dope, *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent\(^1\)

**This week’s dope**

This we will learn how to

1. Make basic plots using `qplot()`
2. Use aesthetics to enhance plots
3. Use non-default geoms and stats to increase the palette of plots available
4. Use faceting to create sub-plots

This Dope Sheet includes terse descriptions and examples of the main things covered this week. See the other course materials and the *ggplot2* book for more complete descriptions and additional examples. *But note that there have been some changes to *ggplot2* since the book was published, so some of the code used in the book is no longer consistent with the current version of the package.*

**Packages and Data**

We will be using the *ggplot2* and *plyr* packages throughout this class, so we should get in the habit of making sure they are loaded before we do anything else. In addition, for these examples, we will use some data from the *mosaic* package.

```r
require(ggplot2)
require(plyr)
require(mosaic)
```

Here are the first few lines of a data set we will use for illustrative purposes.

```r
head(HELPrct, 3)
```

```r
# age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b
# 1 37 1 yes 49 3 177 225 0 NA
# 2 37 1 yes 30 22 2 NA 0 NA
# 3 26 1 yes 39 0 3 365 20 NA
# female sex glb homeless i1 i2 id indtot linkstatus link mcs
# 1 0 male yes housed 13 26 1 39 1 yes 25.112
# 2 0 male yes homeless 56 62 2 43 NA <NA> 26.670
# 3 0 male no housed 0 0 3 41 0 no 6.763
# pcs pss_fr racegrp satreat sexrisk substance treat
# 1 58.41 0 black no 4 cocaine yes
# 2 36.04 1 white no 7 alcohol yes
# 3 74.81 13 black no 2 heroin no
```

\(^1\)definitions selected from Webster’s online dictionary

©2013 • Randall Pruim
Use

?HELPrtc

to find out more about this data set. Some of the variables have pretty opaque names. Let’s create a new
variables with a better name for the average and maximum number of drinks per day over the 30 days
prior to admission for substance abuse.

HELPrtc <- transform(HELPrtc, aveDrinks = i1, maxDrinks = i2)

The transform() command creates a new data frame with additional variables computed from the
variables in HELPrtc. By assigning this new data frame to HELPrtc, we have essentially updated HELPrtc
with two new variables. (As an alternative, we could have chosen to simply rename the variables i1 and
i2, but transform() is a generally useful function to know about.)

1 Getting Started with qplot()

The easiest way to make a plot in ggplot2 is with qplot(). In it’s simplest form, qplot() requires only
two or three pieces of information:

- the name(s) of the variable(s) providing the data to be displayed, and
- the name of a data frame containing those variables (data=).

qplot() will inspect the variables and attempt to make a reasonable plot depending on whether one or
two variables are provided and whether they are categorical (factors) or quantitative (numeric).

1.1 One variable plots

# one categorical variable -> bar chart
qplot(substance, data = HELPrtc)
# one quantitative variable -> histogram
qplot(aveDrinks, data = HELPrtc)

## stat_bin: binwidth defaulted to range/30. Use ’binwidth = x’ to adjust this.
1.2 Two variable plots

When two variables are provided, the result is a scatter plot. The first variable goes on the horizontal axis and the second on the vertical axis.

```r
# two quantitative variables
qplot(aveDrinks, maxDrinks, data = HELPrct)
# two categorical variables
qplot(sex, substance, data = HELPrct)
```

The points along and below the diagonal are an indication that some data cleaning is still required for this data.

```r
# one categorical and one quantitative variable
qplot(sex, aveDrinks, data = HELPrct)
qplot(aveDrinks, sex, data = HELPrct)
```

Overplotting of data or labels in these plots might render them less than ideal. This is a common problem in large data sets or when one or more variables are categorical. We’ll learn some methods for dealing with this soon.

2 Using aesthetics

In addition to position, other aesthetics like size, color, shape, and transparency (called alpha) can be used to represent information on a plot. Aesthetics can be mapped to a variable or set to a constant value.
2.1 Mapping aesthetics

Mapping assigns aesthetics based on the value of a variable, using reasonable default mappings for each type of aesthetic.

```r
googplot(aveDrinks, maxDrinks, data = HELPrct, color = sex, shape = substance)
```

These plots reveal the need for some data cleaning – several subjects have a value of 0 for `maxDrinks` but a non-zero value for `aveDrinks`. A simple rule would be discard values of `maxDrinks` that are less than the value of `aveDrinks`. There are many subjects for whom these two variables are equal as well – suggesting that `maxDrinks` might not be very reliable. For now, let’s remove all these potentially problematic issues and deal with a subset of the original data.

```r
HELPrct <- subset(HELPrct, maxDrinks > aveDrinks)
```

Fill is used to set interior colors (as opposed to outlines).

```r
googplot(substance, data = HELPrct, color = sex)
googplot(substance, data = HELPrct, fill = sex)
```
2.2 Setting aesthetics

To set an aesthetic to a constant value, wrap that value in `I()`.

```r
qplot(aveDrinks, maxDrinks, data = HELPrct, alpha = I(0.2), size = I(4), shape = I(18), color = I("navy"))
```

The use of alpha to make the points quite transparent illustrates one way to deal with overplotting. Where there is more data, the points (diamond-shaped in this plot) become darker.

3 Choosing non-default geoms and stats

**Geoms** are the objects responsible for displaying the data in a `ggplot2` plot. **Stats** transform the data before the geoms do their thing. A histogram, for example, uses the "bar" geom and the "bin" stat. The "bin" stat bins the data, producing a summary data frame tabulating the bins used and the count in each bin. This summarized data is then plotted as bars by the "bar" geom. All this happened automatically when we provided quantitative data to `qplot()`. By selecting different geoms and stats, we can create other representations of our data.

Each geom is paired with a default stat and vice versa. When only one is specified, the corresponding default is used for the other. For example, the "density" geom uses the code "density" stat by default, but the "density" stat uses the "area" geom by default. This provides two slightly different density plots. We can create our own variation by combining the "line" geom with the "density" stat to remove the extra horizontal and vertical lines added by the "density" geom.

```r
qplot(aveDrinks, data = HELPrct, geom = "density")
qplot(aveDrinks, data = HELPrct, stat = "density")
# We can change the geom to avoid the additional line segments added by
# the density geom
qplot(aveDrinks, data = HELPrct, stat = "density", geom = "line")
```
Similarly, we can create frequency polygons by using the "bin" stat used to create histograms with the "polygon" or "line" geoms.

```r
qplot(aveDrinks, data = HELPrc, geom = "polygon", stat = "bin", binwidth = 10)
qplot(aveDrinks, data = HELPrc, geom = "line", stat = "bin", binwidth = 10)
```

