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. Describe plots using the grammar of graphics
2. Make basic plots using `ggplot()` and `qplot()`
3. Use aesthetics to enhance plots
4. Use non-default geoms and stats to increase the palette of plots available
5. 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.*

0 Preliminaries

0.1 Resources

In addition to these Dope Sheets, here are some other useful resources you may wish to consult as you learn `ggplot2`.

- The disucssion forums at `http://statistics.com`.

  Take advantage of the discussion forums that will be set up for each week of the course. This is a place where you can ask individual questions and share your experience with `ggplot2`.

- *The R Graphics Cookbook* by Winston Chang

  Hadley Wickham's description: "[This book] provides a set of recipes to solve common graphics problems. Read this book if you want to start making standard graphics with `ggplot2` as quickly as possible."

  There are a few examples done using `lattice`, too.

- *ggplot2: Elegant Graphics for Data Analysis* by Hadley Wickham

  Hadley Wickahm's description: "[This book] describes the theoretical underpinnings of `ggplot2` and shows you how all the pieces fit together. This book helps you understand the theory that underpins ggplot2, and will help you create new types of graphic specifically tailored to your needs. You can read sample chapters and download the book code from the book website."

  Once caveat: This book is now a bit out of date and is not completely consistent with the newest versions of `ggplot2`. I'll try to point out some of the differences in the relevant sections of the course to avoid confusion.

[1] definitions selected from Webster's online dictionary

- http://docs.ggplot2.org/

  This site has some nicely formatted documentation for each component in the ggplot2 system. Thumbnail images can help you locate the right thing, and the examples include plots (unlike the documentation within R) so you can quickly scan to find an example that matches your needs.

0.2 R and RStudio

I strongly recommend that you use

- an up-to-date version of R

  Some of our examples may require R 3.1 or later. It is a good idea to update your R packages as well. (You can use `update.packages()` to automate the process.)

- an up-to-date version of RStudio

  If you have never used RStudio before, you can learn RStudio as bonus material for this course. RStudio is an integrated development environment (IDE) for R that runs on Windows, Mac, and Linux. (It can even be run in a browser if you have access to an RStudio server). It simplifies the creation and management of files, and generally organizes your work in R much better than the various alternatives. I've been using it for 4 or 5 years now and can't imagine going back to use other interfaces.

- RMarkdown

  RMarkdown provides an easy way to create documents that include text, R code, R output, and graphics. RMarkdown is a simple mark-up language that can be used to generate HTML, PDF, or Word documents in a reproducible workflow. RStudio makes it very easy to work with RMarkdown. RMarkdown provides the easiest way to do your assignments for this course, but its uses extend well beyond this and should be a part of every R users workflow.

  Find out more about RMarkdown at http://rmarkdown.rstudio.com/

  Notes: To create PDF documents, you must have LaTeX installed on your computer. For LaTeX users among you, there is also a way to mix R with LaTeX to give you more control over the output formatting. That's how these notes were created.

0.3 Packages and Data

We will be using the ggplot2 and dplyr packages throughout this class, so we should get in the habit of making sure they are loaded before we do anything else.[2] In addition, for these examples, we will use some data from the mosaicData package.

```
require(mosaicData)
require(ggplot2)
require(dplyr)
```

Additional packages will be introduced as needed. All of the packages used in this course can be installed via CRAN. RStudio provides a simple interface for doing this, but you can do it manually as well using, for example

[2] If you are using RMarkdown, remember that packages must be loaded in each RMarkdown file since the RMarkdown files do not have access to the console environment.

```
install.packages("mosaic")
```

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

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

```
##   age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female  sex g1b homeless
## 1  37            1    yes   49  3        177      225        0  NA      0 male yes   housed
## 2  37            1    yes   30 22          2       NA        0  NA      0 male yes homeless
## 3  26            1    yes   39  0          3      365       20  NA      0 male  no   housed
##   i1 i2 id indtot linkstatus link       mcs      pcs pss_fr racegrp satreat sexrisk substance
## 1 13 26  1     39          1  yes 25.111990 58.41369      0   black      no       4   cocaine
## 2 56 62  2     43         NA <NA> 26.670307 36.03694      1   white      no       7   alcohol
## 3  0  0  3     41          0   no  6.762923 74.80633     13   black      no       2    heroin
##   treat
## 1   yes
## 2   yes
## 3    no
```

Use

```
?HELPrct
```

to find out more about this data set. Some of the variables have pretty opaque names. Let's rename two variables to give better names for the average and maximum number of drinks per day over the 30 days prior to admission for substance abuse.

```
HELPrct <- rename(HELPrct, aveDrinks = i1, maxDrinks = i2)
head(HELPrct, 2)
```

```
##   age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female  sex g1b homeless
## 1  37            1    yes   49  3        177      225        0  NA      0 male yes   housed
## 2  37            1    yes   30 22          2       NA        0  NA      0 male yes homeless
##   aveDrinks maxDrinks id indtot linkstatus link      mcs      pcs pss_fr racegrp satreat
## 1        13        26  1     39          1  yes 25.11199 58.41369      0   black      no
## 2        56        62  2     43         NA <NA> 26.67031 36.03694      1   white      no
##   sexrisk substance treat
## 1       4   cocaine   yes
## 2       7   alcohol   yes
```

The `rename()` function creates a new data frame with some of the variables renamed. By assigning this new data frame to `HELPrct`, we have essentially updated `HELPrct` with different names for two of the variables. (As an alternative, we could have chosen to add two new variables without losing the original ones. The `mutate()` function can be used for this.)

We will also use the `Births78` data set.

```
head(Births78)
```

```
##         date births dayofyear  wday
## 1 1978-01-01   7701         1   Sun
## 2 1978-01-02   7527         2   Mon
## 3 1978-01-03   8825         3  Tues
## 4 1978-01-04   8859         4   Wed
## 5 1978-01-05   9043         5 Thurs
## 6 1978-01-06   9208         6   Fri
```

This data set records the number of live births in the United States for each day of 1978. If we ever modify a data set from a package and need to restore the original version of the data, we can use

```
data(HELPrct)        # (re)load the data from the package
data(Births78)
```

1 The Grammar of Graphics

ggplot2 is based on (but also different from) a **grammar of graphics** described in *The Grammar of Graphics* by Wilkenson *et al.* The ggplot2 approach is to build up layered graphics by describing the elements of the graph in a structured way. It helps to begin with a bit of the vocabulary of the ggplot2 grammar:

geom the geometric "shape" used to display data (other terminology: glyph, mark)

- bar, point, line, ribbon, text, etc.

aesthetic an attribute controlling how geom is displayed (other terminology: property)

- x position, y position, color, fill, shape, size, etc.

coordinate system (a.k.a. coord) Among the aesthetics, the x and y positions are special and get mapped to positions on the graphics device (e.g., computer screen, pieces of paper, etc.) using a coordinate system (other terminology: frame).

- x position, y position

statistic (a.k.a., stat) a transformation applied to data before a geom gets it

- example: histograms and bar charts are created from binned data rather than the original data

mapping the matching up of aesthetics with variables so that different values of a variable are represented by different values of the aesthetic.

scale conversion of raw data to visual display

- particular assignment of colors, shapes, sizes, etc.

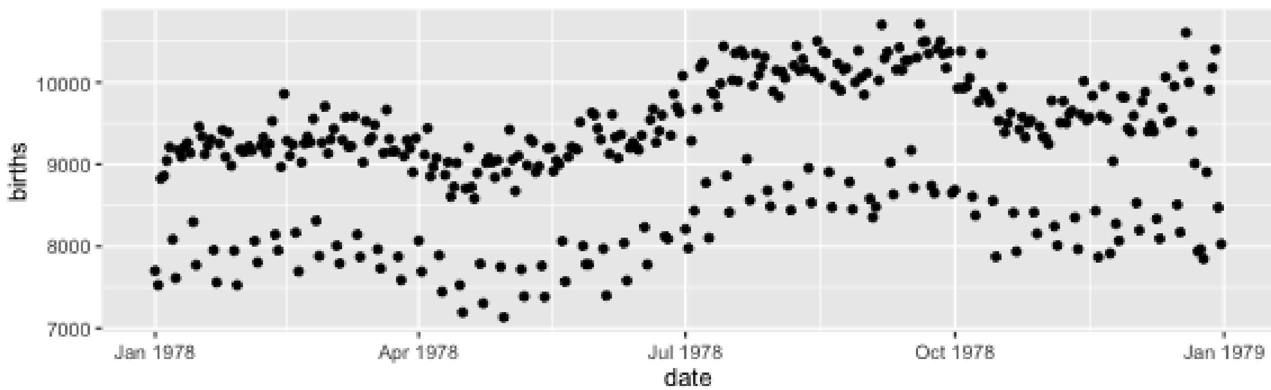
guide helps user convert visual data back into raw data

- legends, axes

annotation additional labeling (titles, arrows, etc.) can sometimes make the plot easier to read.

1.1 Describing a plot

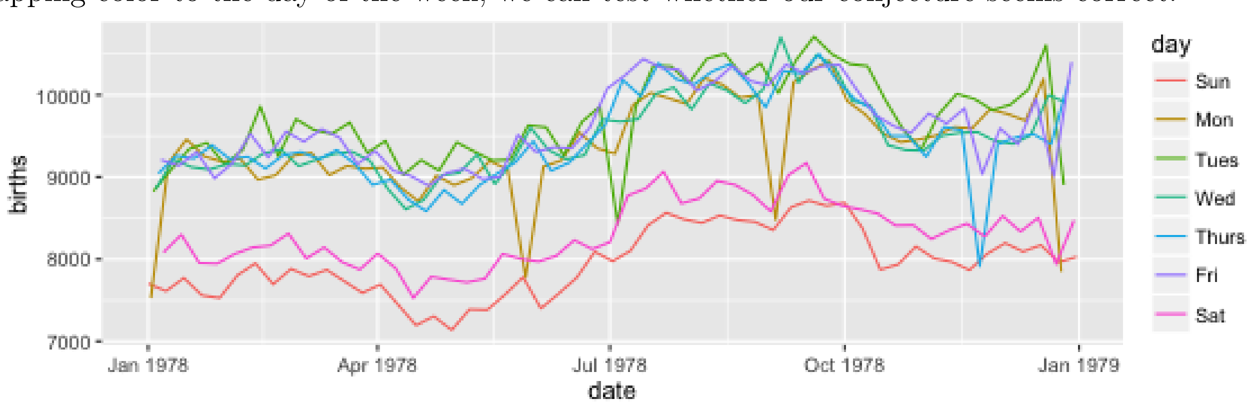
One key to successfully creating plots with ggplot2 is learning to describe plots using the grammar outlined above. For example, consider the following plot.



- **coordinate system**: The frame is determined by **mapping** `date` to the x-axis and `births` to the y-axis in the usual Cartesian coordinate system. (Note: ggplot2 is date aware and does the right thing with the dates on the x-axis.)
- **geoms**: Each row of our `Births78` data frame is represented by a point on the plot.
- **other aesthetics**: In this example no other variables are being mapped to aesthetics – all the dots are **set** to the same shape, size, color, transparency, etc.
- **statistic**: In this example (and in many others) the identity statistic is being used – the data are being mapped directly to positions on the plot without an additional transformation.
- **guides**: The only aesthetics being mapped are x and y. The axes on the plot serve as the guides that help us do the reverse mapping from a position on the plot onto values that occur in the data.

1.2 Describing another plot

We might conjecture that the reason for the two parallel waves is that fewer children were born on weekends. By mapping color to the day of the week, we can test whether our conjecture seems correct.



Compared to the previous plot, the following things have changed:

- The **geom** has changed from points to lines. (This makes it easier to detect days that are "out of place" relative to the overall pattern.)

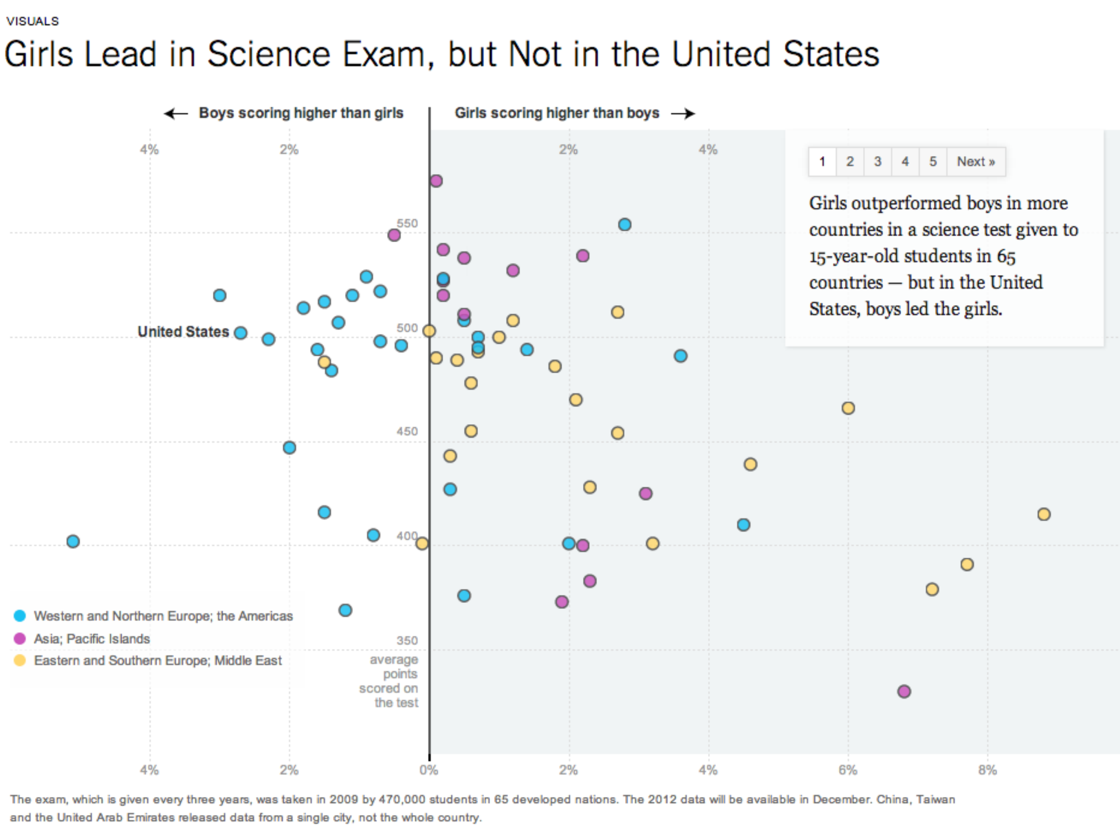
- We are now **mapping** the day of the week to the color **aesthetic**.
- An additional **guide** has beeen added to show which days are mapped to which colors. (The mapping takes days to colors for displaying and the guide helps us go from colors to days for interpreting the result.)

The result is a plot that confirms our suspiscion. With only a few exceptions, the days in the lower wave are weekends and the days in the upper wave are weekdays. (The exceptions are readily seen to coincide with major US holidays.)

In the next section we will begin learning how to describe the elements of plots such as these in a language that `ggplot2` understands. But the first step to creating plots in `ggplot2` is identifying its key components as outlined above. This is different from some other graphics systems, like `lattice`, where one begins by identifying the type of high level plot (histogram, bar chart, scatter plot, etc.) that one wants. This makes `ggplot2` more expressive and flexible, but can make getting started a bit more complicated.[3]

1.3 One more example

Now let's consider a more complicated example. This one comes from the *New York Times*.



- The **frame** for this graphic is produced by **mapping** the average score on a mathematics test to the y-axis and the difference in performance for boys and girls on the x-axis.

[3] `ggplot2` does provide a `qplot()` function that makes it easier to create several simple plots. We opted to delay introducing `qplot()` to make sure we understand the `ggplot2` system first.

- The **guide** for the y-axis is located in the center of the plot rather than on an edge. The **guide** for the x-axis is in the usual place.
- Each country is represented by a point. The region of the world is **mapped** to one of three *fills*. (In ggplot2 color usually refers to the outer color and fill to the internal color.)
- A **guide** for the mapping of region to fill is in the lower left corner.
- The color of each dot is **set** to black.
- Some additional **text annotations** have been added to indicate which dot represents the United States, to add an alternative guide for the x-axis (at the top of the plot), and to add some additional contextual information.
- Finally, the left and right halves of this plot are different colors. This could be done by placing semi-transparent rectangle **geom** over a portion of the plot.

Once you can look at a plot and identify it componenents like this, it will become relatively straightforward to recreate the plot using ggplot2 – we just need to learn the particulars of how we communicate the grammar of graphics to R. If you want some additional exercises, take a look at the idata visulizations of the *New York Times* and see if you can describe them using the vocabulary of this grammar of graphics.

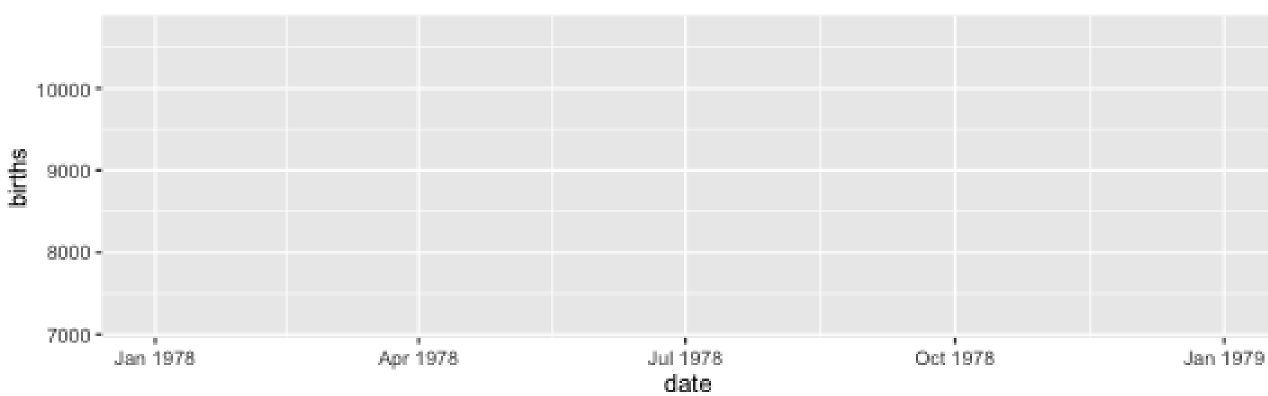
2 Describing Plots with `ggplot()`

2.1 Data and aesthetics

All plots begin with data. For ggplot2, the data must usually 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.[4]

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 variables in the data onto position (`x` and `y`), color, size, transparency, etc. when they are plotted.[5]

```
ggplot( Births78, aes(x=date, y=births) )
```



[4] Often, the largest part of the taks of creating a plot is getting the data into the correct shape. We'll talk more about how to do that in Week 3.

