reindeer (*Rangifer tarandus*), also known as caribou in North America, which is a species of deer native to Arctic and Subarctic regions. Reindeer are ideally designed for life in hostile, cold environments, as their fur, antlers, noses, hooves, and vision have adapted to the low temperatures.

Their fur coat consists of an outer layer of straight, hollow, tubular hairs, which provide insulation from the cold and buoyancy in water, and a woolly undercoat. The coat is such an efficient insulator that when they lay on the snow, the snow does not melt. Reindeer are the only species of deer in which both male and female (and even calves) have antlers, and they have the largest antlers relative to body size among living deer species. Their antlers are shed annually and new antler growth occurs in the spring and summer.

Reindeer hooves adapt to the season: in the summer, when the tundra is soft and wet, the footpads become sponge-like and provide extra traction. In the winter, the pads shrink and tighten, exposing the rim of the hoof, which cuts into the ice and crusted snow to keep the deer from slipping. This also enables them to dig down (an activity known as cratering) through the snow to their favorite food, a lichen known as reindeer moss.

In 2012, researchers at University College London discovered reindeer are the only mammals that can see ultraviolet light. While human vision cuts off at wavelengths around 400 nm, reindeer can see up to 320 nm. This range only covers the part of the spectrum we can see with the help of a black light, but it is still enough to help reindeer see things in the glowing white of the Arctic that they would otherwise miss.

In the Santa Claus tale, Santa Claus's sleigh is pulled by flying reindeer. These were first named in the 1823 poem "A Visit from St. Nicholas," where they are called Dasher, Dancer, Prancer, Vixen, Comet, Cupid, Dunder, and Blixem.

The cover image is from Shaw's *Zoology*. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.