For a complete list of geoms and stats, use function `apropos()`:

```r
apropos("^stat_") # list all functions starting stat_

## [1] "stat_abline"    "stat_bin"      "stat_bin2d"
## [4] "stat_bindot"    "stat_binhex"  "stat_boxplot"
## [7] "stat_contour"   "stat_density" "stat_density2d"
## [10] "stat_ecdf"      "stat_function" "stat_hline"
## [13] "stat_identity"  "stat_qq"      "stat_quantile"
## [16] "stat_smooth"    "stat_spoke"   "stat_sum"
## [19] "stat_summary"   "stat_summary_hex" "stat_summary2d"
## [22] "stat_unique"    "stat_vline"   "stat_ydensity"
```

```r
apropos("^geom_") # list all functions starting geom_
```

```r
## [1] "geom_abline"    "geom_area" "geom_bar"
## [4] "geom_bin2d"     "geom_blank" "geom_boxplot"
## [7] "geom_contour"   "geom_crossbar" "geom_density"
## [10] "geom_density2d" "geom_dotplot" "geom_errorbar"
## [13] "geom_errorbarh" "geom_freqpoly" "geom_hex"
## [16] "geom_histogram" "geom_hline" "geom_jitter"
## [19] "geom_line"      "geom_linerange" "geom_map"
## [22] "geom_path"      "geom_point" "geom_pointrange"
## [25] "geom_polygon"   "geom_quantile" "geom_raster"
## [28] "geom_rect"      "geom.ribbon" "geom_rug"
## [31] "geom_segment"   "geom_smooth" "geom_step"
## [34] "geom.text"      "geom_tile" "geom.violin"
```
```r
qplot(aveDrinks, data = HELPrct, geom = "line", stat = "density", color = substance)
qplot(aveDrinks, data = HELPrct, geom = "line", stat = "bin", binwidth = 10,
     color = substance)
```

```
0.00
0.01
0.02
0.03
0.04
0.05
0 50
aveDrinks
density
substance
alcohol
cocaine
heroin
```

```
qplot(substance, aveDrinks, data = HELPrct, geom = "boxplot")
qplot(substance, aveDrinks, data = HELPrct, geom = "jitter")
```

```
        0
10
20
0 50 100
0
10
20
4 Faceting
Faceting creates separate sub-plots for each subset. Faceting is described with a formula.

qplot(aveDrinks, maxDrinks, data = HELPrct, color = substance, shape = sex)
```

```
        0
50
100
0
50
100
alcohol cocaine heroin
substance
aveDrinks
maxDrinks
```

```
qplot(aveDrinks, maxDrinks, facets = sex ~ substance)
```

©2013 • Randall Pruim
Redundant coding is sometimes useful even when faceting.

```r
qplot(aveDrinks, maxDrinks, data = HELPrct, facets = sex ~ substance, color = substance, shape = sex)
```

5 More than one geom

It is possible to specify multiple geoms at once. The smooth geom creates a LOESS or regression smoothing and adds it to the plot.

```r
qplot(aveDrinks, maxDrinks, data = HELPrct, facets = . ~ sex, geom = c("point", "smooth"), col = sex)
```

```r
qplot(aveDrinks, maxDrinks, data = HELPrct, geom = c("point", "smooth"), color = sex, alpha = I(0.5))
```

©2013 • Randall Pruim
Many other combinations are possible as well.

```r
ggplot2 @ statistics.com

```r
qplot(substance, aveDrinks, data = HELPrct, geom = c("boxplot", "jitter"),
      alpha = I(0.2))
qplot(substance, aveDrinks, data = HELPrct, geom = c("violin", "jitter"),
      alpha = I(0.2))
```

This method works as long as all the geoms involved use the same information. Notice how setting the color below sets the color for both the lines and the bars. Similarly, `binwidth` is shared by both geoms.

```r
qplot(aveDrinks, data = HELPrct, geom = c("bar", "line"), stat = "bin",
      fill = I("skyblue"), binwidth = 10)
qplot(aveDrinks, data = HELPrct, geom = c("bar", "line"), stat = "bin",
      fill = I("skyblue"), color = I("navy"), binwidth = 10)
```

To achieve finer control, we need to work with geoms, stats, their cousins directly. We’ll learn to do this next time.
6 Dealing with overplotting in large data sets

We will return to this issue repeatedly during the course, learning more approaches as we go along. Solutions fall into one of several general categories:

- Use faceting to reduce the amount of data in each subplot
- Use transparency (alpha) to reveal where overplotting is occurring
- Use the jitter geom instead of the point geom when there is significant discreteness in the data. This moves the points by a small random amount. Some accuracy is sacrificed for the sake of readability.
- Use plots that employ data reduction (e.g., boxplots, histograms)
This week's dope

This week we will learn how to

1. Use `ggplot()` to make plots without `qplot()`
2. Use geoms, stats, and scales directly to gain a better understanding of the `ggplot2` grammar of graphs and to achieve finer control over our plots

This Dope Sheet includes terse descriptions and examples of the main things covered this week. See the other course materials and the `ggplot2` book for more complete descriptions and additional examples.

Note: The plots in this document are rendered as png files to reduce the overall file size and the price of some quality.

1 The `ggplot2` workflow

1.1 Data and aesthetics

All plots begin with data. For `ggplot2`, the data must be in a data frame. If you have data in other forms, the first thing you must do is create a data frame containing your data.

```r
a <- 1:10
b <- 4 * a^3
artificial <- data.frame(a = a, b = b)
head(artificial, 3)
```

```text
## a  b
## 1 4
## 2 32
## 3 108
```

Our first decisions when making a plot are generally to identify the data we want to plot and to define the appropriate aesthetics. Aesthetics tell `ggplot2` how to map the data onto position (x and y), color, size, transparency, etc. when they are plotted.

```r
require(ggplot2)
ggplot(diamonds, aes(y = carat, x = color))
```

```text
## Error: No layers in plot
```

This alone doesn’t show any plot, however. Unlike `qplot()`, `ggplot()` does not know about any default geoms or stats, so we need to add those to the plot.

---

1 definitions selected from Webster’s online dictionary
2 Next week we will learn tools for manipulating data sets to get them into the correct form for our plot.
1.2 Layers, geoms, and stats

Each layer of a ggplot2 plot consists of data, aesthetics, a geom, a stat (and scales, which we will get to shortly). To get a plot, we need to add information about our desired geom and stat for each layer:

```r
p <- ggplot(diamonds, aes(y = carat, x = color))
p + layer(geom = "boxplot", geom_params = list(color = "navy", outlier.colour = "red", fill = "skyblue"), stat = "boxplot", stat_params = list(coef = 3))
```

Since each geom comes with a default stat and each stat with a default geom, it suffices to supply only one of these (provided we are happy with the default value for the other). The functions beginning geom_ and stat_ are short cuts that create layers in a way that is less verbose. We can even pass through the stat_params or geom_params.

```r
p + geom_boxplot(colour = "navy", outlier.colour = "red", fill = "skyblue", coef = 3)
```

Furthermore, we can override the default geom or stat as well. The example below shows how to create a frequency polygon by combining geom_polygon() and stat_bin():

```r
p2 <- ggplot(diamonds, aes(x=carat, y=..count..))
p2 + stat_bin(geom="polygon") # uses geom_polygon() instead of geom_histogram()
p2 + geom_polygon(stat="bin") # same as above.
```
1.3 Multiple layers

Plots may have multiple layers (each layer with its own data, aesthetics, geom, and stat). Here we show how a frequency polygon and a histogram are related by using two layers. By using alpha transparency in the top layer, we can see through it to the layer below.

```
p2 + stat_bin() + stat_bin(geom = "polygon", fill = "skyblue", alpha = 0.5, color = "white", )
```

Multiple layers need not use the same data or aesthetics.