[5] The situation is actually a little bit more complicated. The mapping establishes the link between the data and an aesthetic, but a **scale** determines precisely which values of the data are mapped to which aesthetic values. For now, we will use the default scales, but eventually we will want to know how to have more control by determining the scales ourselves.

This alone doesn't show any plot, however, because `ggplot()` does not know about any default geoms or stats, so it doesn't know what marks to put at each `x` and `y` location until we add that information to our plot object.

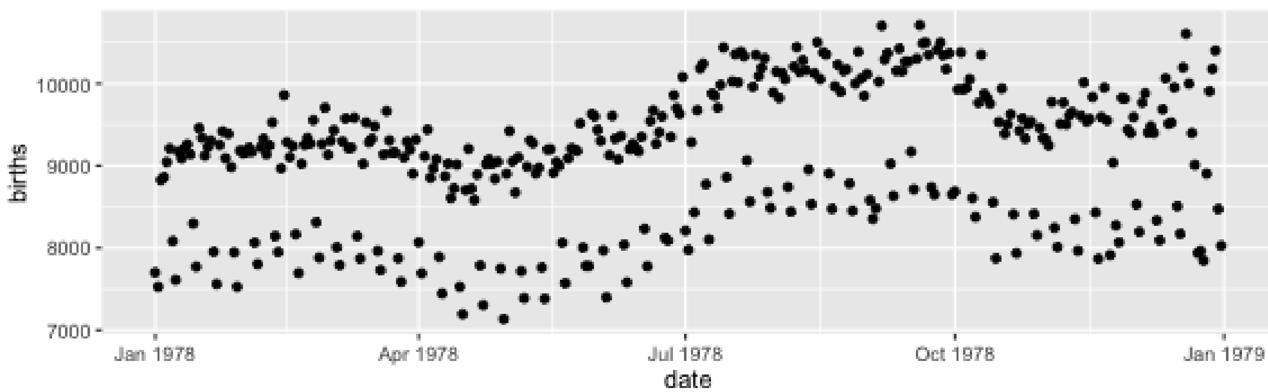
2.2 Layers, geoms, and stats

Each layer of a `ggplot2` plot requires data, aesthetics, a geom, and a a stat. To get a plot, we need to add information about our desired geom and stat for each layer:

```
# We will learn an easier way to do this in just a moment.
ggplot( data=Births78, aes(x=date, y=births) ) +
  layer( geom="point", stat="identity")

## Error:  Attempted to create layer with no position.
```

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, so in practice, we will essentially never use the `layer()` function. Instead we will do something like

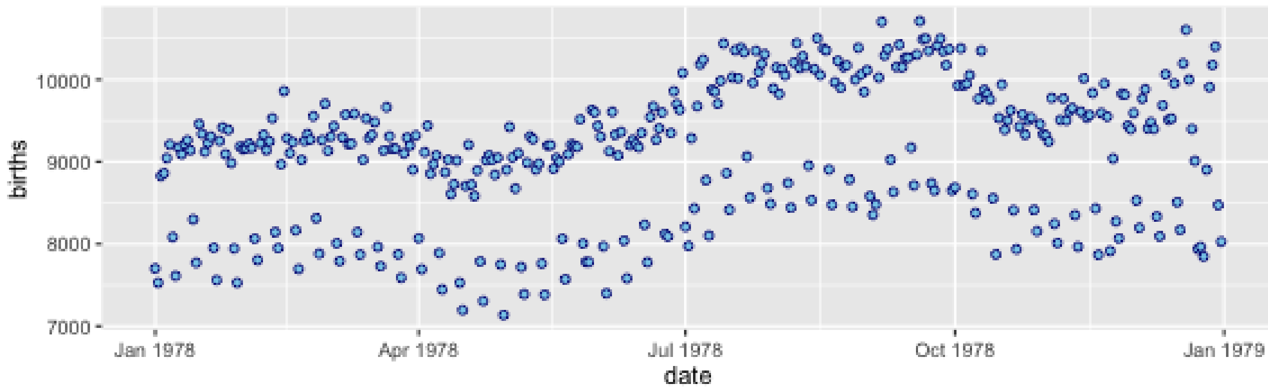
```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point()   # stat_identity is the default
```



2.3 Aesthetics: mapping and setting

We can make this a bit fancier by setting some of the aesthetics for the points on our plot.

```
# shape = 21 is a circle with separate color and fill aesthetics.
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point(color="navy", fill="skyblue", shape=21)
```



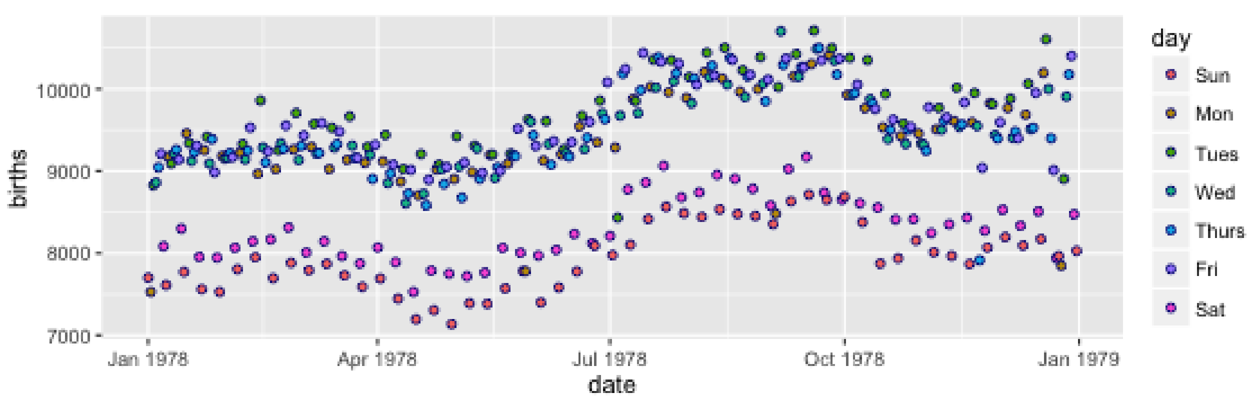
Our use of color, fill, and shape here do not code any additional information. We have simply set them to different values to change the overall look of the plot.

If we want the fill to show the days of the week, we must use **mapping** rather than **setting**. To make things easier, let's add a variable to our data that stores the day of the week. The `wday()` function from the `lubridate` package will do the computation of weekday from date and `mutate()` creates a new data frame that includes the additional variable.

```
require(lubridate)
Births78 <- mutate(Births78, day=wday(date, label=TRUE, abbr=TRUE))
head(Births78, 2)
```

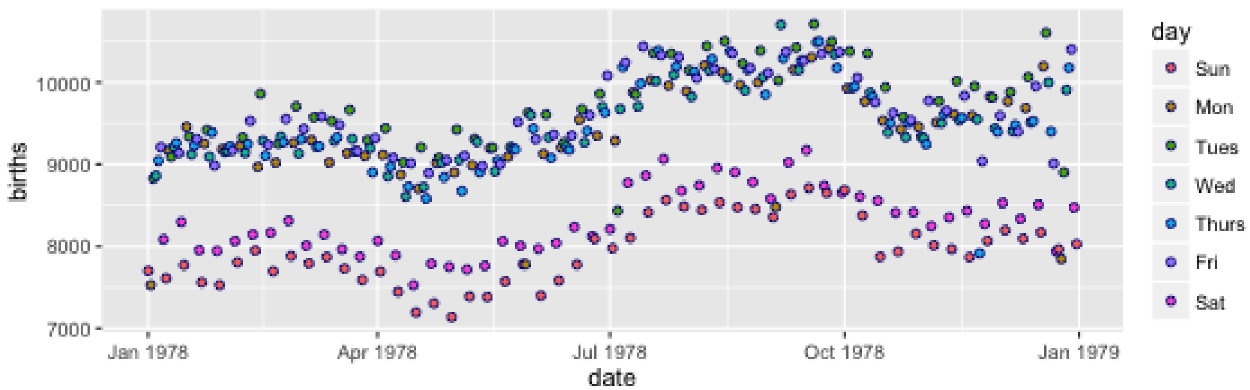
```
##         date births dayofyear wday day
## 1 1978-01-01   7701         1  Sun Sun
## 2 1978-01-02   7527         2  Mon Mon
```

```
ggplot( Births78, aes(x=date, y=births, fill=day) ) +
  geom_point(color="navy", shape=21)
```



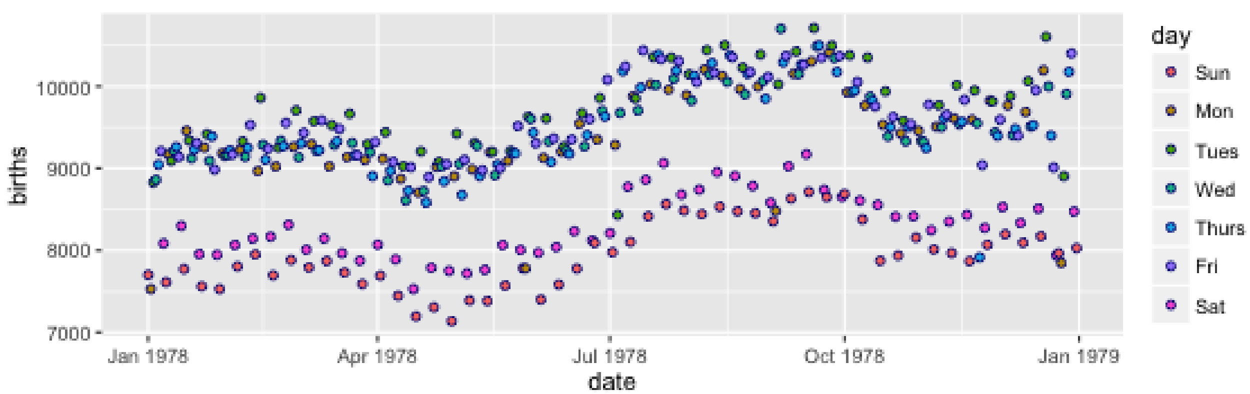
Notice that `fill=` is now inside `aes()` in the call to `ggplot()` rather than an argument to `geom_point()`. This is how we distinguish between **mapping** and **setting**. We could also have made the plot this way:

```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point(aes(fill=day), color="navy", shape=21)
```



or a bit more verbosely using

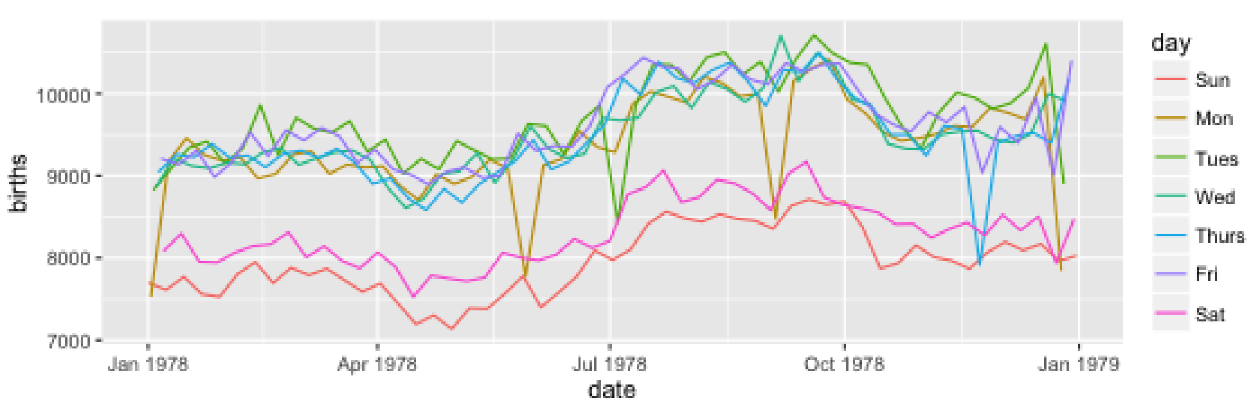
```
ggplot( data=Births78, mapping=aes(x=date, y=births) ) +
  geom_point( mapping=aes(fill=day), color="navy", shape=21 )
```



When there is only one layer in the plot, it does not matter whether the aesthetic mapping is defined in `ggplot()` or in `geom_point()`. When there are multiple layers, however, the mapping done in `ggplot()` will affect all of the layers, but the mapping done in each layer affects only that layer. The same is true for `data`, since different layers may use different data frames.

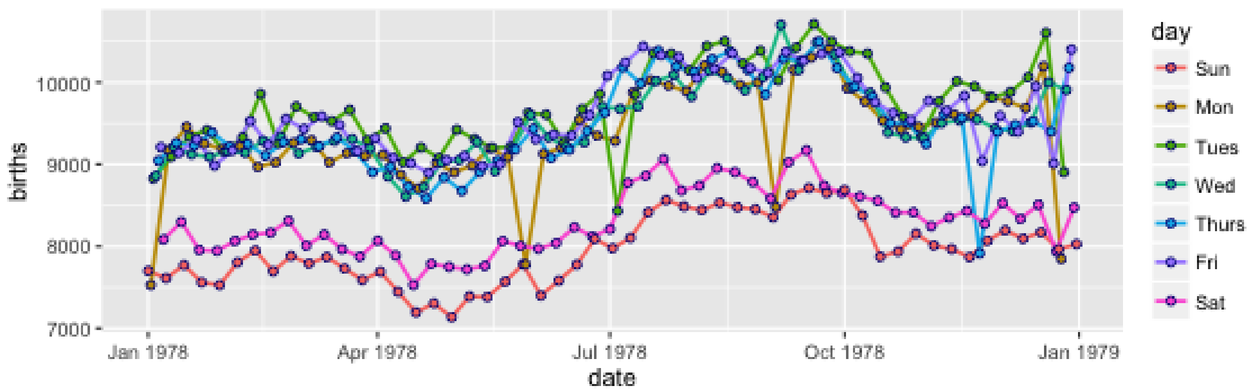
If we would prefer to use lines, rather than points, we can easily change the geom:

```
ggplot( Births78, aes(x=date, y=births, color=day) ) +
  geom_line()
```



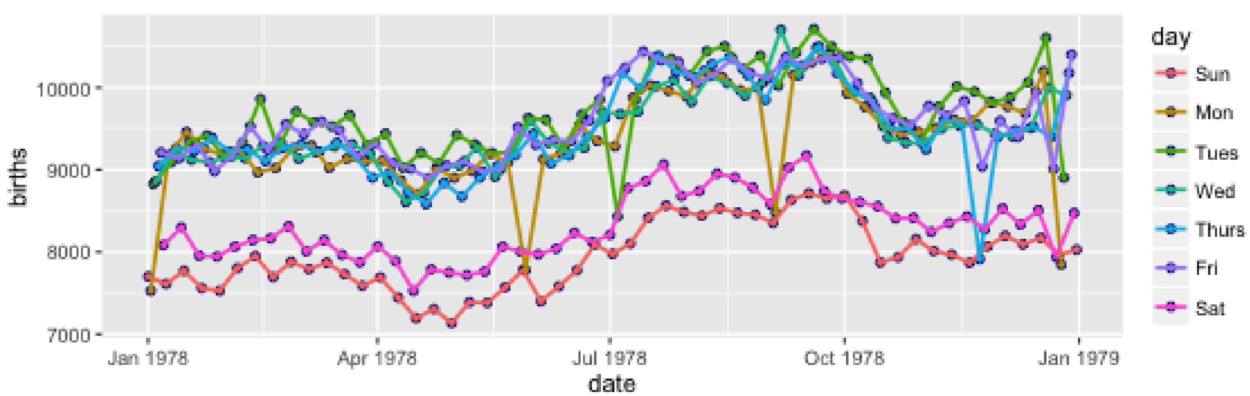
We can even choose to do both points and lines by creating two layers:

```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_line( aes(color=day), size=0.8 ) +
  geom_point( aes(fill=day), color="navy", shape=21 )
```



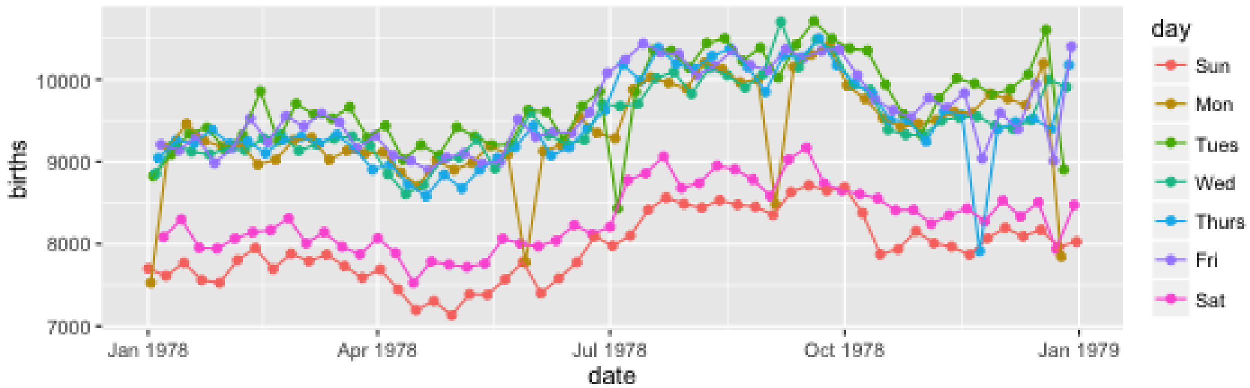
The order of the layers matters. Each layer is created *on top* of the previous layers.

```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point( aes(fill=day), color="navy", shape=21 ) +
  geom_line( aes(color=day), size=0.8 )
```



In this case, it is probably better to leave off the dots. The story is clearer with just the lines. Removing the border on the dots and making the lines a bit thinner would also improve the plot if we wanted to retain the dots.

```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point( aes(color=day) ) +
  geom_line( aes(color=day), size=0.5 )
```

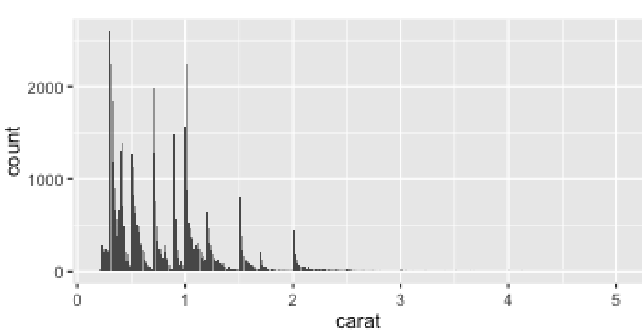


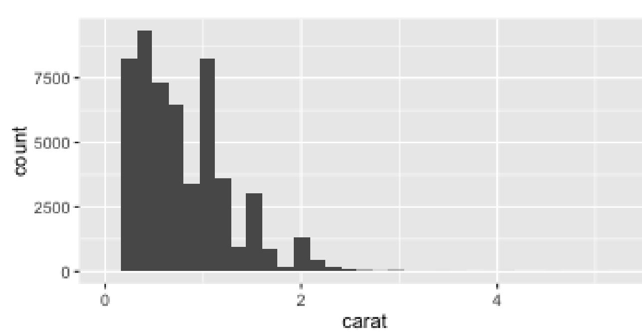
2.4 Stats

So far we have seen only two types of geoms – points and lines. Using the `diamonds` data set (in the `ggplot2` package), we we will illustrate some additional geoms.