```
require(scales) # to get the alpha function
D <- subset(diamonds, color == "D")
J <- subset(diamonds, color == "J")
p3 <- ggplot(D, aes(x = carat, y = ..density..))
p3 + geom_histogram(fill = alpha("navy", 0.5), color = "navy") + geom_histogram(data = J, fill = alpha("red", 0.5), color = "red")
p3 + stat_bin(geom = "polygon", fill = alpha("navy", 0.5), color = "navy") +
  stat_bin(geom = "polygon", data = J, fill = alpha("red", 0.5), color = "red")
```
1.4 Data flow

As a plot is created the original data set undergoes a sequence of transformations. At each stage, the available data are stored in a data frame (actually, one data frame for each layer of each facet).3

\[
\text{original data} \xrightarrow{\text{stat}} \text{statified data} \xrightarrow{\text{aesthetics}} \text{aesthetic data} \xrightarrow{\text{scales}} \text{scaled data}
\]

Since at each step an entire data frame is transformed into an entire data frame, it is possible for new variables to be introduced along the way. This explains why the following example works.

\[
\text{ggplot}(\text{data} = \text{diamonds}, \text{aes}(x = \text{carat}, \text{fill} = \text{..count..})) + \text{stat_bin()}
\]

There is no variable called \text{..count..} in \text{diamonds}, but \text{stat_bin()} produces a new data frame with one row for each bin and new columns \text{..count..}, \text{..density..}, \text{..ncount..}, and \text{..ndensity}. Here’s a more unusual representation of the same information.

\[
\text{ggplot}(\text{data} = \text{diamonds}, \text{aes}(x = \text{carat}, y = \text{..density..}, \text{color} = \text{..count..})) + \\
\text{stat_bin(geom} = \text{"point", \text{aes(size} = \text{..count..)})}
\]

1.5 Scales

The final data transformation is performed by scales, which translate the aesthetic data into something the computer can use for plotting. (Scales also control the rendering of guides – a collective term for axes and legends – which are the inverse of scales in that they help humans convert from what is visible on the plot back into the units of the underlying data.) Positions are mapped to the interval \([0, 1]\), and other aesthetics must be mapped to actual colors, sizes, etc. that are used by the geom to render the plot.

---

3This is a bit of an oversimplification. Actually, the aesthetics get computed twice, once before the stat and again after. For a histogram, for example, we need to look at the aesthetics to figure out which variable to bin (that’s the stats job), but it isn’t until after the binning that we will know the bin counts, which become part of the aesthetics. Nevertheless, the simple version depicted is a useful starting point. See also page 36 in the \text{ggplot2} book.
1.5.1 Position scales

Position scales may be either continuous (for numerical data), discrete (for factors, characters, and logicals), or date. Each scale has appropriate arguments for controlling how the scale works. Here are a few examples.

```r
a <- 1:10; b <- 4 * a^3
artificial <- data.frame(a=a, b=b)

p4 <- ggplot(data=artificial, aes(x=a, y=b)) + geom_point(); p4
# log scaling on the x axis
p4 + scale_x_continuous(trans="log10")
# Note: dollar is a _function_ that formats things as money
p4 + scale_x_continuous(trans="log10", breaks=seq(2,10,by=2), label=dollar) +
  scale_y_continuous(trans="log10")
```

1.5.2 Color and shape scales

Scales are also used to control how aesthetics are mapped to colors and shapes.

```r
p5 <- ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species))
p5 + geom_point() + scale_color_manual(values = c("orange", "purple", "skyblue"))
p5 + geom_point(size = 5, alpha = 0.5) + scale_color_brewer(type = "qual",
  palette = 1, name = "iris species")
```
For more examples, see the ggplot2 book. To find a list of scale functions, use `apropos()`:

```r
apropos("^scale_")
```

```r
# [1] "scale_alpha"           "scale_alpha_continuous"
# [3] "scale_alpha_discrete"  "scale_alpha_identity"
# [5] "scale_alpha_manual"    "scale_area"
# [7] "scale_color_brewer"    "scale_color_continuous"
# [9] "scale_color_discrete"  "scale_color_gradient"
# [11] "scale_color_gradient2" "scale_color_gradientn"
# [13] "scale_color_grey"      "scale_color_hue"
# [15] "scale_color_identity"  "scale_color_manual"
# [17] "scale_colour_brewer"   "scale_colour_continuous"
# [19] "scale_colour_discrete" "scale_colour_gradient"
# [21] "scale_colour_gradient2" "scale_colour_gradientn"
# [23] "scale_colour_grey"     "scale_colour_hue"
# [25] "scale_colour_identity" "scale_colour_manual"
# [27] "scale_fill_brewer"     "scale_fill_continuous"
# [29] "scale_fill_discrete"   "scale_fill_gradient"
# [31] "scale_fill_gradient2"  "scale_fill_gradientn"
# [33] "scale_fill_hue"        "scale_fill_manual"
# [35] "scale_fill_identity"   "scale_fill_manual"
# [37] "scale_linetype"       "scale_linetype_continuous"
# [39] "scale_linetype_discrete" "scale_linetype_identity"
# [41] "scale_linetype_manual" "scale_shape"
# [43] "scale_shape_continuous" "scale_shape_discrete"
# [45] "scale_shape_identity"  "scale_shape_manual"
# [47] "scale_size"           "scale_size_area"
# [49] "scale_size_continuous" "scale_size_discrete"
# [51] "scale_size_identity"  "scale_size_manual"
# [53] "scale_x_continuous"   "scale_x_date"
# [55] "scale_x_datetime"     "scale_x_discrete"
# [57] "scale_x_log10"        "scale_x_reverse"
# [59] "scale_x_sqrt"         "scale_y_continuous"
# [61] "scale_y_date"         "scale_y_datetime"
# [63] "scale_y_discrete"     "scale_y_log10"
# [65] "scale_y_reverse"      "scale_y_sqrt"
```
1.6 Coordinates

A coordinate system controls how positions are mapped to the plot. The most common coordinate system is Cartesian coordinates, but polar coordinates, and spherical projection (for maps) are also available. One other important coordinate system is `coord_flip()`, which reverses the roles of the horizontal and vertical axes. This allows us, for example, to create horizontal boxplots.

```r
p6 <- ggplot(diamonds, aes(x = color, y = carat)) + geom_boxplot()
p6
p6 + coord_flip()
```

![Horizontal Boxplots]

One additional use for a coordinate system is to zoom in on a portion of a plot using `xlim` and `ylim`. Setting limits in a scale removes all data outside those limits from the data frame used to generate the plot; setting the limits in a coord merely restricts the view – all the data are still used. The difference is illustrated below. Notice the differences in the smoothing function and also differences in the plot window used.

```r
p7 <- ggplot(mpg, aes(cty, hwy))
p7 + geom_point() + geom_smooth()
p7 + geom_point() + geom_smooth() + scale_x_continuous(limits = c(15, 25))
p7 + geom_point() + geom_smooth() + coord_cartesian(xlim = c(15, 25))
```

![Zooming in on a Plot]

©2013 • Randall Pruim
When working with discrete data, the limits are set as numerical values in the coord system but as natural values in the scale (and need not be contiguous or in the original order). There's really no reason for a legend on these plots, so let's turn it off.\footnote{We'll have more to say about \texttt{theme()} later. Note that \texttt{theme()} replaces \texttt{opt()} from older versions of \texttt{ggplot2}.}

\begin{verbatim}
p8 <- ggplot(diamonds, aes(x = color, y = carat, color = color)) + geom_boxplot()
p8 + scale_x_discrete(limits = c("E", "J", "G")) + theme(legend.position = "none")
p8 + coord_cartesian(xlim = c(1.5, 4.5)) + theme(legend.position = "none")
\end{verbatim}

1.7 Position – jitter and dodge

We've already seen \texttt{geom_jitter()}, which is really just \texttt{geom_point()} with position="jitter".

\begin{verbatim}
ggplot(diamonds, aes(y = carat, x = color)) + geom_jitter(alpha = 0.2)
ggplot(diamonds, aes(y = carat, x = color)) + geom_point(position = "jitter", alpha = 0.2)
\end{verbatim}

Dodging is similar to jitter, but for discrete variables.
p9 <- ggplot(diamonds, aes(x = color, fill = cut))
p9 + geom_bar()
p9 + geom_bar(position = "dodge")

Dodging often yields results that are similar to faceting, but the labeling is different, and the results can be less than pleasing if some of the discrete categories are unpopulated.

ggplot(mpg, aes(x = class, group = cyl, fill = factor(cyl))) + coord_flip() + geom_bar(position = "dodge")

2 Some shortcuts

2.1 qplot()

Since `qplot()` produces a plot, we can add to it just like we do other plots.

`qplot(carat, data = diamonds) + coord_flip()`
qplot() also provides quick access to some common plot modifications:

- `log='x'`, `log='y'`, and `log='xy'` can be used for log-transformations of x or y.
- `main=` can be used to create a plot title.

### 2.2 xlim() and ylim()

Since limiting the values on a plot is so common, ggplot2 provides the xlim() and ylim() to set limits without needing to directly invoke scales.

```r
qplot(carat, price, data = diamonds, alpha = I(0.3)) + xlim(c(1, 2))
```

### 2.3 labs(), xlab(), ylab()

labs() provides a quick way to add labels for the axes and a title for the plot. xlab() and ylab() are equivalent to labs(x=...) and labs(y=...).

```r
p8 + labs(title = "Now I have a title", x = "color of diamond", y = "size (carat)")
```

### 3 Dealing with overplotting in large data sets

When there is a high degree of discreteness in the data or the data is very large, multiple points may land on the same position making it impossible to tell whether there is a lot or a little data there. Here are several methods for dealing with “too much data for the plot”.

#### 3.1 Jittering

If the primary problem is discreteness and the data set is not too large, simply jittering the position a bit can be a sufficient solution.
3.2 Using transparency

A similarly simple solution is to use transparency so that where there is more data, the plot becomes darker.

3.3 Use plots that rely on data reduction

For 1d plots, histograms, density plots, and boxplots all use data reduction methods, so these plots work well for data sets of all sizes – except very small data sets, I suppose.

For 2d data, instead of scatterplots, we can use 2d versions of histograms that use rectangular or hexagonal (requires the hexbin package) bins and use color to represent the bin frequency.

```r
p10 <- ggplot(diamonds, aes(x = carat, y = depth))
p10 + stat_bin2d()
require(hexbin)  # You may need to install this package

## Loading required package: hexbin

p10 + stat_binhex()
```

Alternatively, we can plot the contours of a 2d kernel density estimate.

```r
p10 + stat_density2d()
```

```r
p10 + ylim(c(55,68)) + stat_binhex(aes(fill=..density..)) +
    stat_density2d(color="gray75", size=.3)

## Warning: Removed 110 rows containing missing values (stat_hexbin).
## Warning: Removed 110 rows containing non-finite values (stat_density2d).
```
**dope**, *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent

**This week’s dope**

Often the key to creating a good plot in ggplot is getting data into a format that makes it possible to create the plot you want. So, while we will continue to hone our skills at making plots using ggplot, much of this week’s content will be about “data wrangling.” In particular, we will learn how to

1. Use `subset()` to select rows from a data frame meeting some criterion
2. Use `transform()` and `mutate()` to add new variables to a data frame
3. Use `interaction()` to combine two variables into one
4. Use `summarize()` to create new data frames summarizing existing data frames.
5. Use `ddply()` and the split-apply-combine methodology to apply a function to many subsets of a data frame.
6. Use `merge()` to combine data from multiple data frames.

Most of this week’s material can be found in Chapter 9 of the ggplot2 book.

**Baby Names**

The primary data set used in our examples is not in the ggplot2 or plyr packages, but is available as a csv file in Lesson 3. This data set has information on the 1000 most popular US names for each sex for each year from 1880 to 2008.

```r
require(ggplot2)
require(plyr)
# ddply() lives here
# so we can return to original data without reloading it, we'll save the data with two names.
Bnames.orig <- read.csv("baby-names.csv", as.is = TRUE)
Bnames <- Bnames.orig
head(Bnames, 3)
## year name prop sex
## 1 1880 John 0.08154 boy
## 2 1880 William 0.08051 boy
## 3 1880 James 0.05006 boy

dim(Bnames)
## [1] 258000 4

qplot(year, data = Bnames, fill = sex, binwidth = 1)
```

1 definitions selected from Webster’s online dictionary
1 Subsetting with subset()

While is is possible to do subsetting operations by passing logicals into the [] operator in R, it is sometimes simpler and clearer to use the subset() function. The general form of a subsetting command is:

```r
newdataframe <- subset(dataframe, condition)
```

The condition should evaluate to a logical and will often reference the variables in the data frame being subsetting. The $ operator is not required to reference these variables, which is a big savings for complicated subsets.

```r
Boys <- subset(Bnames, sex == "boy")
Girls <- subset(Bnames, sex == "girl")
```

Here's something a bit more interesting:

```r
AtLeast1PercentGirls <- subset(Bnames, sex="girl" & prop > 0.01)
qplot(year, data=AtLeast1PercentGirls, 
      binwidth=1, origin=1879.5, # center bins on integers
      fill=I("skyblue"))
```

So over past few decades, the number of names given to at least 1% of the girls has been decreasing. In section 5.4 we will see how to plot the proportion of girls who have names shared with at least 1% of the girls born each year.

2 Adding new variables to a data frame with transform() and mutate()

If we would prefer to work with percentages instead of proportions, we could create a new variable for that.\(^2\)

\(^2\)But the only reason to do so is to get percentages to appear on a plot, there is not reason to do so. Instead, ggplot can be instructed to format the scales using percent notation rather than decimal notation – simply pass `labels=percent` to the appropriate scale(s). See Section 5.4 for an example.
Bnames <- transform(Bnames.orig, percent = 100 * prop)

head(Bnames, 3)

## year  name  prop  sex percent
## 1 1880  John 0.08154  boy 8.154
## 2 1880  William 0.08051  boy 8.051
## 3 1880  James 0.05006  boy 5.006

More interestingly, we can compute some information from the names themselves. First, let’s define a few helper functions.