We begin with a histogram of the size (carat) of the diamonds which we can make using `geom_bar()` or `geom_histogram()`, which is essentially another name for `geom_bar()`.

```
head(diamonds,2)


## Source: local data frame [2 x 10]
##
##   carat     cut  color clarity depth table price     x     y     z
##   (dbl)  (fctr) (fctr)  (fctr) (dbl) (dbl) (int) (dbl) (dbl) (dbl)
## 1  0.23   Ideal      E     SI2  61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium      E     SI1  59.8    61   326  3.89  3.84  2.31


ggplot(data=diamonds, aes(x=carat)) +
  geom_bar()
ggplot(data=diamonds, aes(x=carat)) +
  geom_histogram()

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```

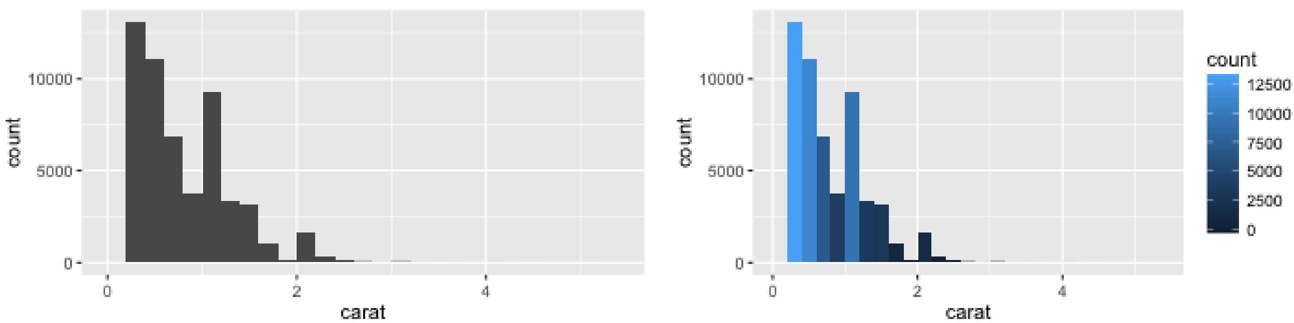




This is mostly straightforward. We have mapped `carat` to the x-axis and selected the histogram geom. We are even warned that it would be better if we selected the width of the bins ourselves rather than using the default value. We'll do that in our next histogram.

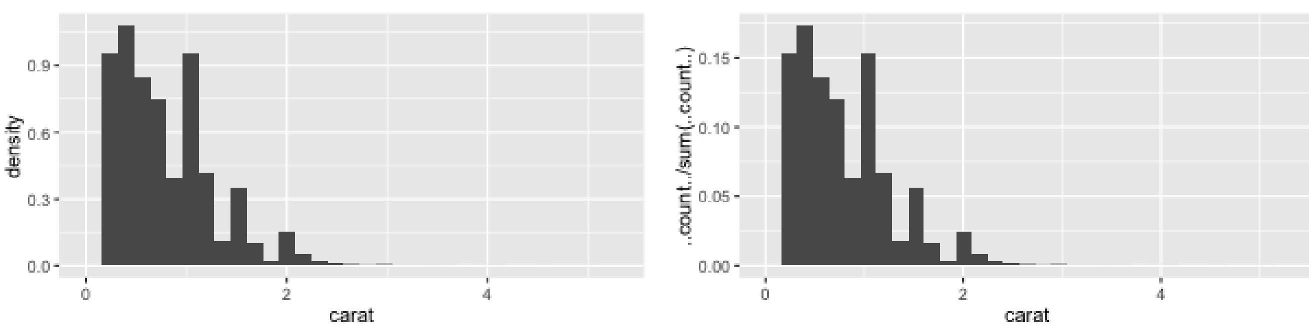
But what about the y-axis? In this case, the default stat is `stat_bin()`, which creates a new data frame with one row for each of the bins. A new variable `..count..` gives the number of observations in each bin. This happens *before* `geom_histogram()` draws the bars. We could have chosen to be explicit about the y-aesthetic, or we can use the new `..count..` in other ways:

```
ggplot(data=diamonds, aes(x=carat, y=..count..)) +
  geom_histogram(binwidth=.2)
ggplot(data=diamonds, aes(x=carat, fill=..count..)) +
  geom_histogram(binwidth=.2)
```



Other variables in the statified data frame allow us to create other types of histograms.

```
# density
ggplot(data=diamonds, aes(x=carat, y=..density..)) +
  geom_histogram()

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.

# proportions -- labeling is ugly, but well learn how to fix that later
ggplot(data=diamonds, aes(x=carat, y=..count../sum(..count..))) +
  geom_histogram()

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```

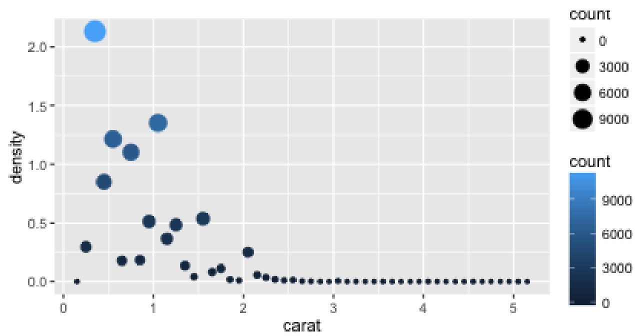


The list of additional variables computed by `stat_bin()` can be found with

```
?stat_bin
```

We end this section with a more unusual representation of the same information.

```
ggplot(data=diamonds, aes(x=carat, y=..density..)) +
     stat_bin(geom="point", aes(size=..count.., color=..count..), binwidth=0.1)
```



By now you should be sensing the power in ggplot2's flexible appraoch to describing graphics. By combining a few elements (data, aesthetics, geoms, and stats) in a variety of ways, we can create many useful (and some not so useful) plots.

2.5 Data flow

As a plot is created the original data set undergoes a sequence of transformations.

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

At each stage, the available data are stored in a data frame (actually, one data frame for each layer of each facet).[6] At each step the working data frame is transformed into a new data frame, and it is possible for new variables to be introduced along the way. This explains why the examples above work the way they do.

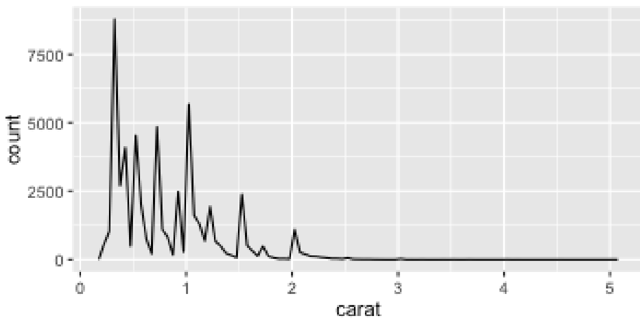
2.6 Some more geoms and stats

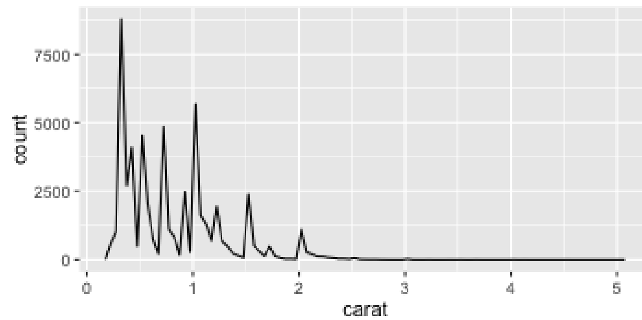
2.6.1 Frequency polygons

A histogram is not the only way to display the distribution of a quantiative variable. Frequency polygons are often preferable. Note that there is no special geom for a frequency polygon. Instead, we combine `geom_line()` with `stat_bin()`. Since neither is the default for the other, we must specify both.

```
# combining stat_bin() with geom_line() gives a frequency polygon.
ggplot(data=diamonds, aes(x=carat)) +
  geom_line(stat="bin", binwidth=0.05)
ggplot(data=diamonds, aes(x=carat)) +
  stat_bin(geom="line", binwidth=0.05)
```

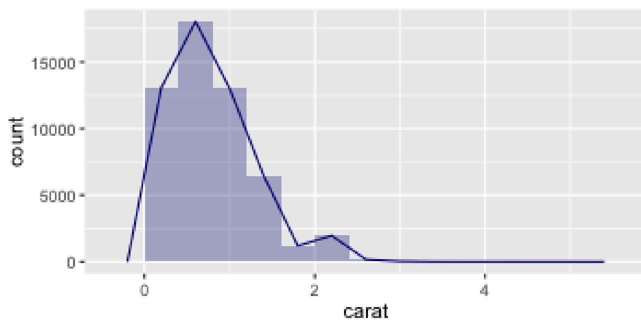
[6] This is a bit of an oversimplification. Actually, the aesthetics get computed twice, once before the stat and again after. For a histogram, for example, we need to look at the aesthetics to figure out which variable to bin (that's the stats job), but it isn't until after the binning that we will 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 ggplot2 book.





For those unfamiliar with frequency polygons, we can illustrate their connection to histograms by creating a plot with two layers. (We'll use fewer bins here so it is easier to see the connection between the two types of plots.)

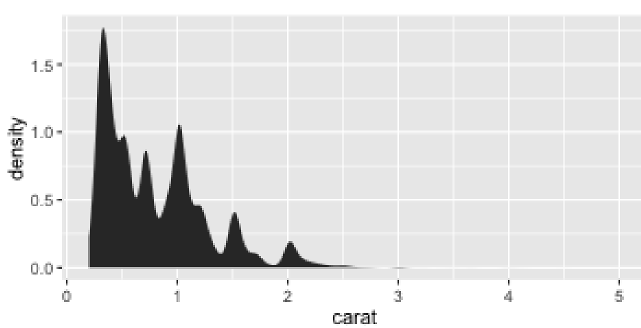
```
ggplot(data=diamonds, aes(x=carat)) +
  geom_histogram(alpha=.3, binwidth=0.4, fill="navy") +
  geom_line(stat="bin", binwidth=0.4, color="navy")
```



2.6.2 Density plots

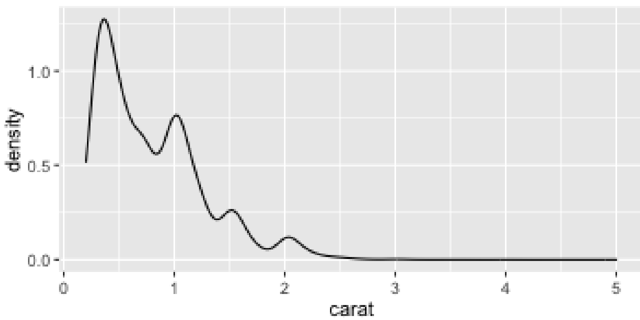
Another common representation of the distribution of a quantitative variable is a density plot. This requires a new stat: `stat_density()`.

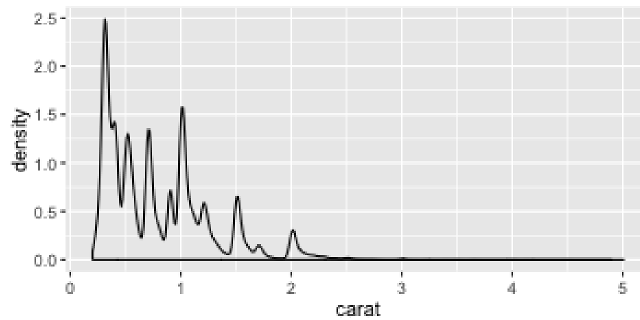
```
ggplot(data=diamonds, aes(x=carat)) +
  stat_density()
```



The default geom being used is `geom_area()`, but we could use `geom_line()` or `geom_density()` instead if we preferred. We can also use `adjust` to control how much smoothing takes place. This is roughly the equivalent of choosing a `binwidth` for a histogram. The default value for `adjust` is 1. Larger values indicate more smoothing.

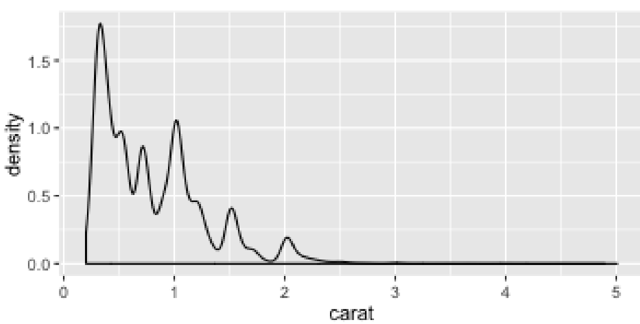
```
ggplot(data=diamonds, aes(x=carat)) +
  stat_density(geom="line", adjust=2)
ggplot(data=diamonds, aes(x=carat)) +
  stat_density(geom="density", adjust=0.5)
```

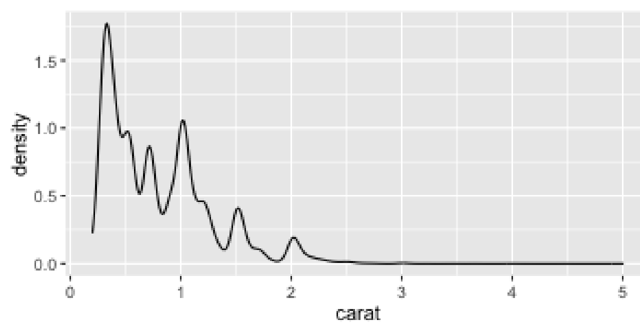


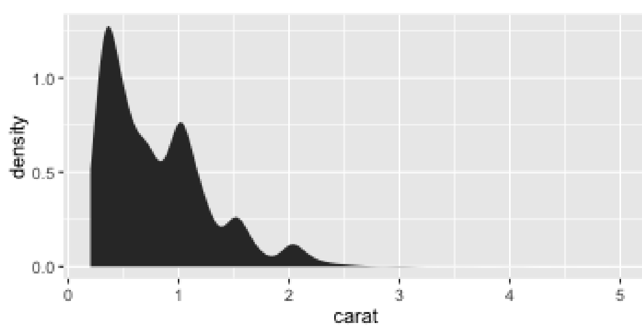


Alternatively, we could specify the geom and stat this way

```
ggplot(data=diamonds, aes(x=carat)) +
  geom_density()
ggplot(data=diamonds, aes(x=carat)) +
  geom_line(stat="density")
ggplot(data=diamonds, aes(x=carat)) +
  geom_area(stat="density", adjust=2)
```







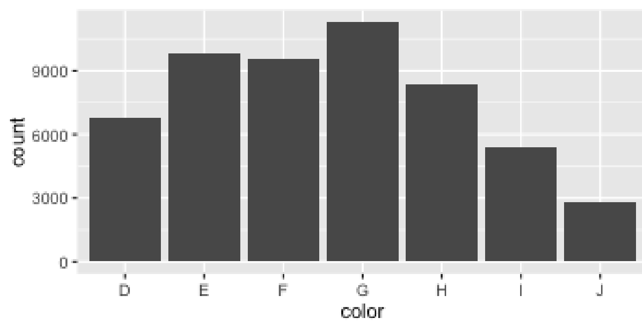
Whichever way we choose to do it, we are telling `ggplot()` both the stat and the geom that should be used.

In this particular case, we need to be cautious not to oversmooth and lose part of the story. We have a lot of data (53940 rows), so the spikey appearance of these plots is likely a feature and not an artifact. In fact, there is a plausible explanation: the peaks coincide with round numbers. Likely the diamond producers are targeting diamonds that are approximately 0.5 carat, 1.0 carat, etc. In fact, the peaks seem to be biased to be just a tad over these round numbers.