```r
letter <- function(x, start = 1, stop = start) {
  # convert negative numbers to positions relative to end of string
  n <- c(start, stop)
  n <- ifelse(n < 0, nchar(x) + n + 1, n)
  tolower(substr(x, min(n), max(n)))
}

vowels <- function(x) {
  nchar(gsub("[^aeiouy]", "", tolower(x)))
}
```

Now let’s add some variables to our data frame:

```r
Bnames <- transform(Bnames.orig, first = letter(name, 1), last = letter(name, -1), length = nchar(name), vowels = vowels(name))

head(Bnames, 3)

## year  name  prop  sex first last length vowels
## 1 1880  John 0.08154  boy j n 4 1
## 2 1880  William 0.08051  boy w liam 7 3
## 3 1880  James 0.05006  boy j es 5 2
```

We could also have used `mutate()` to do this. The primary difference between `mutate()` and `transform()` is that `mutate()` can refer to variables created earlier in the `mutate()` command, but `transform()` cannot.

```r
Bnames <- mutate(Bnames.orig, first = letter(name, 1), last = letter(name, -1), length = nchar(name), vowels = vowels(name), consonants = length - vowels, vowelFrac = vowels/length)

head(Bnames, 3)

## year  name  prop  sex first last length vowels consonants vowelFrac
## 1 1880  John 0.08154  boy j n 4 1 3 0.2500
## 2 1880  William 0.08051  boy w liam 7 3 4 0.4286
## 3 1880  James 0.05006  boy j es 5 2 3 0.4000
```

3 Combining variables with `interaction()`

Sometimes it is convenient to combine two (or more) variables into one factor that contains the values of each of the original variables.

3.1 Vowels and consonants

Here is an example using `interaction()` to add a factor called `vcsplit` that tells us how many vowels and how many consonants are in a name. `drop=TRUE` removes unpopulated levels from the factor.
Bnames <- transform(Bnames, vcsplit = interaction(vowels, consonants, sep = ":", drop = TRUE))

head(Bnames, 3)
## year name prop sex first last length vowels consonants vowelFrac vcsplit
## 1 1880 John 0.08154 boy j n 4 1 3 0.2500 1:3
## 2 1880 William 0.08051 boy w liam 7 3 4 0.4286 3:4
## 3 1880 James 0.05006 boy j es 5 2 3 0.4000 2:3

t <- table(Bnames$vcsplit)
t[tail(order(t), 5)]  # 5 most common vowel/consonant splits
##
## 2:2 3:4 2:4 3:3 2:3
## 25585 26108 27948 38057 41115

3.2 mpg data set

Returning to our mpg data set for a moment, we might like to create a variable that encodes more specific model information than the model variable does. First we’ll add “wd” (for ‘wheel drive’) to the drv variable, and then we’ll combine several variables into a new definition of model, which we will call model2

mpg2 <- transform(mpg, drv = paste(drv, "wd", sep = ""))
mpg2 <- transform(mpg2, model2 = interaction(manufacturer, model, trans, fl, drv, sep = " ", drop = TRUE))

head(mpg2, 3)
## manufacturer model displ year cyl trans drv cty hwy fl class model2
## 1 audi a4 1.8 1999 4 auto(l5) fwd 18 29 p compact audi a4 auto(l5) p fwd
## 2 audi a4 1.8 1999 4 manual(m5) fwd 21 29 p compact audi a4 manual(m5) p fwd
## 3 audi a4 2.0 2008 4 manual(m6) fwd 20 31 p compact audi a4 manual(m6) p fwd

sep=" " defines the separator between the components, and drop=TRUE drops unused levels from the resulting factor. After making one more modification (to order the models according to their best fuel efficiency), we can use our modified data frame to create the plot in Figure 1.

mpg2 <- transform( mpg2, model2=reorder(model2, hwy, max) )
ggplot( aes(hwy, model2, colour=factor(year), shape=factor(year)),
data=subset(mpg2, class=="compact")) +
  geom_point(size=4, alpha=.5) +
  scale_colour_manual(values=c("blue","red")) +
  labs(shape="Year", colour="Year")

4 Summarising a data frame with summarise()

summarise() is analogous to transform(), but summarise() creates a new data frame instead of adding variables to an existing data frame. As with transform(), we can create several variables at once if we like.

summarise(Bnames, avg.length = mean(length), max.pop = max(prop))
## avg.length max.pop
## 1 5.838 0.08154

The real power of summarise() comes in combination with dply(), the topic of our next section.
Figure 1: Fuel efficiency for each year and each model of car
Note: `ggplot2` also includes a synonym, `summarizei()`, but there is also a `summarize()` function in the `Hmisc` package. If you use the `Hmisc` package, you will need to load things in the right order to have `summarize()` do what you want it to do.

5 Split-apply-combine using `ddply()`

Suppose we would like to know how the average name length and maximum popularity have changed over time, separately for each sex. That is, we would like to

1. **Split** our data frame into many subsets, one for each unique combination of year and sex;
2. **Apply** our `summarise()` function to each of these subset; and
3. **Combine** all the results into one data frame.

This is exactly what `ddply()` facilitates. The general format of a call to `ddply()` is

\[
\text{NewData} \leftarrow \text{ddply( OriginalData, .(var1, var2, ...), function, ...)}
\]

1. **Split**: `.var1, var2, ...` tells how split – subsets are formed for each unique combination of the variables.
2. **Apply**: `function` is applied to each of these subsets. The final `...` are arguments supplied to `function`.
3. **Combine**: `ddply()` combines all the results into one data frame.

Let’s give it a try:

```r
BnameSummaries <-
  ddply(Bnames, .(year,sex), summarise,
        max.prop = max(prop),
        avg.length = mean(length),
        avg.length2 = sum(length * prop) / sum(prop),
        vowelFrac = mean(vowelFrac ),
        vowelFrac2 = sum(vowelFrac * prop) / sum(prop))
head(BnameSummaries,3)
```

<table>
<thead>
<tr>
<th>year</th>
<th>sex</th>
<th>max.prop</th>
<th>avg.length</th>
<th>avg.length2</th>
<th>vowelFrac</th>
<th>vowelFrac2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1880</td>
<td>boy</td>
<td>0.08154</td>
<td>5.606</td>
<td>5.592</td>
<td>0.4099</td>
<td>0.4099</td>
</tr>
<tr>
<td>1880</td>
<td>girl</td>
<td>0.07238</td>
<td>5.776</td>
<td>5.410</td>
<td>0.4949</td>
<td>0.4949</td>
</tr>
<tr>
<td>1881</td>
<td>boy</td>
<td>0.08098</td>
<td>5.598</td>
<td>5.574</td>
<td>0.4056</td>
<td>0.4056</td>
</tr>
</tbody>
</table>

Notes:

1. When computing means, if there are missing values, `mean()` will return `NA` unless you set `na.rm=TRUE`. There are no missing values in this data set.
2. When computing the averages, we need to think about what we are averaging over. Do we want the average length of the top 1000 names or the average length of a name given to a child with a top 1000 name? Notice that we have computed averages two ways above.

5.1 Split-apply-combine in Slow Motion

It can be instructive to look at the split-apply-combine steps more carefully.

Split

Let choose just one year and one sex (2000, boy) and create the required subset manually.