2.6.3 Bar charts

Bar charts are much like histograms but work with a categorical variable. A combination of `geom_bar()` and `stat_bin()` produces a bar chart.

```
ggplot(data=diamonds, aes(x=color)) +
  stat_bin()    # geom_bar is the default

## Error:  StatBin requires a continuous x variable the x variable is discrete.  Perhaps you want stat="count"

ggplot(data=diamonds, aes(x=color)) +
  geom_bar()    # stat_bin is the default
```

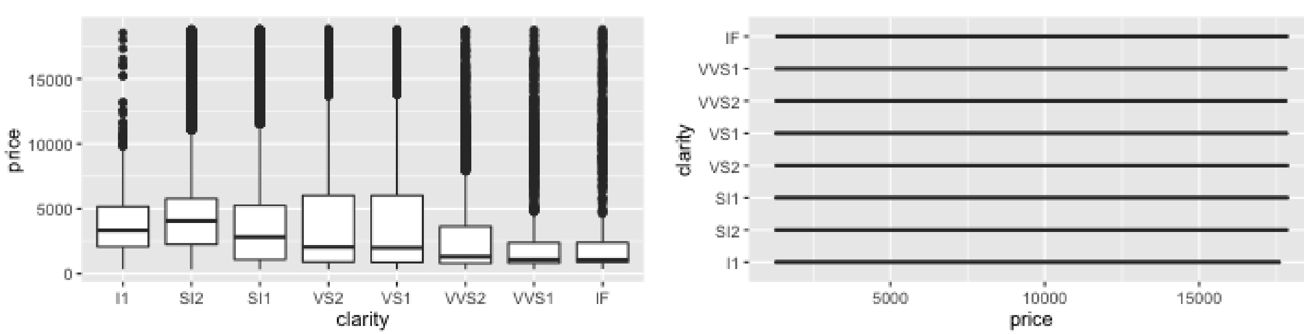


2.7 Boxplots

We'll illustrate one more geom before moving on to other things. Like histograms, boxplots involve a non-identity stat (`stat_boxplot()`) that transforms the data. In this case, there is also a special geom (`geom_boxplot()`) for rendering the plot.

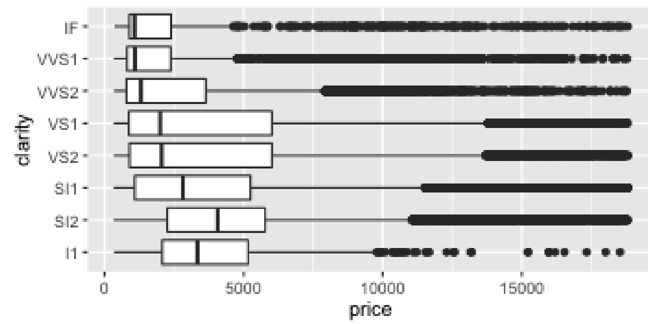
That's almost everything you need to know to make a boxplot. The other thing you need to know is that the variable being summarized with a boxplot must be the `y`-variable. This explains why the second of the plots below doesn't do anything useful.

```
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_boxplot()
# this doesnt do what we might have expected.
ggplot( data=diamonds, aes(y=clarity, x=price) ) +
  geom_boxplot()

## Warning:  position_dodge requires non-overlapping x intervals
```



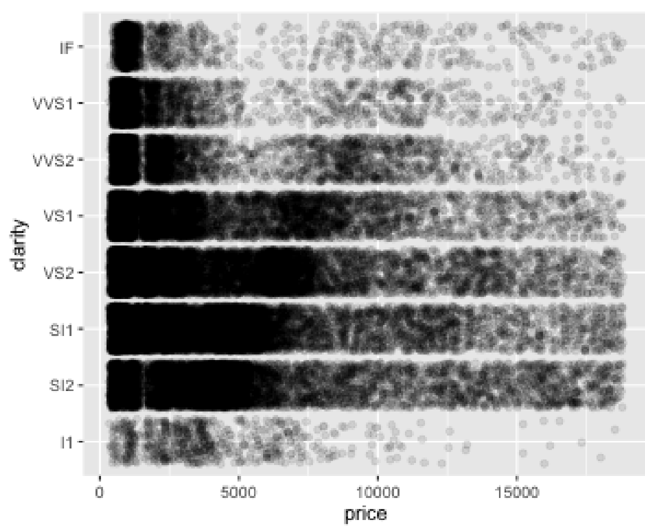
If we want horizontal boxplots, we need to flip them by using a different coordinate system.

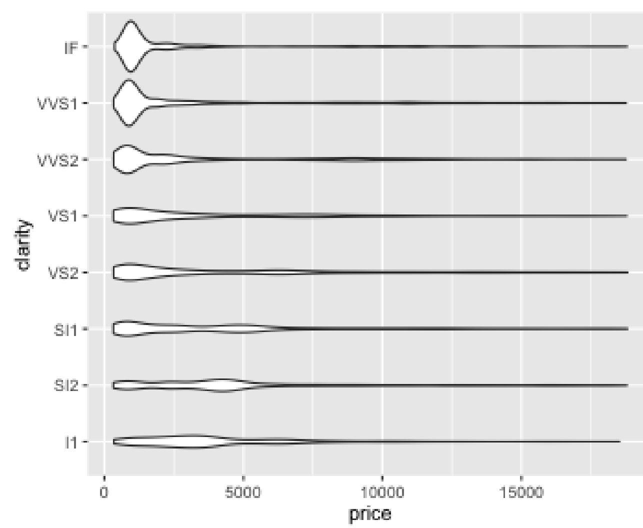
```
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_boxplot() +
  coord_flip()
```



With large data sets like this, the "whiskers" of a boxplot may not provide a meaningful comparison of the different groups. Here are two other options for displaying these data.

```
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_jitter(alpha=.1) +     # randomly jitter the points a bit
  coord_flip()
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_violin() +
  coord_flip()
```





2.8 But wait, there's more

A complete list of geoms and stats can be obtained using `apropos()`.

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

```
##  [1] "geom_abline"     "geom_area"       "geom_bar"        "geom_bin2d"      "geom_blank"
##  [6] "geom_boxplot"    "geom_contour"    "geom_count"      "geom_crossbar"   "geom_curve"
## [11] "geom_density"    "geom_density_2d" "geom_density2d"  "geom_dotplot"    "geom_errorbar"
## [16] "geom_errorbarh"  "geom_freqpoly"   "geom_hex"        "geom_histogram"  "geom_hline"
```

```
## [21] "geom_jitter"     "geom_label"      "geom_line"       "geom_linerange"  "geom_map"
## [26] "geom_path"       "geom_point"      "geom_pointrange" "geom_polygon"    "geom_qq"
## [31] "geom_quantile"   "geom_raster"     "geom_rect"       "geom_ribbon"     "geom_rug"
## [36] "geom_segment"    "geom_smooth"     "geom_spoke"      "geom_step"       "geom_text"
## [41] "geom_tile"       "geom_violin"     "geom_vline"
```

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

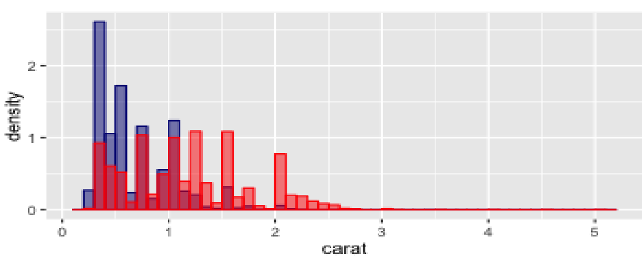
```
##  [1] "stat_bin"          "stat_bin_2d"       "stat_bin_hex"      "stat_bin2d"
##  [5] "stat_binhex"       "stat_boxplot"      "stat_contour"      "stat_count"
##  [9] "stat_density"      "stat_density_2d"   "stat_density2d"    "stat_ecdf"
## [13] "stat_ellipse"      "stat_function"     "stat_identity"     "stat_qq"
## [17] "stat_quantile"     "stat_smooth"       "stat_spoke"        "stat_sum"
## [21] "stat_summary"      "stat_summary_2d"   "stat_summary_bin"  "stat_summary_hex"
## [25] "stat_summary2d"    "stat_unique"       "stat_ydensity"
```

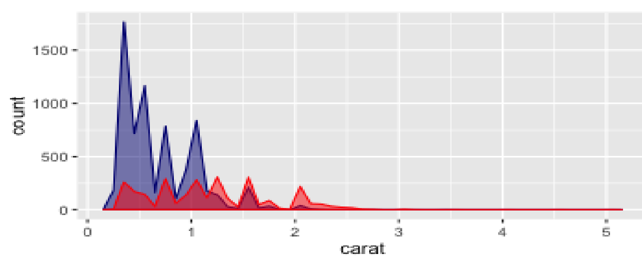
The documentation for these functions can be consulted to learn the options available for each.

2.9 Multiple layers with multiple data sets

Each layer must have data, aesthetics, a geom and a stat, but multiple layers need not share any of these. It is good practice to put into the arguments of `ggplot()` those things we are used by all or most of the layers and leave the rest to be declared in each layer as it is created. In an extreme case, `ggplot()` might not use any arguments.

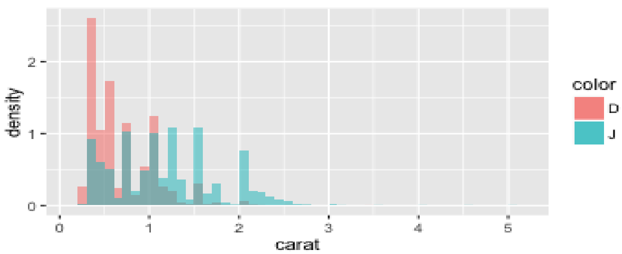
```
require(scales) # to get the alpha function
D <- filter(diamonds, color=="D")
J <- filter(diamonds, color=="J")

ggplot(mapping=aes(x=carat, y=..density..)) +
  geom_histogram(data=D, fill=alpha("navy",0.5), color="navy", binwidth=0.1) +
  geom_histogram(data=J, fill=alpha("red",.5), color="red", binwidth=0.1)
ggplot(mapping=aes(x=carat, ymax=..density..)) +
  stat_bin(data=D, geom="polygon", fill=alpha("navy",0.5), color="navy", binwidth=0.1) +
  stat_bin(data=J, geom="polygon", fill=alpha("red",0.5), color="red", binwidth=0.1)
```

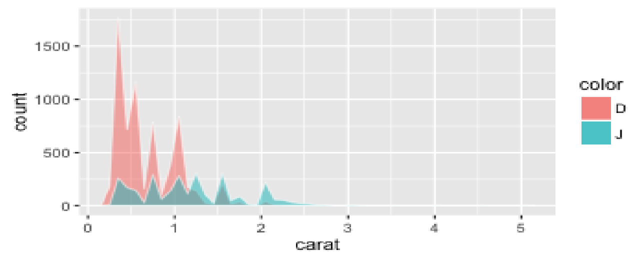




Notice that the plots above do not have a legend for fill. This is because fill is set, not mapped. We could use mapping instead, like this:

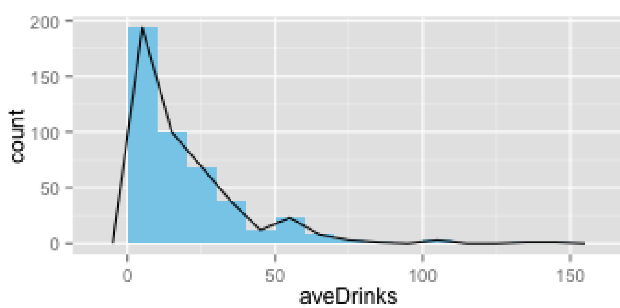
```
ggplot(mapping=aes(x=carat, y=..density..)) +
  geom_histogram(data=D, aes(fill=color), alpha=0.5, binwidth=0.1) +
  geom_histogram(data=J, aes(fill=color), alpha=0.5, binwidth=0.1)
ggplot(mapping=aes(x=carat, ymax=..density..)) +
  stat_bin(data=D, geom="polygon", aes(fill=color), alpha=0.5, binwidth=0.1) +
  stat_bin(data=J, geom="polygon", aes(fill=color), alpha=0.5, binwidth=0.1)
```





An alternative to this approach collects the data on diamonds with colors D and J into single data frame.

```
DorJ <- filter(diamonds, color %in% c("D", "J"))
ggplot( data=DorJ, aes(x = carat, y=..density.., fill = color) ) +
  stat_bin( geom="ribbon", alpha=0.3, binwidth=0.1, color="black")

## Error:  geom_ribbon requires the following missing aesthetics:  ymin, ymax
```



But this does something different. In this plot, the two colors (D and J) are *stacked* rather than overlaid. We'll learn how to avoid this stacking and also how to control the colors selected by the scale in Week 2.

3 Describing Plots with `qplot()`

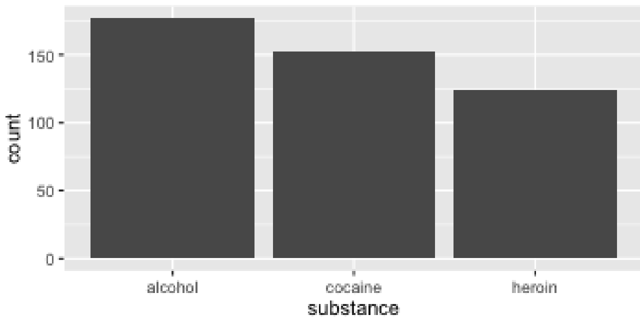
ggplot2 provides a short-cut that makes it easy to describe certain simple plots using the `qplot()` function. In it's simplest form, `qplot()` requires only two or three pieces of information:

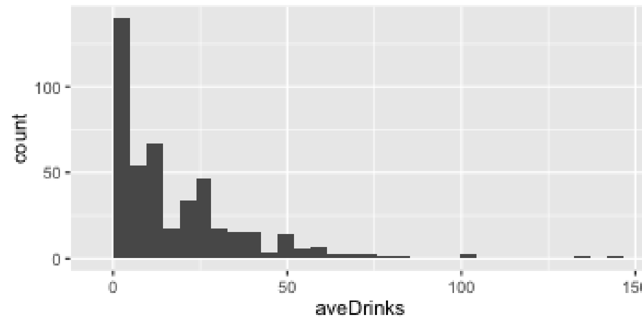
- the name(s) of one or two variable(s) providing the data to be displayed (`x=` and `y=`), and
- the name of a data frame containing those variables (`data=`).

`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).

3.1 One variable plots

```
# one categorical variable -> geom_bar + stat_bin
qplot( substance, data=HELPrct)
# one quantitative variable -> geom_histogram (+ stat_bin)
qplot( aveDrinks, data=HELPrct)

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```

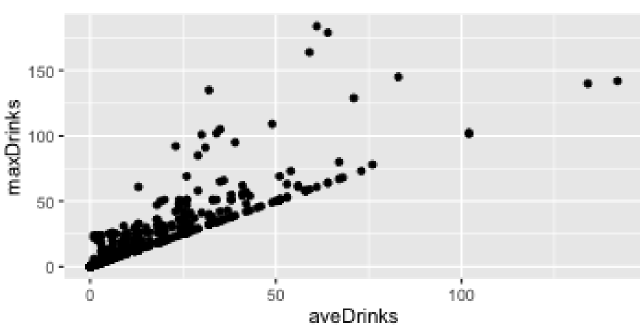


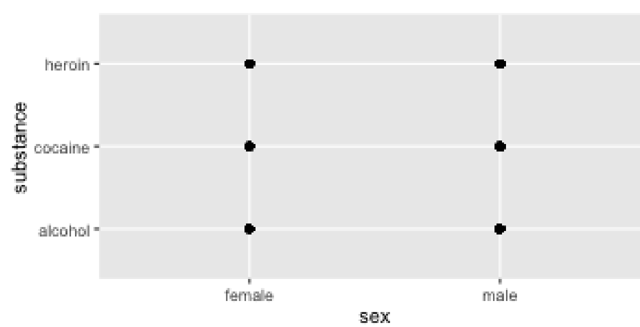


3.2 Two variable plots

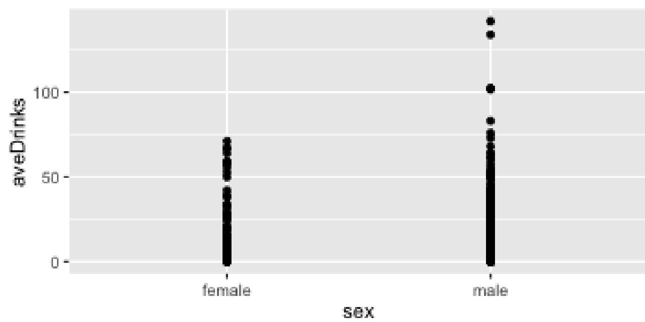
When two variables are provided, the result is a scatter plot. The first variable goes on the horizontal axis and the second on the vertical axis.

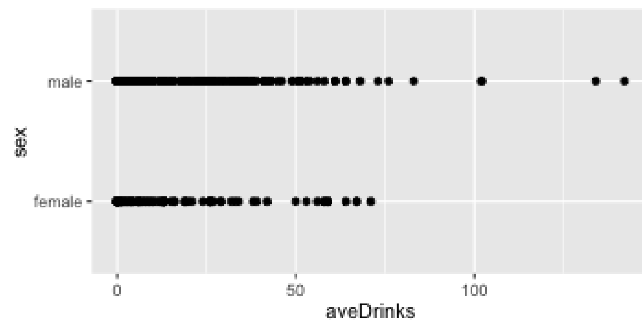
```
# two quantitative variables -> geom_point (+ stat_identity)
qplot( aveDrinks, maxDrinks, data=HELPrct)
# two categorical variables -> geom_point (+ stat_identity)
qplot( sex, substance, data=HELPrct)
```






```
# one categorical and one quantitative variable -> geom_point (+ stat_identity)
qplot( sex, aveDrinks, data=HELPrct)
qplot( aveDrinks, sex, data=HELPrct)
```



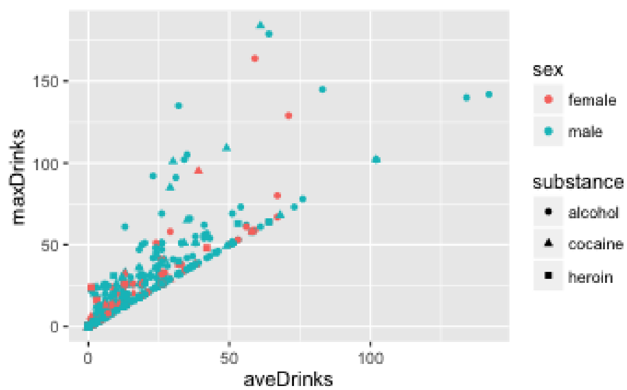


Overplotting of data or labels in these plots might render them less than ideal. This is a common problem in large data sets or when one or more variables are categorical. We'll learn some methods for dealing with this soon.

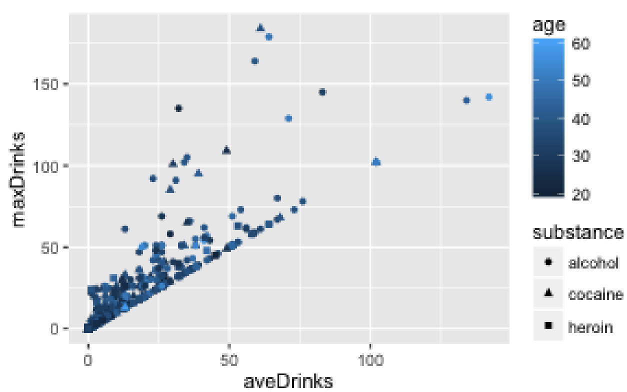
3.3 Mapping aesthetics with `qplot()`

With `qplot()` we can map variables to aesthetics without using the `aes()` wrapper.

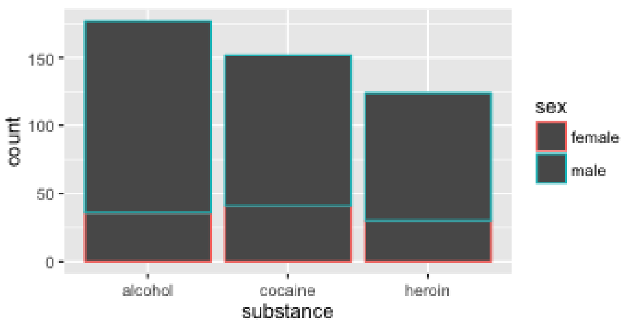
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=sex, shape=substance)
```

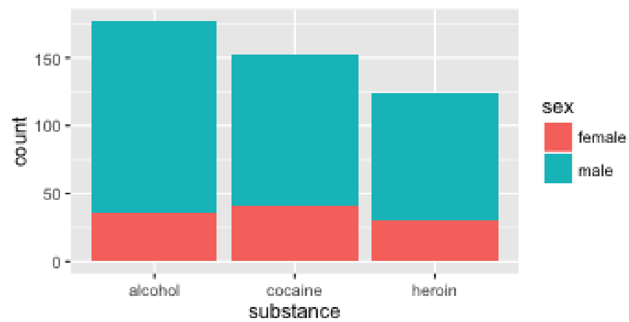



```
qplot( aveDrinks, maxDrinks, data=HELPrct, shape=substance, color=age)
```




```
qplot( substance, data=HELPrct, color=sex )
qplot( substance, data=HELPrct, fill=sex )
```

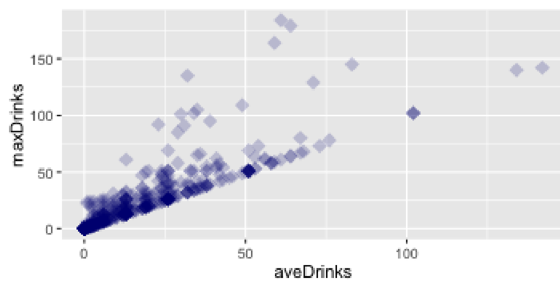




3.4 Setting aesthetics

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

```
qplot( aveDrinks, maxDrinks, data=HELPrct, alpha=I(0.20), size=I(4), shape=I(18),
          color=I("navy") )
```

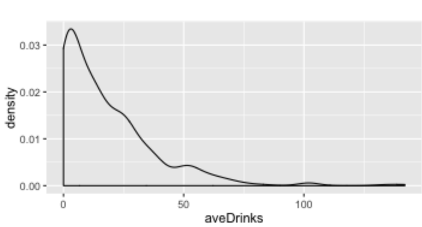


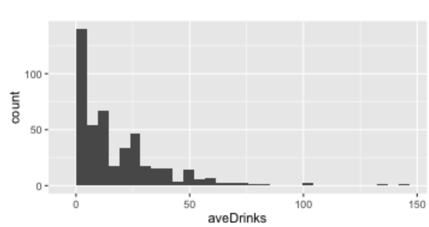
The use of alpha to make the points quite transparent illustrates one way to deal with overplotting. Where there is more data, the points (diamond-shaped in this plot) become darker.

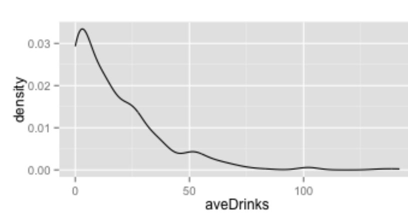
3.5 Choosing non-default geoms and stats

We can also select non-default geoms and stats.

```
qplot( aveDrinks, data=HELPrct, geom="density")
qplot( aveDrinks, data=HELPrct, stat="density")

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.

# We can change the geom to avoid the additional line segments added by the density geom
qplot( aveDrinks, data=HELPrct, stat="density", geom="line")

## Error:  geom_line requires the following missing aesthetics:  y
```





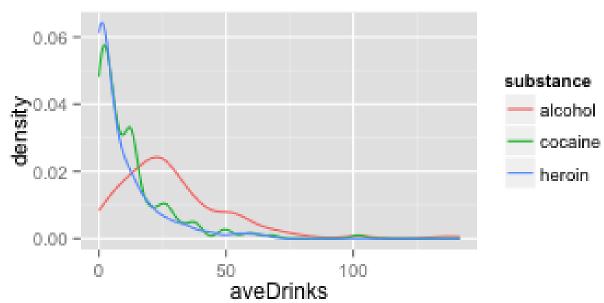


Similarly, we can create frequency polygons by using the `"bin"` stat used to create histograms with the `"polygon"` or `"line"` geoms.

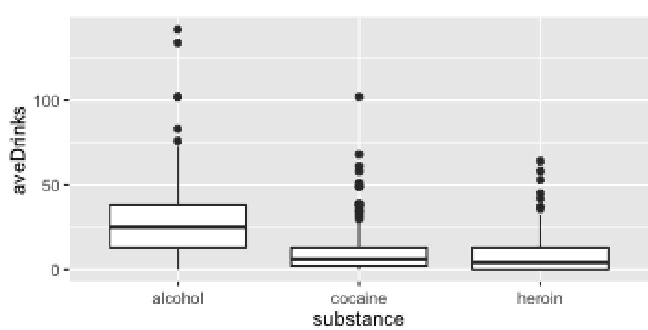
```
qplot( aveDrinks, data=HELPrct, geom="polygon", stat="bin", binwidth=10)

## Warning:  `stat` is deprecated
## Error:  Unknown parameters:  binwidth

qplot( aveDrinks, data=HELPrct, geom="line", stat="bin", binwidth=10)

## Warning:  `stat` is deprecated
## Error:  Unknown parameters:  binwidth
```

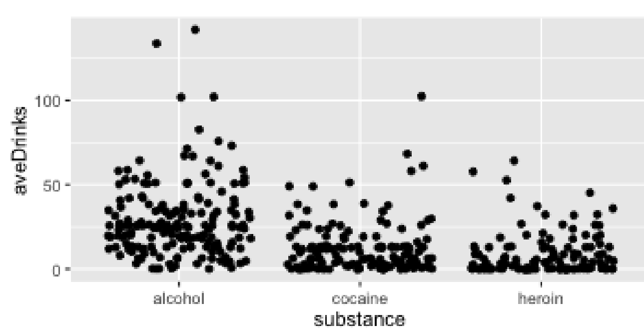
Here are some additional examples.

```
qplot( aveDrinks, data=HELPrct, geom="line", stat="density", color=substance)

## Warning:  `stat` is deprecated
## Error:  geom_line requires the following missing aesthetics:  y

qplot( aveDrinks, data=HELPrct, geom="line", stat="bin", binwidth=10, color=substance)

## Warning:  `stat` is deprecated
## Error:  Unknown parameters:  binwidth
```




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

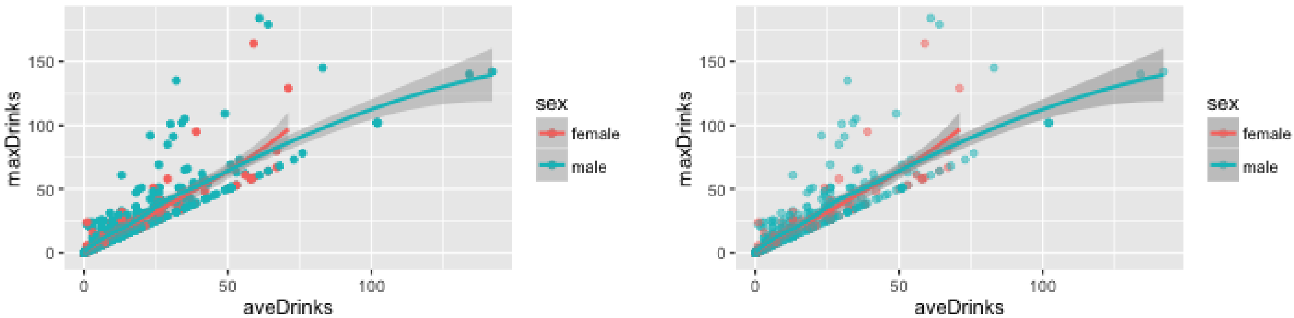




3.6 More than one geom

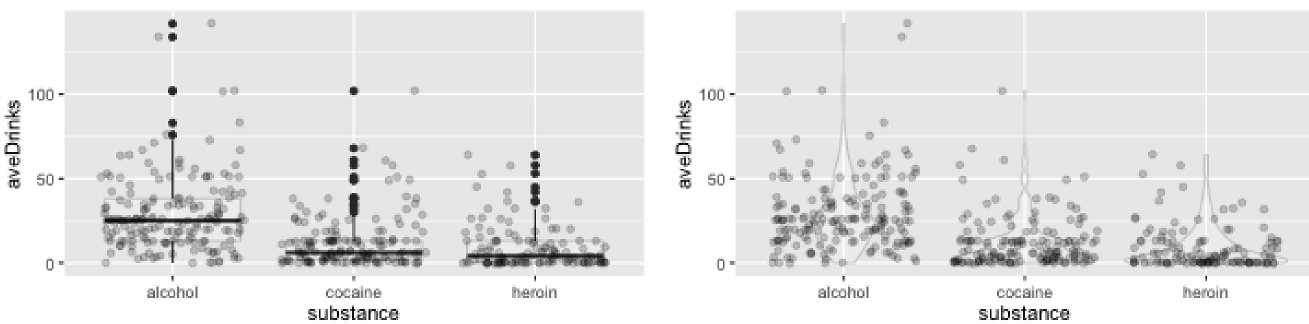
It is possible to specify multiple geoms at once. The smooth geom creates a LOESS or regression smoothing and adds it to the plot.

```
qplot( aveDrinks, maxDrinks, data=HELPrct, geom=c("point","smooth"), color=sex )
qplot( aveDrinks, maxDrinks, data=HELPrct, geom=c("point","smooth"), color=sex, alpha=I(.5) )
```



Many other combinations are possible as well.

```
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))
```

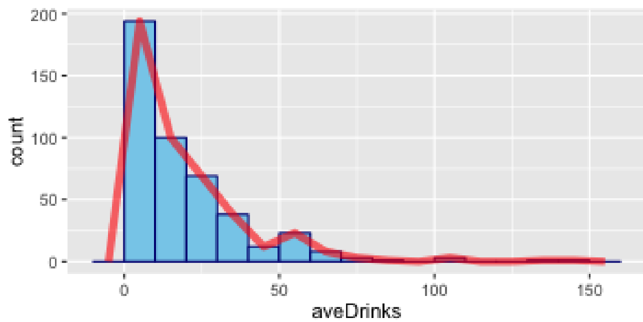


This method works as long as all the geoms involved use the same information. Notice how setting the color below sets the color for both the lines and the bars. Similarly, `binwidth` is shared by both geoms.

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

## Warning:  'stat' is deprecated
## Warning:  'geom_bar()' no longer has a 'binwidth' parameter.  Please use 'geom_histogram()' instead.
## Error:  Unknown parameters:  fill, binwidth

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

## Warning:  'stat' is deprecated
## Warning:  'geom_bar()' no longer has a 'binwidth' parameter.  Please use 'geom_histogram()' instead.
## Error:  Unknown parameters:  fill, binwidth
```

To achieve finer control, we need to work with geoms and stats directly.

```
ggplot( data=HELPrct, aes(x=aveDrinks))  +
  geom_bar(stat="bin", fill="skyblue", color="navy", binwidth=10) +
  geom_line(stat="bin", color="red", size=2, alpha=0.6, binwidth=10)

## Warning:  'geom_bar()' no longer has a 'binwidth' parameter.  Please use 'geom_histogram()' instead.
```



3.7 `qplot()` or `ggplot()`?

It is tempting for beginners to focus their attention on `qplot()` and to ignore `ggplot()`: Many of the most commonly used plots are easily made using `qplot()`, the amount of typing is often slightly less, and one doesn't need to understand the `ggplot2` system as well to get a plot made. The immediate gratification of making beautiful plots without much effort or thought can be intoxicating.

The downside of `qplot()` is that is not as flexible, and eventually `ggplot()` will be required. Learning to use `ggplot()` early will make it clearer why `ggplot2` behaves the way it does and is better preparation for the day when a truly custom plot is required to communicate the story of the data clearly. You are encouraged to use `ggplot()` as much as possible, even early on. If you are able to make a plot with `qplot()` and cannot replicate it with `ggplot()`, treat it as an opportunity to understand more fully how `ggplot2` works.

4 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 occuring
- 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, LOESS)

Exercises

1 Take a look at the data visulizations of the *New York Times* and see if you can describe them using the vocabulary of the grammar of graphics. (You won't be able to recreate them unless you can find the necessary data somewhere.)

2 For each plot in this chapter created using `qplot()`, recreate the plot using `ggplot()`.

3 Read the help page for a geom that you haven't used before and use it to make a plot. Note that each geom lists the aesthetics that it understands (and which are required) as well as the default statistic that it uses.

You might prefer the documentation at http://docs.ggplot2.org/ to the documentation in the package since all the plots from the examples appear in the documentation.

Document Creation

- File creation date: 2016-01-14
- R version 3.2.3 Patched (2015-12-10 r69760)
- `ggplot2` package version: 2.0.0
- `dplyr` package version: 0.4.3.9000
- `mosaicData` package version: 0.13.0

> **dope** , *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent[7]

This week's dope

This week we will learn how to

1. Use **faceting** to create multi-panel plots
2. Use **position** controls (jitter, dodge, and stack)
3. Customize the **scales** used when mapping variables to aesthetics
4. Set the limits of the viewing window for a plot.
5. Modify the labeling of axes and add titles to plots.

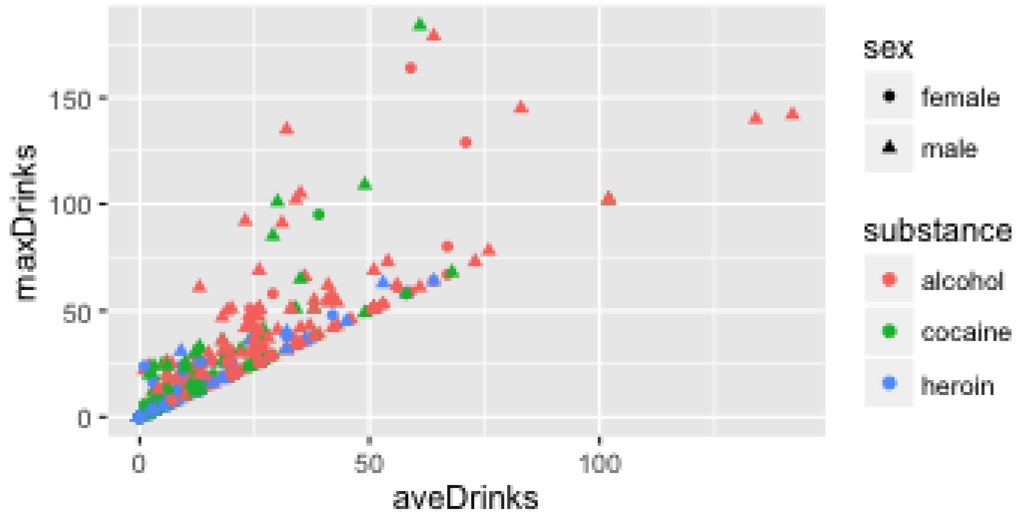
```
Note: The plots in this document are rendered as png files to reduce the overall file size at
```

5 Faceting

Sometimes overplotting can obscure patterns rather than reveal them.

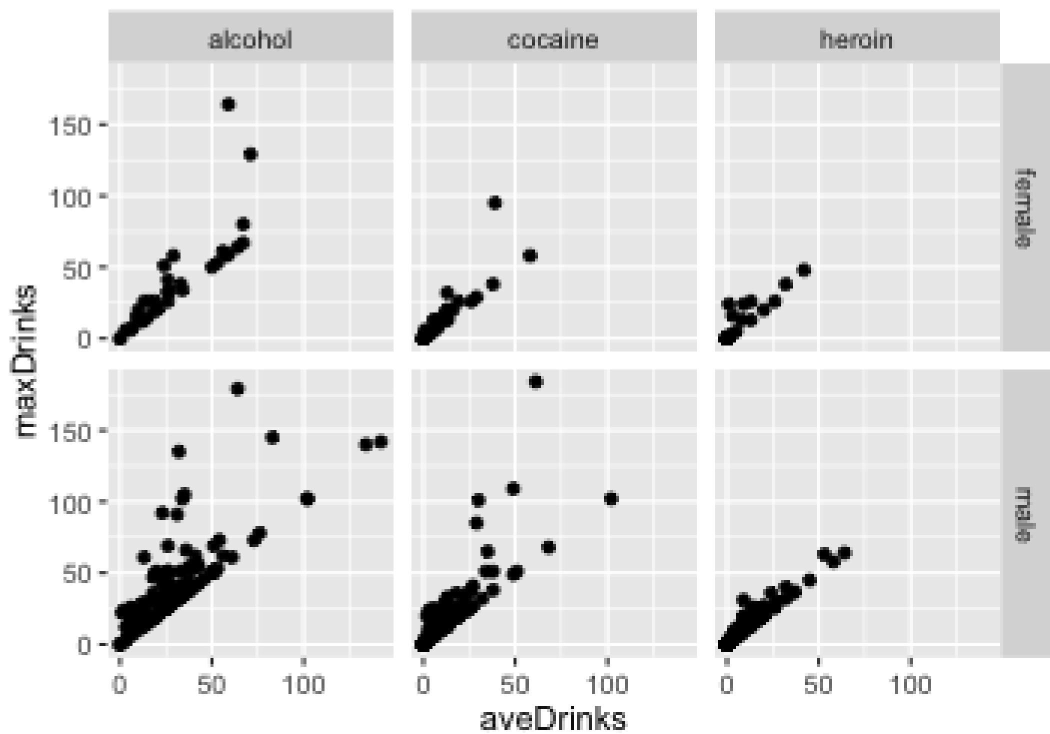
```
HELPrct <- mutate(HELPrct, aveDrinks = i1, maxDrinks = i2)