```r
Male2000 <- subset(Bnames, year == 2000 & sex == "boy")
head(Male2000, 3)
```

This is precisely what `ddply()` is doing with every combination of year and sex.

Apply

Now can apply our function to one subset.

```r
summarise(Male2000, max.prop = max(prop), avg.length = mean(length))
```

Again, `ddply()` will apply this function to each of the subsets.
Combine

In the combine step, the results of apply a function to each of the subset data frames are combined together into one data frame. If you are having trouble getting `ddply()` to do what you are expecting, it can be helpful to create one subset and make sure that the function you apply to that one subset is working the way you imagine. If it is, you should be ready to go. If not, it will be faster to debug it by running it on just one subset until you get it right than to repeatedly try to run it on all subsets of a very large data frame.

5.2 Plots

Once our new data frame has been created by `ddply()`, we can use it for plotting just like we would any other data frame. Here are plots showing the trends in name length and maximum proportion over time.

```r
qplot(year, avg.length, data = BnameSummaries, color = sex)
qplot(year, max.prop, data = BnameSummaries, color = sex)
qplot(year, vowelFrac, data = BnameSummaries, color = sex)
qplot(year, vowelFrac2, data = BnameSummaries, color = sex)
```

5.3 A trick for showing multiple plots together

Sometimes it is nice to place multiple plots together. One way to do this is to have each plot be a facet of a larger plot. We create the faceting by adding a different value of some new variable (`plot`) in this example to the data frame passed to each layer.

```r
ggplot( aes(colour=sex), data=BnameSummaries) +  # stuff common to all layers
    geom_point(aes(x=year, y=avg.length),
               data=transform(BnameSummaries, plot="ave. length")) +
    geom_point(aes(x=year, y=max.prop),
               data=transform(BnameSummaries, plot="max. proprtion")) +
    geom_point(aes(x=year, y=vowelFrac),
               data=transform(BnameSummaries, plot="vowel fraction (by name)")) +
```

©2013 • Randall Pruim
Earlier we saw that the number of girls names surpassing 1% popularity has been going down. Now we see that the maximum proportion is also going down over time. This suggests that more and more names are being used – driving down the proportions, including for the most popular names.

### 5.4 How many girls have popular names?

Now we can return our question of how many girls have names shared with at least 1% of the girls born in the same year they were born. We’ll call these popular names.
As we see, since about 1950 it has become steadily less popular to give girls “popular” names.

### 5.5 \texttt{ddply()} and \texttt{transform()}

We’ve been using \texttt{ddply()} with \texttt{summarise()}, but it can be used with other functions as well. In particular, we can use it with \texttt{transform()} to add new variables to a data frame that are computed relative to its subsets. For example, lets compute the rank of each name, each year (separately for each sex) and add that to the original data set.

```r
Bnames4 <- ddply(Bnames, .(year, sex), transform, rank = rank(-prop))
```

It’s a good idea to do a sanity check:

```r
# every rank should occur the same number of times, but ties (especially near the bottom due to
# rounding, make it not quite what we expect.
qplot(rank, data = Bnames4, binwidth = 10, fill = sex)
# make sure the relationship between prop and rank is monotonic.
qplot(prop, rank, data = subset(Bnames4, year == 2000), colour = sex, geom = "line")
```

Now we can see how a names rank changes over time:

```r
qplot(year, prop, data = subset(Bnames4, sex == "boy" & name == "John"), geom = "line")
qplot(year, rank, data = subset(Bnames4, sex == "boy" & name == "John"), geom = "line") + scale_y_reverse()
qplot(year, prop, data = subset(Bnames4, sex == "boy" & name == "John"), geom = "line", colour = rank) +
  guides(colour = guide_colourbar(reverse = TRUE))
qplot(xmin = year - 0.5, xmax = year + 0.5, ymin = 0, ymax = prop, data = subset(Bnames4, sex == "boy" & name == "John"), fill = rank, geom = "rect") +
  guides(colour = guide_colourbar(reverse = TRUE))
```
Reversing the direction of the scale for rank makes it clearer that John is has been dropping from its number 1 rank and is in danger of falling out of the top 20.

6 Using merge() to combine data frames

Sometimes data for individual observational units is located in multiple data frames and this data must be brought together. The merge() function enables this. In the fastR package, the fusion1 data set contains genotype information (for one SNP) from subjects in a genome-wide association study and the pheno data set contains phenotype information for the same individuals.

```r
require(fastR)
head(fusion1, 3)
##  id marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose
## 1 9735 RS12255372 1 3 3 GG 0 0 2 0
## 2 10158 RS12255372 1 3 3 GG 0 0 2 0
## 3 9380 RS12255372 1 3 4 GT 0 0 1 1

head(pheno, 3)
## id t2d bmi sex age smoker chol waist weight height whr sbp dbp
## 1 1002 case 32.86 F 70.76 former 4.57 112.0 85.6 161.4 0.9868 135 77
## 2 1009 case 27.39 F 53.92 never 7.32 93.5 77.4 168.1 0.9397 158 88
## 3 1012 control 30.47 M 53.86 former 5.02 104.0 94.6 176.2 0.9327 143 89
```

The id variable in each data frame contains an identifier that allows us to combine the data sets if we want to compare genotype and phenotype information.

```r
gwa <- merge(fusion1, pheno, by = "id", all.x = FALSE, all.y = FALSE)
head(gwa, 3)
##  id marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose t2d bmi sex age
## 1 1002 RS12255372 1 3 3 GG 0 0 2 0 case 32.86 F 70.76
## 2 1009 RS12255372 1 3 3 GG 0 0 2 0 case 27.39 F 53.92
## 3 1012 RS12255372 1 3 4 GT 0 0 1 1 control 30.47 M 53.86
```

©2013 • Randall Pruim
The arguments `all.x=FALSE` and `all.y=FALSE` instruct R to maintain a record only for individuals that appear in both data sets. Setting one or both of these to `TRUE` would include individuals with information in one file but not the other (and fill in with `NA` where information is missing from the other data set). If the identifying variable has a different name in each data frame, then `by.x` and `by.y` can be used to give the separate names for each.

Now we can see how various phenotypes might differ by sex and genotype.

```r
ggplot(aes(x = genotype, y = whr), data = gwa, groups = genotype) + geom_boxplot() + facet_grid(~sex)
```

We can check on those warning messages with one of the following, which show that our warning messages are due to individuals who did not have recorded `whr` values in our data set.

```r
ddply( gwa, .(sex), summarize, missing.whr=sum(is.na(whr)) )
```

```r
ddply( gwa, .(sex), function(x){ favstats(x$whr)} )
```

The following table shows that the risk for type 2 diabetes appears to be higher for individuals with more copies of the T allele at this marker:

```r
gwa2 <- ddply ( gwa, .(sex,genotype), summarise, 
  diabetic = sum(t2d=="case"), 
  nondiabetic=sum(t2d=="control"),
  prop.diabetic = diabetic / (diabetic + nondiabetic) 
)
gwa2
```

©2013 • Randall Pruim
7 Other Examples

One difficulty in analysing baby names is that there are many spellings for some names and many names that are quite similar. (There may be some errors in the data as well. Perhaps ‘Ann’ has never actually been a top 1000 name for boys, for example.)

```r
AnnNames <- subset(Bnames, letter(name, 1, 3) == "ann")
tally(~sex + name, data = AnnNames, format = "count")
```

<table>
<thead>
<tr>
<th>name</th>
<th>sex</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>boy</td>
<td>1</td>
</tr>
<tr>
<td>Anna</td>
<td>boy</td>
<td>38</td>
</tr>
<tr>
<td>Annabel</td>
<td>boy</td>
<td>0</td>
</tr>
<tr>
<td>Annabella</td>
<td>boy</td>
<td>8</td>
</tr>
<tr>
<td>Annabelle</td>
<td>boy</td>
<td>85</td>
</tr>
<tr>
<td>Annamae</td>
<td>boy</td>
<td>11</td>
</tr>
<tr>
<td>Annabel</td>
<td>girl</td>
<td>32</td>
</tr>
<tr>
<td>Annabella</td>
<td>girl</td>
<td>129</td>
</tr>
<tr>
<td>Annabelle</td>
<td>girl</td>
<td>62</td>
</tr>
<tr>
<td>Annabel</td>
<td>girl</td>
<td>49</td>
</tr>
<tr>
<td>Annabelle</td>
<td>girl</td>
<td>8</td>
</tr>
<tr>
<td>Annamae</td>
<td>girl</td>
<td>85</td>
</tr>
<tr>
<td>Anneliese</td>
<td>girl</td>
<td>11</td>
</tr>
<tr>
<td>Ann</td>
<td>Total</td>
<td>130</td>
</tr>
<tr>
<td>Anna</td>
<td>Total</td>
<td>167</td>
</tr>
<tr>
<td>Annabel</td>
<td>Total</td>
<td>62</td>
</tr>
<tr>
<td>Annabella</td>
<td>Total</td>
<td>49</td>
</tr>
<tr>
<td>Annabelle</td>
<td>Total</td>
<td>8</td>
</tr>
<tr>
<td>Annamae</td>
<td>Total</td>
<td>85</td>
</tr>
<tr>
<td>Annabel</td>
<td>Total</td>
<td>11</td>
</tr>
<tr>
<td>Annabella</td>
<td>Total</td>
<td>30</td>
</tr>
<tr>
<td>Annabel</td>
<td>Total</td>
<td>32</td>
</tr>
<tr>
<td>Annabelle</td>
<td>Total</td>
<td>129</td>
</tr>
<tr>
<td>Annamae</td>
<td>Total</td>
<td>1</td>
</tr>
</tbody>
</table>

One solution to this is to group names, for example by some prefix. Here we create a function that shows the popularity of all names with a specified prefix. Creating functions that make it easy to create many similar plots is a good illustration of the DRY principle (Don’t Repeat Yourself) and is much better coding practice than copying and pasting code several time and making light edits in each one. In particular, this makes it much easier to make uniform changes across a sequence of similar plots.

```r
beginningNamesPlot <- function(data, prefix, legend = FALSE) {
  Names <- ddply(subset(data, letter(name, 1, nchar(prefix)) == tolower(prefix)), .(year, name, sex), .drop = FALSE, summarise, prop = sum(prop))
  numNames <- length(unique(Names$name))
  qplot(year, prop, fill = name, colour = I("transparent"), data = Names, geom = "area", stat = "identity",
        + scale_y_continuous(labels = percent) + scale_fill_manual(values = rep(brewer_pal(pal = "Blues")(8),
          ceil(numNames/8))) + facet_grid(sex ~ .) + labs(title = paste("Popularity of names beginning",
            prefix)) + guides(fill = legend)
}
```

These plots would be even nicer if we gussied them up with a little bit of interactivity so that mousing over the regions will reveal the particular name represented. See `gridSVG` and `SVGAnnotation` for two packages that provide the capability to create such plots. Also, `ggvis` is coming soon and should make this much easier for people who know ggplot2 since it is being designed and written by Hadley Wickham. But even as static plots, these are quite nice.
From the plots below we can see the rise and fall of names beginning ‘An’ and ‘Ed’. We can also see that the mix of names beginning ‘An’ has changed some over time.

We can also see that the mix of names beginning ‘An’ has changed some over time.
**dope**, *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent[1]

This week’s dope

This we will learn how to polish plots by

1. Customizing axis labels and plot titles
2. Customizing plots with themes and theme elements
3. Selecting specific mappings of colors, shapes, line types, etc. to aesthetics.

This Dope Sheet includes terse descriptions and examples of the main things covered this week. See the other course materials and the ggplot2 book for more complete descriptions and additional examples. But note that there have been a number of changes to ggplot2 since the book was published that affect themes and plot customization work. Some of these changes are discussed at https://github.com/wch/ggplot2/wiki/New-theme-system.

1 Modifying guides using `labs()`

Perhaps the most common change you will want to make to an individual plot is changing the plot title and default labeling of the guides (the collective term for axes and legends or keys used for other aesthetics). These can be changed using `labs()`.

```r
p <- qplot( cty, hwy, color=class, size=factor(cyl), data=mpg, alpha=I(.3) )
p
q <- p + labs(title="Mileage for popular models of cars",
             x = "city fuel economy (mpg)",
             y = "hiway fuel economy (mpg)",
             color = "Class of car",
             size = "No. of cylinders")
q
```

![Mileage for popular models of cars](image)

---

[1] definitions selected from Webster’s online dictionary

©2013 • Randall Pruim
2 Selecting a theme

Themes affect the way various plot elements look. Using themese to control these elements makes it easier to achieve a consistent look across multiple plots. The default theme is produced by `theme_gray()`. `theme_bw()` and `theme_minimal()` produce alternative themes; and the `ggthemes` packages contains a number of other themes.

A theme is really just a list of settings.

```r
theme_gray
## function (base_size = 12, base_family = "")
## {
##   theme(line = element_line(colour = "black", size = 0.5, linetype = 1,
##     lineend = "butt"), rect = element_rect(fill = "white",
##     colour = "black", size = base_size, hjust = 0.5,
##     vjust = 0.5, angle = 0, lineheight = 0.9), axis.text = element_text(size = rel(0.8),
##     colour = "grey50"), strip.text = element_text(size = rel(0.8)),
##     axis.line = element_blank(), axis.text.x = element_text(vjust = 1),
##     axis.text.y = element_text(hjust = 1), axis.ticks = element_line(colour = "grey50"),
##     axis.title.x = element_text(), axis.title.y = element_text(angle = 90),
##     axis.ticks.length = unit(0.15, "cm"), axis.ticks.margin = unit(0.1,
##     "cm"), legend.background = element_rect(colour = NA),
##     legend.margin = unit(0.2, "cm"), legend.key = element_rect(fill = "grey95",
##     colour = "white"), legend.key.size = unit(1.2, "lines"),
##     legend.key.height = NULL, legend.key.width = NULL, legend.text = element_text(size = rel(0.8)),
##     legend.text.align = NULL, legend.title = element_text(size = rel(0.8),
##     face = "bold", hjust = 0), legend.title.align = NULL,
##     legend.position = "right", legend.direction = NULL, legend.justification = "center",
##     legend.box = NULL, panel.background = element_rect(fill = "grey90"),
##     panel.border = element_blank(), panel.grid.major = element_line(colour = "white"),
##     panel.grid.minor = element_line(colour = "grey95", size = 0.25),
##     panel.margin = unit(0.25, "lines"), strip.background = element_rect(fill = "grey80"),
##     strip.text.x = element_text(), strip.text.y = element_text(angle = -90),
##     plot.background = element_rect(colour = "white"), plot.title = element_text(size = rel(1.2)),
##     plot.margin = unit(c(1, 1, 0.5, 0.5), "lines"), complete = TRUE)
## }
## <environment: namespace:ggplot2>
```

You can create your own theme by defining a similar sort of function with your favorite settings. The `ggthemes` package contains several additional themes, each intended to mimic the look of the plots from some other context (Excel, the economist, etc.)