## Error in eval(expr, envir, enclos):  binding not found:  'i1'

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

[7] definitions selected from Webster's online dictionary



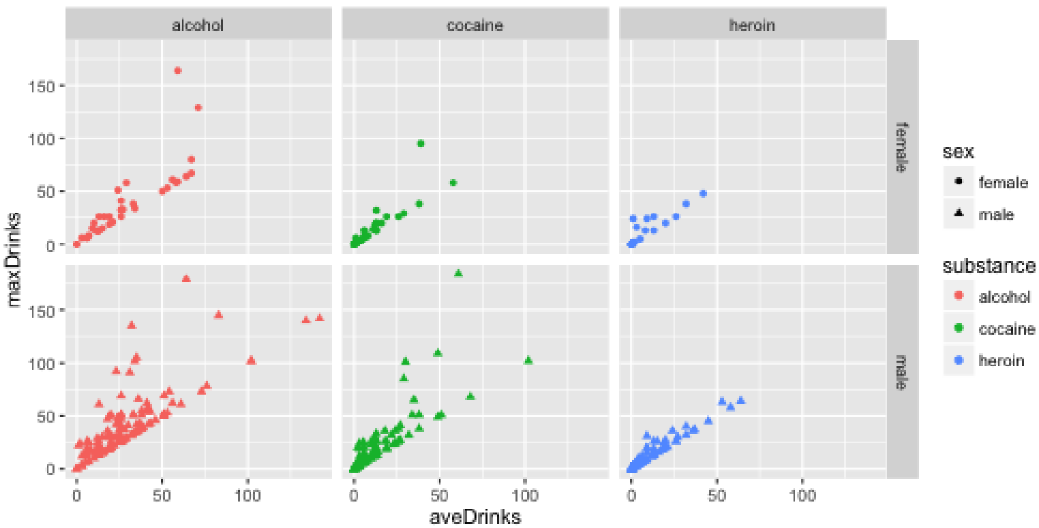
Faceting creates separate sub-plots for each subset. Faceting is described with a formula and works the same way whether we create our plot with `ggplot()` or with `qplot()`.

```
qplot( aveDrinks, maxDrinks, data=HELPrct) +
  facet_grid(sex ~ substance)
```



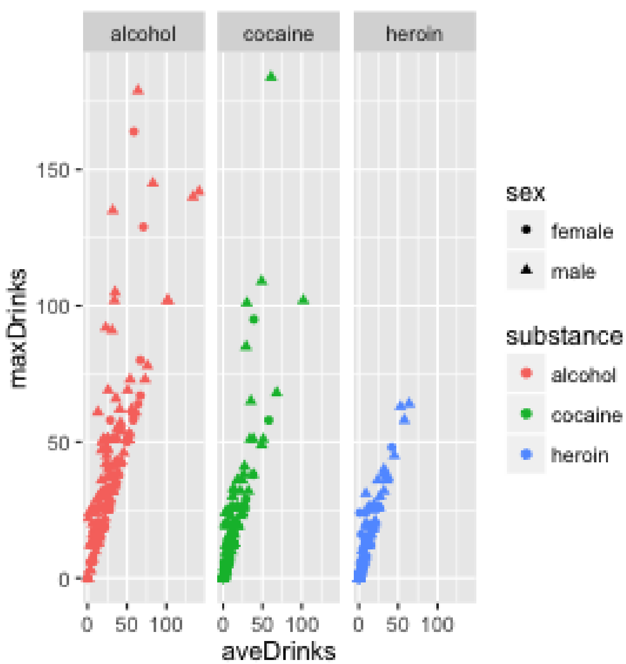
Redundant coding is sometimes useful even when faceting.

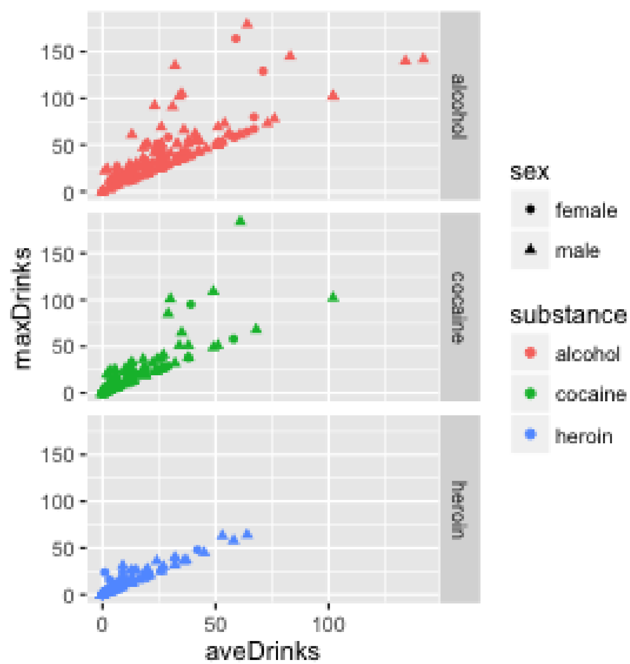
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid(sex ~ substance)
```



We can also use `facet_grid()` with only one variable – just put a "dot" in place of the missing variable:

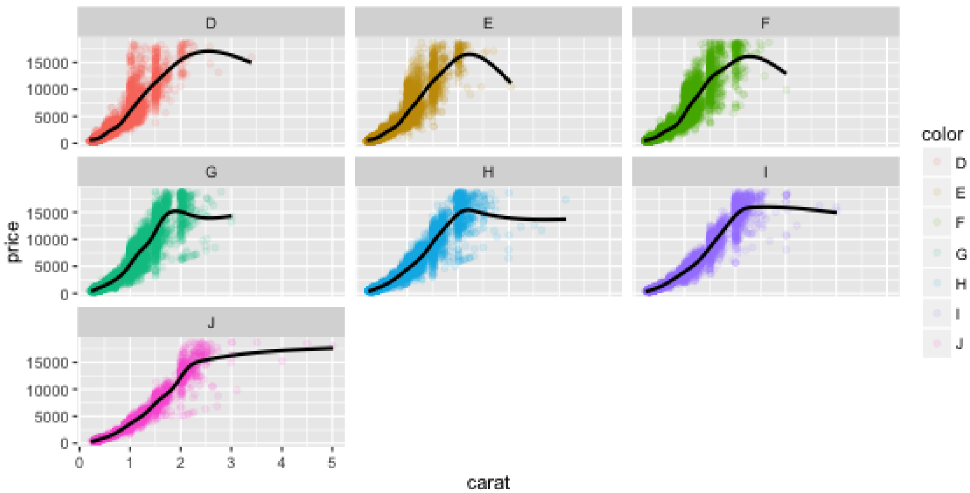
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid(. ~ substance)
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid( substance ~ . )
```





`facet_wrap()` can be used when we have one variable for faceting and that variable has many levels

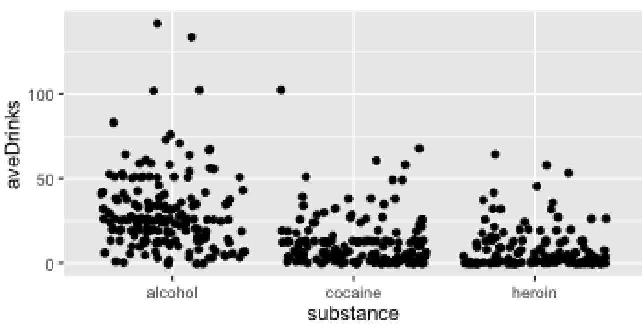
```
ggplot( data=diamonds, aes(x=carat, y=price, color=color) ) +
  geom_point( alpha=0.1 ) +
  geom_smooth( se=FALSE, color="black" ) +
  facet_wrap( ~ color)
```

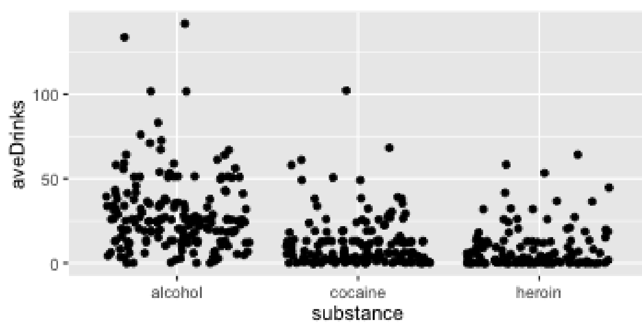


6 Position – jitter, stack, and dodge

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

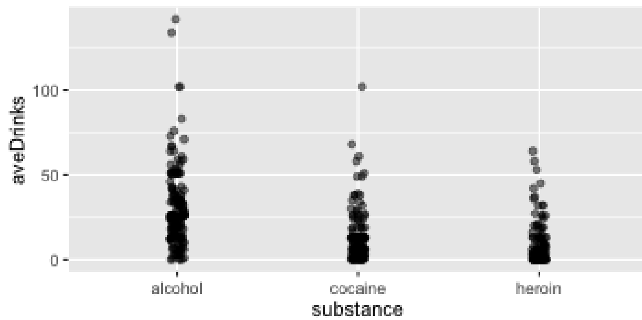
```
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_jitter()
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_point(position="jitter")
```





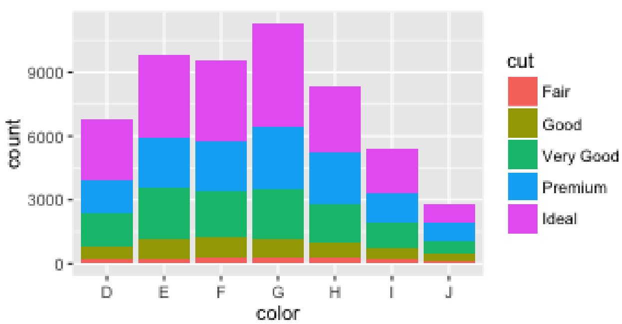
Jittering adds a bit of random noise to the data to help avoid collisions. We can exert more precise control over the jittering if we use `position_jitter()`. Since no information is lost when jittering (slightly) a categorical variable, it is often a good idea to jitter only in one direction when plotting a categorical varaible against a quantitative variable.

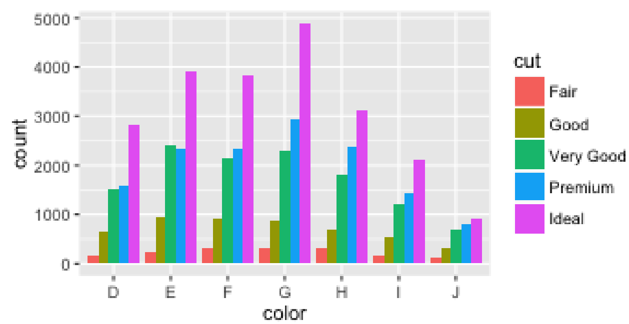
```
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_point(position=position_jitter(width=0.1, height=0), alpha=0.5)
```



Two other position adjustments are stacking and dodging. Stacking is the default for `geom_bar()` and results in segmented bar charts. Dodging is similar to jitter, but for discrete variables.

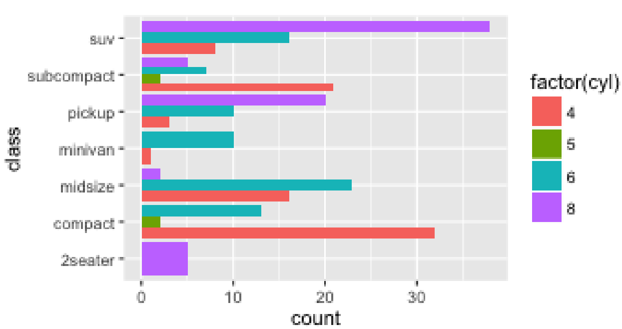
```
ggplot( diamonds, aes(x=color, fill=cut)) +
  geom_bar()
ggplot( diamonds, aes(x=color, fill=cut)) +
  geom_bar(position="dodge")
```

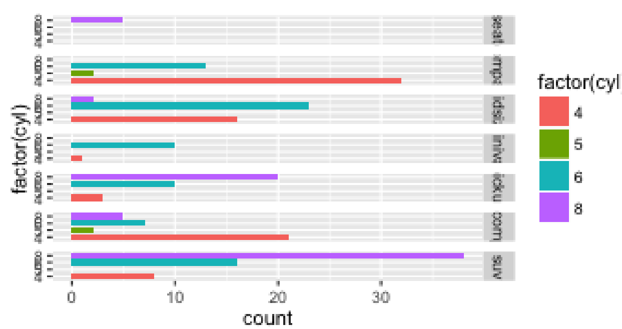




Dodging often yields results that are similar to faceting, but the labeling is different, and the results can be less than pleasing if some of the discrete categories are unpopulated.

```
ggplot(mpg, aes(x=class, fill=factor(cyl))) +
  coord_flip() +
  geom_bar(position="dodge")
ggplot(mpg, aes(x=factor(cyl), fill=factor(cyl))) +
  coord_flip() +
  geom_bar() +
  facet_grid( class ~ . )
```





7 Scales

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).[8]

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

In Week 1 we looked at a schematic of the data flow when creating plots with `ggplot2`, but we said relatively little about scales.

The final data transformation performed by **scales** translates 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.) The `x`- and `y`-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.

7.1 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. Commonly used arguments include

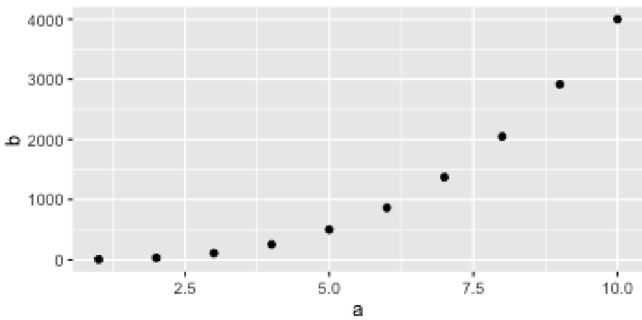
- `trans` a transformation to apply to the variable (e.g., `"log10"`)
- `breaks` manually set the break points for the axes.
- `label` the name of a emphfunction for processing the values appearing on the axes. Use `format=dollar` to format as US currency (`dollar()` is in the `scales` package).
- `name` the name displayed on the guide. This can be useful if the variable name is not appropriate for this purpose.
- `limits` for limiting the data used to create the plot.

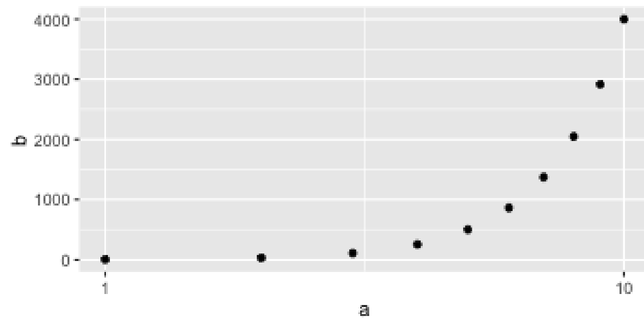
Here are a few examples.