```r
require(ggthemes)
q + theme_excel()
```
The themes in `ggplot2` take two arguments that control the size of the fonts and the font family used.

```r
q + theme_gray(base_size = 14, base_family = "serif")
q + theme_gray(base_size = 10, base_family = "Times")
```

The default theme can be set using `theme_set()`, which returns the previous theme, in case you want to revert back to it later.

```r
old_theme <- theme_set(theme_minimal())
q
theme_set(old_theme)
q
```
### 3 Modifying theme elements

In addition to selecting a complete theme, `ggplot2` allows you to set individual theme elements either for individual plots or to modify a complete theme. For example, to turn off the labeling of the x axis, we set `axis.text.x`, `axis.title.x`, and `optionaxis.ticks.x` each to `element_blank()`.

```r
q + theme(axis.text.x = element_blank(), axis.title.x = element_blank(),
          axis.ticks.x = element_blank())
```

Alternatively, we might like to change the style of the axis labeling using `element_text()` and `element_line()`. In the example below different colors are used to make it clear which element is which.

```r
q + theme( axis.text.x = element_text(colour='red', size=14, face='italic'),
          axis.title.x = element_text(colour='navy', size=16, face='bold', family='serif'),
          axis.ticks.x = element_line(colour='orange', size=1)
)```
Changes to theme elements can be used to update the current default theme using `theme_update()`.

```r
theme_update(axis.text.x = element_text(colour='red', size=14, face='italic'),
             axis.title.x = element_text(colour='navy', size=16, face='bold', family='serif'),
             axis.ticks.x = element_line(colour='orange', size=1)
)
```

In addition to `element_blank()`, `element_line()`, and `element_text()`, there is also `element_rect()` for setting features of rectangular elements (like the plot background).

```r
apropos("element_")
```

## [1] "element_blank" "element_line" "element_rect"
## [4] "element_text"

The documentation for these functions gives a complete list of their arguments.

### 3.1 Taking advantage of the hierarchy

The hierarchy of theme element names is illustrated in this graph. (The R code that generates this graph can be found at [https://github.com/wch/ggplot2/wiki/New-theme-system](https://github.com/wch/ggplot2/wiki/New-theme-system))
Elements with longer names inherit from elements with shorter (substring) names. This allows us to make many changes with a small amount of code. For example, if we wanted to make all of the text in our plot orange, we could set the `colour` attribute of the `text` element of our theme.

```r
q + theme(text = element_text(colour = "orange"))
```

To understand why this first attempt didn’t quite work, we need to look a little more closely at themes and inheritance.

```r
current_theme <- theme_update()  # don't make any changes, just store the current theme
current_theme$axis.text.x
```
The reason that `axis.text.x` didn’t inherit orange from `text` is that the colour had already been changed from `NULL` (meaning inherit from parent elements) to "red". Inheritance only occurs where there is a `NULL`. The following gets us a little closer.

```r
q + theme_gray()
q + theme_gray() + theme(text = element_text(colour = "orange"))
```

All the text is orange now except for `axis.text`. Inspecting `theme_gray()` shows us why.

```r
theme_gray()$text$colour
## [1] "black"
theme_gray()$axis.text$colour
## [1] "grey50"
theme_gray()$axis.text.x$colour
## NULL
theme_gray()$axis.title$colour
## NULL
theme_gray()$axis.title.x$colour
## NULL
```
In the `theme_gray()` theme, the colour in `axis.title` is NULL but in `axis.text` it is "gray50". So if we start from `theme_gray()`, we either need to first change `axis.text$color` to NULL or we need to set `axis.text$color` explicitly.

```r
q + theme_gray() + theme(text = element_text(colour = "orange"), axis.text = element_text(colour = "orange"))
```

If inheritance doesn’t work as you expect, inspecting the theme you are modifying will usually reveal what is blocking the inheritance. Just remember that an element inherits from its parent if its value is NULL, but not if the value is anything else.

### 3.2 Saving your theme

If you want to apply a theme to several plots, it is best to save the theme.

```r
require(scales)  # to get the alpha() function

## Loading required package: scales

theme_orange <- theme_gray() + theme(text = element_text(colour = "orange"),
axis.text = element_text(colour = "orange"),
panel.grid.major = element_line(colour = "orange"),
panel.grid.minor = element_line(colour = "orange"),
plot.background = element_rect(colour = alpha("orange", 0.2)),
panel.background = element_rect(fill = alpha("orange", 0.2)))

q + theme_orange
```

The resulting theme can then be applied as the new default theme, or it can be added to individual plots.
4 Controlling how aesthetics are mapped

Unlike lattice, ggplot2 does not include the color schemes, line widths and types, plot symbols, etc. in its themes. These are set by choosing the appropriate scales for the aesthetics as we saw in Lesson 2.

```
theme_set(theme_gray())
p2 <- ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species))
p2 + geom_point() + scale_color_manual(values = c("orange", "purple", "skyblue"))
p2 + geom_point(size = 5, alpha = 0.5) + scale_color_brewer(type = "qual", palette = 1, name = "iris species")
```

There is no mechanism in ggplot2 for setting default scales as part of a theme, but the scales can be saved for easier reuse in multiple plots.

```
my_color_scale <- scale_color_brewer(type = "qual", palette = 1, name = "iris species")
q + my_color_scale
p2 + geom_point() + my_color_scale
```
5 Annotation

**ggplot2** provides some additional utilities for adding small amounts of additional information to a plot. Like many other **ggplot2** functions we have seen, `annotate()` adds a layer to the plot. Its first argument is a geom. Unlike the other **ggplot2** functions, the `annotate()` function does not require that data be in a data frame, since for adding small amounts of information, it is often easier to work with vectors.

```r
g + annotate("text", x = 10, y = 40, label = "We can add some text", hjust = 0) + annotate("rect", xmin = 20, ymin = 15, xmax = 30, ymax = 20, fill = alpha("navy", 0.2)) + annotate("text", x = 25, y = 17.5, label = "or a rectangle")
```

Inset plots can be added using `annotation_custom()`.

```r
inset <- ggplot(aes(cty), data = mpg) + stat_density() + theme_bw(6)
insetG <- ggplotGrob(inset)
g + annotation_custom(insetG, xmin = 22, xmax = 35, ymin = 12, ymax = 25)
```
There is also an `annotation_raster()` function for placing raster graphics onto a plot.