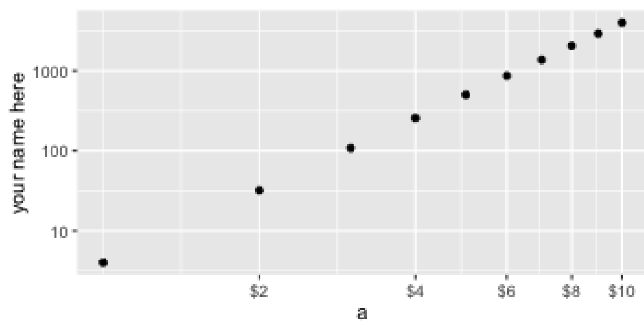
```
a <- 1:10; b <- 4 * a^3
artificial <- data.frame(a=a, b=b)

p <- ggplot(data=artificial, aes(x=a, y=b) ) +
  geom_point()
p
# log scaling on the x axis
p +
  scale_x_continuous(trans="log10")
# Note: dollar is a _function_ in the scales package that formats things as money
```

[8]This 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 `ggplot2` book.

```
require(scales)
p +
  scale_x_continuous(trans="log10", breaks=seq(2,10,by=2), label=dollar) +
  scale_y_continuous(trans="log10", name="your name here")
```

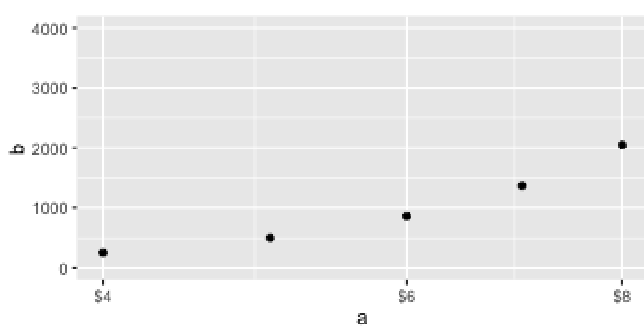






The `limits` argument of a position scale can be used to limit the *data* displayed.

```
p +
  scale_x_continuous(trans="log10", breaks=seq(2,10,by=2), label=dollar, limits=c(4,8))

## Warning:  Removed 5 rows containing missing values (geom_point).
```

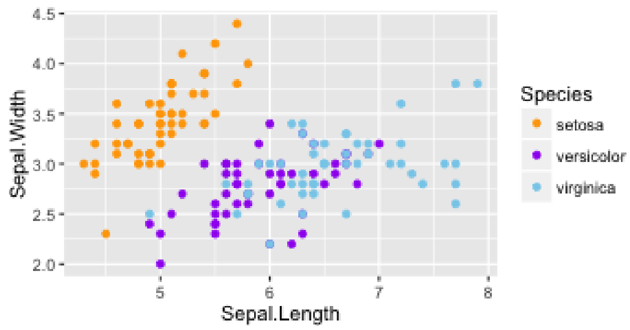


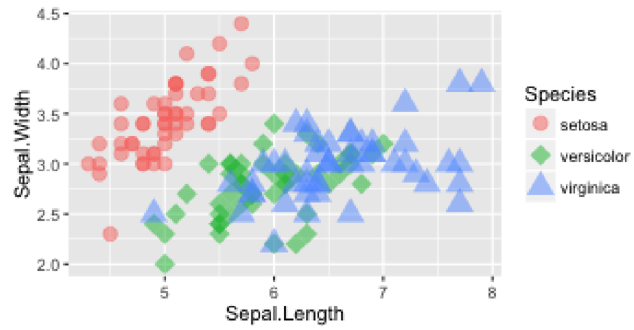
We will see another way to change the viewing window of a plot in Section 8.

7.2 Discrete color and shape scales

Scales are also used to control how aesthetics are mapped to colors and shapes. `scale_color_manual()` can be used to determine exactly which colors are used by a discrete color scale. The values vector used by `scale_color_manual()` and `scale_shape_manual()` may be named or unnamed. If unnamed, the matching is done by order.

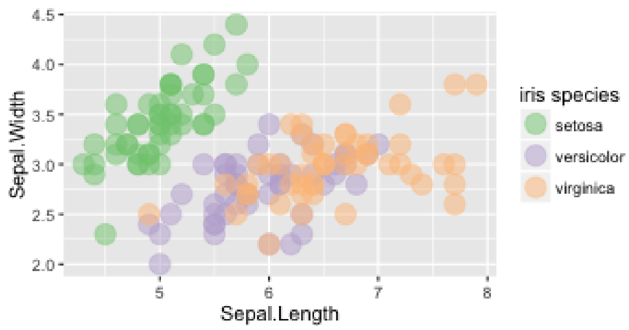
```
p <- ggplot( data=iris, aes(Sepal.Length, Sepal.Width) )
p +
  geom_point(aes(color=Species)) +
  scale_color_manual(
    values=c(setosa="orange", versicolor="purple", virginica="skyblue"))
p +
  geom_point(aes(shape=Species, color=Species), size=5, alpha=.5) +
  scale_shape_manual(values=c(20,18,17))
```





There are also a number of scales that provide various color "palettes".

```
p +
  geom_point(aes(color=Species), size=5, alpha=.5) +
  scale_color_brewer(type="qual", palette=1, name="iris species")
```



For more examples, see the `http://docs.ggplot2.org/`. To find a list of scale functions, use `apropos()`:

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

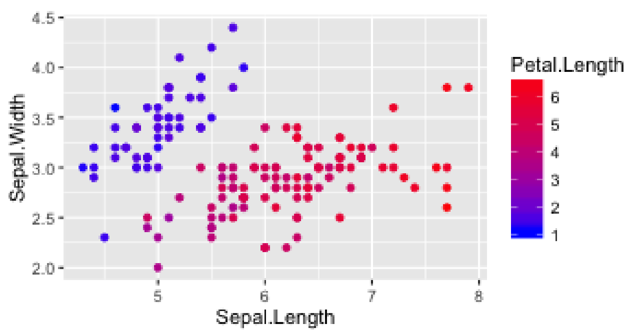
```
##   [1] "scale_alpha"             "scale_alpha_continuous"    "scale_alpha_discrete"
##   [4] "scale_alpha_identity"    "scale_alpha_manual"        "scale_color_brewer"
##   [7] "scale_color_continuous"  "scale_color_discrete"      "scale_color_distiller"
##  [10] "scale_color_gradient"    "scale_color_gradient2"     "scale_color_gradientn"
##  [13] "scale_color_grey"        "scale_color_hue"           "scale_color_identity"
##  [16] "scale_color_manual"      "scale_colour_brewer"       "scale_colour_continuous"
##  [19] "scale_colour_discrete"   "scale_colour_distiller"    "scale_colour_gradient"
##  [22] "scale_colour_gradient2"  "scale_colour_gradientn"    "scale_colour_grey"
##  [25] "scale_colour_hue"        "scale_colour_identity"     "scale_colour_manual"
##  [28] "scale_fill_brewer"       "scale_fill_continuous"     "scale_fill_discrete"
##  [31] "scale_fill_distiller"    "scale_fill_gradient"       "scale_fill_gradient2"
##  [34] "scale_fill_gradientn"    "scale_fill_grey"           "scale_fill_hue"
```

```
## [37] "scale_fill_identity"       "scale_fill_manual"         "scale_linetype"
## [40] "scale_linetype_continuous" "scale_linetype_discrete"   "scale_linetype_identity"
## [43] "scale_linetype_manual"     "scale_radius"              "scale_shape"
## [46] "scale_shape_continuous"    "scale_shape_discrete"      "scale_shape_identity"
## [49] "scale_shape_manual"        "scale_size"                "scale_size_area"
## [52] "scale_size_continuous"     "scale_size_discrete"       "scale_size_identity"
## [55] "scale_size_manual"         "scale_x_continuous"        "scale_x_date"
## [58] "scale_x_datetime"          "scale_x_discrete"          "scale_x_log10"
## [61] "scale_x_reverse"           "scale_x_sqrt"              "scale_y_continuous"
## [64] "scale_y_date"              "scale_y_datetime"          "scale_y_discrete"
## [67] "scale_y_log10"             "scale_y_reverse"           "scale_y_sqrt"
```

7.3 Continuous color scales

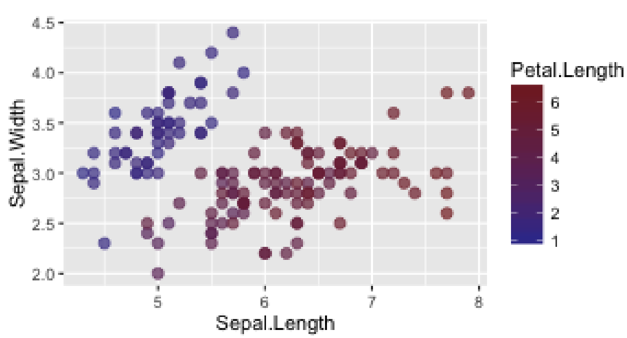
Continuous variables cannot be mapped to shape, but they can be mapped to color, in which case a spectrum of colors is used.

```
ggplot( data=iris, aes(Sepal.Length, Sepal.Width) ) +
  geom_point(aes(color=Petal.Length)) +
  scale_color_continuous(low="blue", high="red")
```



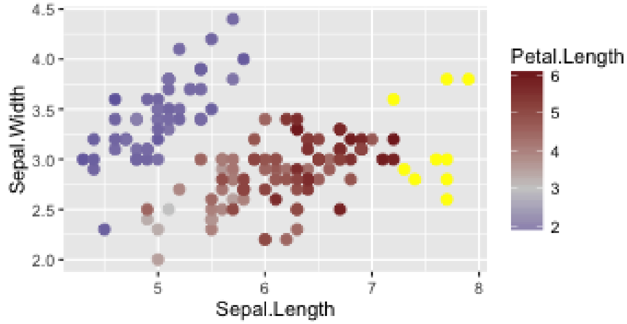
Those colors are bit hard to look at. Let's mute them a bit

```
require(scales)
ggplot( data=iris, aes(Sepal.Length, Sepal.Width) ) +
  geom_point(aes(color=Petal.Length), alpha=0.7, size=2.5) +
  scale_color_continuous(low=muted("blue"), high=muted("red"))
```



Divergent scales move in two directions out from a central value. We can also limit the range of the color scale and map all values outside that range to a color of our choice.

```
ggplot( data=iris, aes(Sepal.Length, Sepal.Width) ) +
  geom_point(aes(color=Petal.Length), size=2.5) +
  scale_color_gradient2(low=muted("blue"), high=muted("red"), mid="gray80",
                        midpoint = 3, limits=c(2,6), na.value="yellow")
```



The use of `limits` can greatly improve the plot if there are small number of outliers with values far from the others.

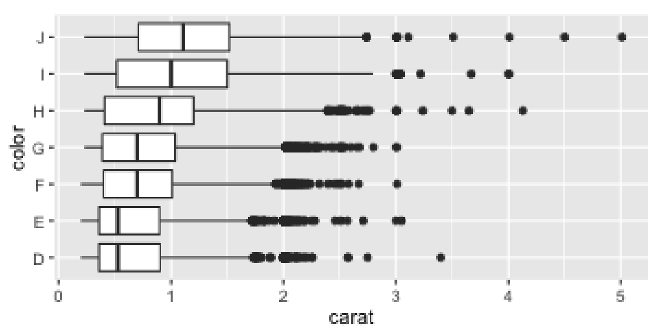
8 Coordinate systems

A coordinate system controls how positions are mapped to the plot. The most common (and default) coordinate system is Cartesian coordinates.

8.1 `coord_flip()`

We have already encountered `coord_flip()`, which reverses the roles of the horizontal and vertical axes. This allows us, for example, to create horizontal boxplots.

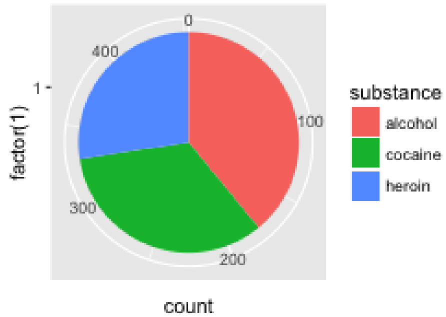
```
ggplot(diamonds, aes(x=color, y=carat)) +
  geom_boxplot() +
  coord_flip()
```



8.2 `coord_polar()`

There are relatively few uses for the polar coordinate scheme, but it does allow us to create pie charts (for which there are also relatively few good uses). A pie chart is simply a stacked bar chart in polar coordinates.

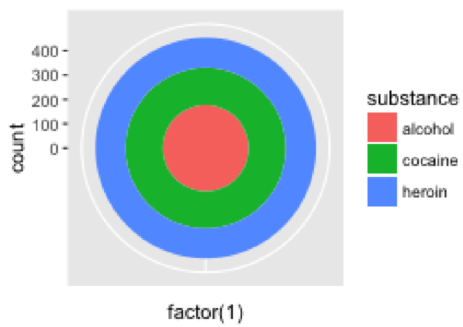
```
ggplot( data=HELPrct, aes(x=factor(1), fill=substance)) +
  geom_bar(width=1) +    # avoid space between "bars"
  coord_polar(theta="y")
```



There is more labeling here than is typical for a pie chart, but since a regular bar chart is easier to read, we won't bother with optimizing this plot.

The default value for `theta` is `"x"`, which produces a "bulls-eye" plot.

```
ggplot( data=HELPrct, aes(x=factor(1), fill=substance)) +
  geom_bar(width=1) +  # avoid space between "bars"
  coord_polar()
```



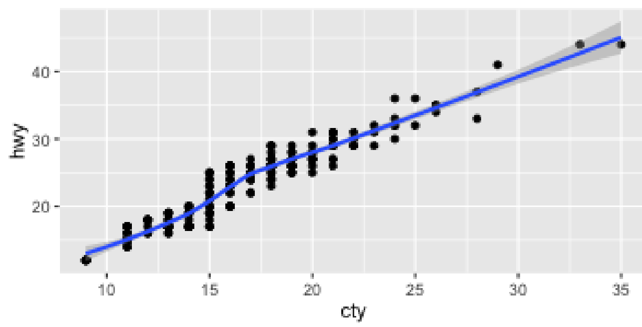
8.3 `coord_map()`

We have not yet discussed maps, which require a projection from a sphere to the plane to render on a flat graphics device. `coord_map()` faciliates selecting one of several such projections when rendering maps.

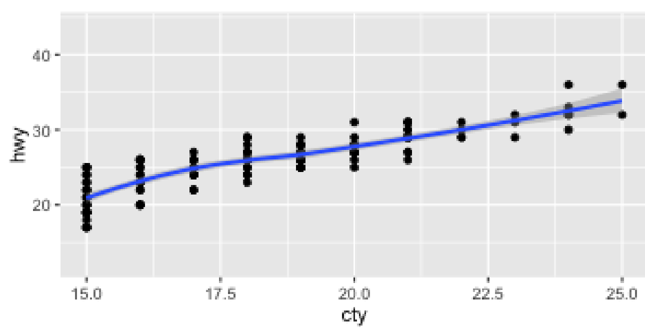
8.4 Using coords to zoom

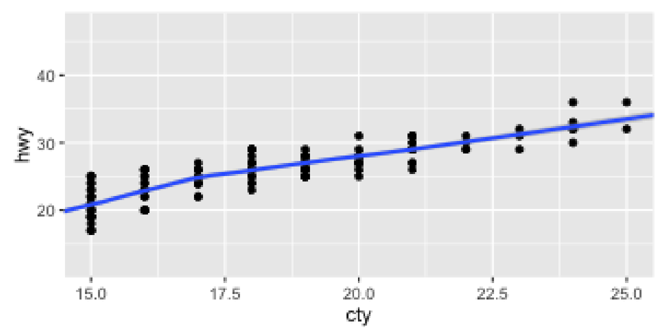
One additional use for a coordinate system is to zoom in on a portion of a plot using `xlim` and `ylim`. Setting limits in a scale removes all data outside those limits from the data frame used to generate the plot; setting the limits in a coord merely restricts the view – all the data are still used.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

```
ggplot(mpg, aes(cty,hwy)) +
  geom_point() + geom_smooth()
```



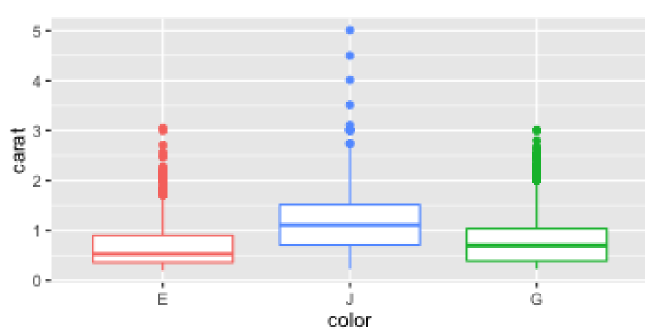

```
ggplot(mpg, aes(cty,hwy)) +
  geom_point() + geom_smooth() + scale_x_continuous(limits=c(15,25))
ggplot(mpg, aes(cty,hwy)) +
  geom_point() + geom_smooth() + coord_cartesian(xlim=c(15,25))
```

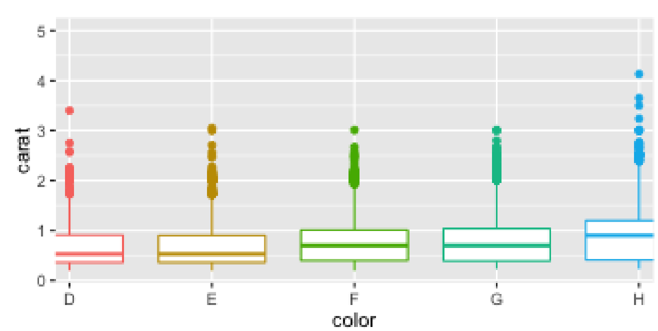




When working with discrete data, the limits are set as numerical values in the coord system but as natural values in the scale (and need not be contiguous or in the original order). There's really no reason for a legend on these plots, so let's turn it off.[9]

```
p <- ggplot(diamonds, aes(x=color, y=carat, color=color)) +
  geom_boxplot()
p +
  scale_x_discrete(limits=c("E","J", "G")) +
  theme(legend.position="none")
p +
  coord_cartesian( xlim=c(1.5,4.5) ) +
  theme(legend.position="none")
```

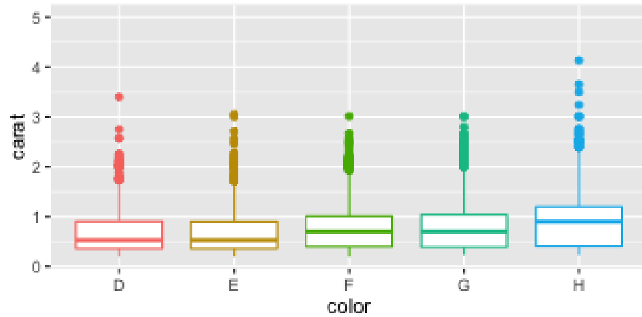




Be sure to leave enough room when setting limits from within a coord as no additional room will be added to accomodate geoms that are not completely contained in the viewing window.

[9]We'll have more to say about `theme()` later. Note that `theme()` replaces `opt()` from older versions of `ggplot2`.

```
p +
  coord_cartesian( xlim=c(1,5) ) +
  theme(legend.position="none")
```

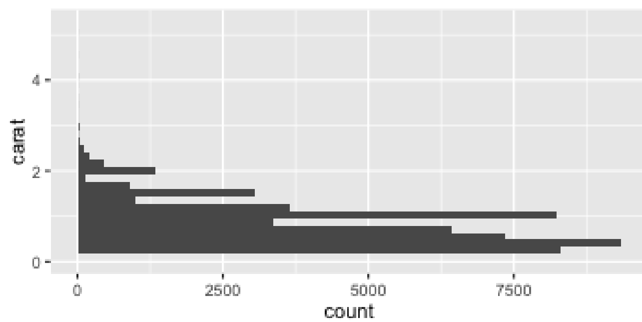


9 Some shortcuts

9.1 qplot()

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

```
qplot( carat, data=diamonds) + coord_flip()
```



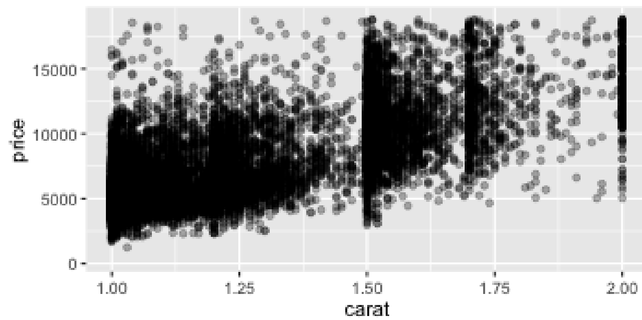
qplot() also provides quick access to some common plot modifications:

- 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.

9.2 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.

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



9.3 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=...).

```
ggplot(diamonds, aes(x=color, y=carat, color=color)) +
  geom_boxplot() +
  labs(title="Now I have a title", x="color of diamond", y="size (carat)")
```

10 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".

10.1 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.

10.2 Using transparency

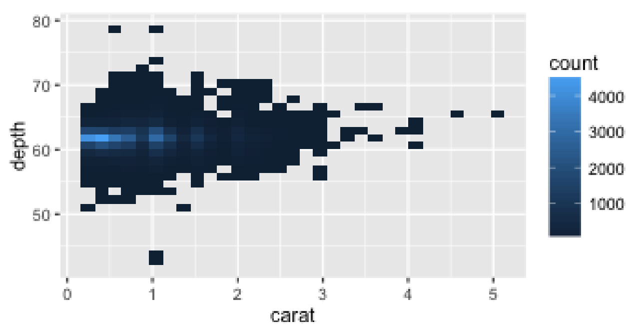
A similarly simple solution is to use transparency so that where there is more data, the plot becomes darker.

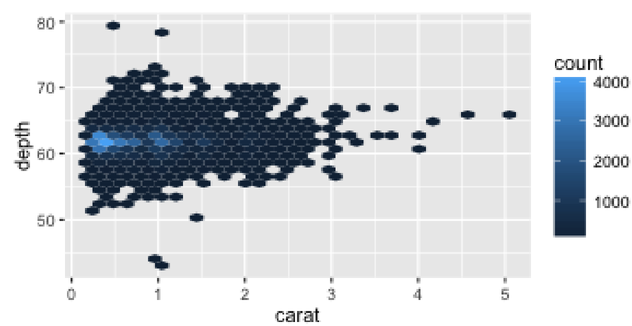
10.3 Use plots that rely on data reduction

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.

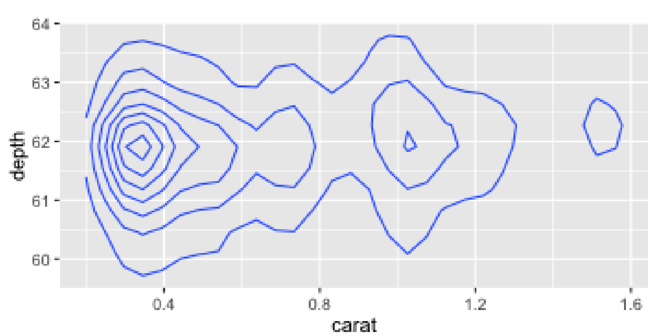
```
p <- ggplot( diamonds, aes(x = carat, y=depth) )
p + stat_bin2d()
require(hexbin)        # You may need to install this package from source

## Loading required package:  hexbin

p + stat_binhex()
```



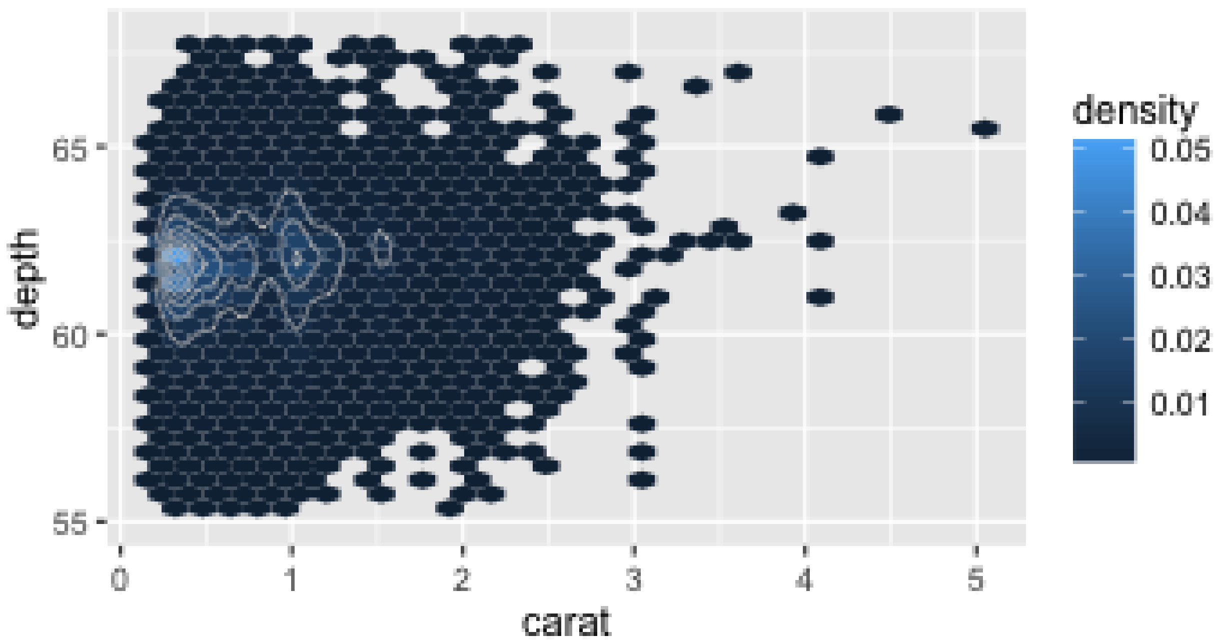


Alternatively, we can plot the contours of a 2d kernel density estimate.

```
p + stat_density2d()
```




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

## Warning:  Removed 110 rows containing non-finite values (stat_binhex).
## Warning:  Removed 110 rows containing non-finite values (stat_density2d).
## Warning:  Removed 11 rows containing missing values (geom_hex).
```



dope , *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent[10]

This week's dope

Often the key to creating a good plot in `ggplot2` 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 `ggplot2`, much of this week's content will be about "data wrangling." In particular, we will learn how to

1. Use the five basic functions in `dplyr` to modify data frames
   (a) `filter()` to select rows from a data frame meeting some criterion
   (b) `select()` to select rows from a data frame meeting some criterion
   (c) `mutate()` to add new variables to a data frame
   (d) `summarise()` to create new data frames summarizing existing data frames.
   (e) `arrange()` to re-order the rows of a data frame.
2. Use `group_by()` and the split-apply-combine methodology to apply a function to many subsets of a data frame.
3. Use `interaction()` to combine two variables into one
4. Use `merge()`, and the various join functions from `dplyr` to combine data from multiple data frames.
5. Use `gather()` and `spread()` from the `tidyr` package to reshape data frames.

[10] definitions selected from Webster's online dictionary

Data Verbs

Most of the functions introduced here have two important properties in common:

1. The first argument is data (typically a `data.frame` or a `tbl_df`).
    - Additional arguments describe what will "happen to" the data.
2. The returned value is data (typically a `data.frame` or a `tbl_df`).

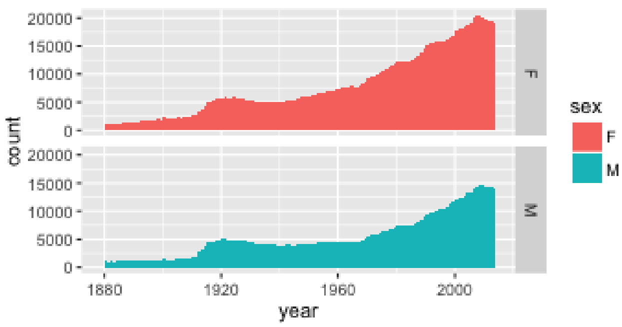
We can think of these as "data verbs" that "do stuff" to data. Once we learn how the data verbs work in isolation, we can chain them together, one after the other, to obtain our desired result.

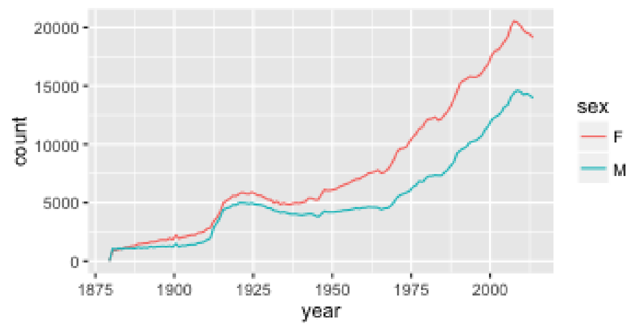
$$\boxed{\text{data}} \xrightarrow{\text{do something}} \boxed{} \xrightarrow{\text{do more}} \boxed{} \xrightarrow{\text{do more}} \cdots$$

Baby Names

The primary data set used in our examples is in the `babynames` R package. The `babynames` data frame has data on all names given to at least 5 children born in the US each year from 1880 through 2013. As we can see, the number of such names is greater for girls than for boys and has been increasing for both girls and boys (although there is an interesting dip for both beginning around 1925).

```
require(babynames)
require(ggplot2)
require(dplyr)
dim(babynames)


## [1] 1792091       5


head(babynames, 3)


## Source: local data frame [3 x 5]
##
##    year   sex  name     n       prop
##   (dbl) (chr) (chr) (int)      (dbl)
## 1  1880     F  Mary  7065 0.07238359
## 2  1880     F  Anna  2604 0.02667896
## 3  1880     F  Emma  2003 0.02052149


ggplot(data=babynames, aes(x=year, fill=sex)) +
  geom_histogram(binwidth=1) + facet_grid( sex ~ . )
ggplot(data=babynames, aes(x=year, color=sex)) +
  geom_line(stat="bin", binwidth=1)
```





11 Subsetting with `filter()`

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 `filter()` function. The general form of a subsetting command is:

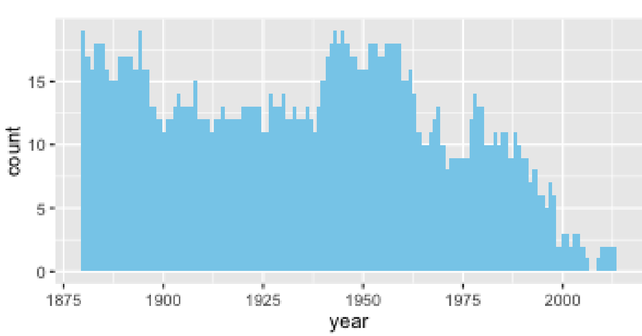
```
newdataframe <- filter( dataframe, condition )
```

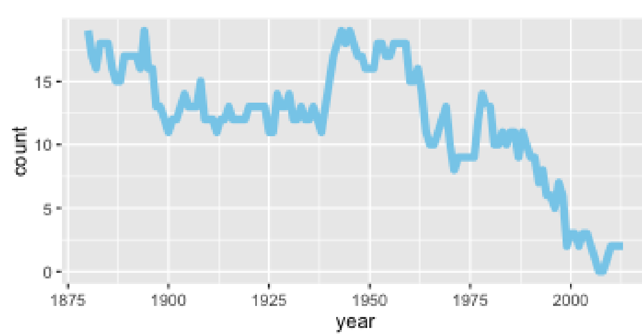
The condition should evaluate to a logical and will often reference variables in the data frame being subsetted. The `$` operator is not required to reference these variables, which is a big savings for complicated subsets.

```
Boys <- filter(babynames, sex=="M")
Girls <- filter(babynames, sex=="F")
```

Here's something a bit more interesting:

```
AtLeast1PercentGirls <- filter(babynames, sex=="F" & prop > 0.01)
ggplot(aes(x=year), data=AtLeast1PercentGirls) +
  geom_histogram(binwidth=1,
      origin=1879.5,    # center bins on integers
      fill="skyblue"
      )
ggplot(aes(x=year), data=AtLeast1PercentGirls) +
  geom_line(binwidth=1,
      origin=1879.5,    # center bins on integers
      color="skyblue", size=2,
      stat="bin"
      )
```





So over past few decades, the number of names given to at least 1% of the girls has been decreasing. In section 19.2 we will see how to plot the proportion of girls who have names shared with at least 1% of the girls born each year.

12 Adding new variables to a data frame with `mutate()`

If we would prefer to work with percentages instead of proportions, we could create a new variable for that.[11]

```
Bnames <- mutate(babynames, percent=100 * prop)
head(Bnames,3)


## Source: local data frame [3 x 6]
##
##    year   sex  name     n       prop  percent
##   (dbl) (chr) (chr) (int)      (dbl)    (dbl)
## 1  1880     F  Mary  7065 0.07238359 7.238359
## 2  1880     F  Anna  2604 0.02667896 2.667896
## 3  1880     F  Emma  2003 0.02052149 2.052149
```

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

```
subword <- function(x, start=1, stop=start) {
  # convert negative numbers to positions relative to end of string
  start <- rep(start, length.out=length(x))
  stop <- rep(stop, length.out=length(x))
  start <- ifelse(start < 0, pmax(nchar(x) + 1 + start), start)
  stop <- ifelse(stop < 0, pmax(nchar(x) + 1 + stop), stop)
  if (any(start > stop)) { warning("Some values of start are greater than values of stop.") }
  tolower(substr(x, start, stop))
}

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

Now let's add some variables to our data frame:[12]

```
Bnames <- mutate( babynames,
  first=subword(name, 1),
  last=subword(name, -1),
  length=nchar(name),
  vowels=vowels(name),
  consonants=length - vowels,   # transform cant do this part
  vowelFrac=vowels/length       # or this part
)
head(Bnames,3)
```

[11] But if the only reason to do so is to get percentages to appear on a plot, there is no reason to do so. Instead, ggplot2 can be instructed to format the scales using percent notation rather than decimal notation – simply pass `labels=percent` to the approprate scale(s). See Section 19.2 for an example.

[12] We could also have used `transform()` to do this. The primary difference between `mutate()` and `tranform()` is that `mutate()` can refer to variables created earlier in the `mutate()` command, but `transform()` cannot. So the example here would require multiple calls to `transform()` or recomputation of the values stored in `length` and `vowels`.

```
## Source: local data frame [3 x 11]
##
##    year   sex  name     n       prop first  last length vowels consonants vowelFrac
##   (dbl) (chr) (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)
## 1  1880     F  Mary  7065 0.07238359     m     y      4      2          2       0.5
## 2  1880     F  Anna  2604 0.02667896     a     a      4      2          2       0.5
## 3  1880     F  Emma  2003 0.02052149     e     a      4      2          2       0.5
```

13 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.

13.1 Vowels and consonants

Here is an example using `interaction()` to add a factor called `vcsplit` that tells us now many vowels and how many consonants are in a name. `drop=TRUE` removes unpopulated levesl from the factor.

```
Bnames <- mutate(Bnames,
  vcsplit=interaction(vowels, consonants, sep=":", drop=TRUE)
)
head(Bnames,3)
```

```
## Source: local data frame [3 x 12]
##
##    year   sex  name     n       prop first  last length vowels consonants vowelFrac vcsplit
##   (dbl) (chr) (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)  (fctr)
## 1  1880     F  Mary  7065 0.07238359     m     y      4      2          2       0.5     2:2
## 2  1880     F  Anna  2604 0.02667896     a     a      4      2          2       0.5     2:2
## 3  1880     F  Emma  2003 0.02052149     e     a      4      2          2       0.5     2:2
```

13.2 `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 <- mutate(mpg,
  drv = paste(drv,"wd", sep=""),
  model2 = interaction(manufacturer, model, trans, fl, drv, sep=" ", drop=TRUE)
)
head(mpg2,3)
```

```
## Source: local data frame [3 x 12]
##
```

```
##   manufacturer model displ  year   cyl      trans   drv   cty   hwy    fl   class
##          (chr) (chr) (dbl) (int) (int)      (chr) (chr) (int) (int) (chr)   (chr)
## 1         audi    a4   1.8  1999     4   auto(l5)   fwd    18    29     p compact
## 2         audi    a4   1.8  1999     4 manual(m5)   fwd    21    29     p compact
## 3         audi    a4   2.0  2008     4 manual(m6)   fwd    20    31     p compact
## Variables not shown: model2 (fctr)
```

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

14 Chaining syntax using %>%

The `magrittr` package (which is automatically loaded when `dplyr` or `mosaic` are loaded) defines a piping operator `%>%` that provides an alternative syntax that is often more covenient, espcially as things become more complicated. The `%>%` operator (which can be read as "then") inserts the result of the left hand side as the first argument to the right hand side. That is, the following two lines of code are equivalent:

```
f(x,y)
x %>% f(y)
```

Here is a more interesting example showing two equivalent data operations.

```
Bnames %>%
  filter(sex=="F" & prop > 0.01) %>%
  head(3)


## Source: local data frame [3 x 12]
##
##    year   sex  name     n       prop first  last length vowels consonants vowelFrac vcsplit
##   (dbl) (chr) (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)  (fctr)
## 1  1880     F  Mary  7065 0.07238359     m     y      4      2          2       0.5     2:2
## 2  1880     F  Anna  2604 0.02667896     a     a      4      2          2       0.5     2:2
## 3  1880     F  Emma  2003 0.02052149     e     a      4      2          2       0.5     2:2


head( filter(Bnames, (sex=="F" & prop > 0.01) ), 3 )
```

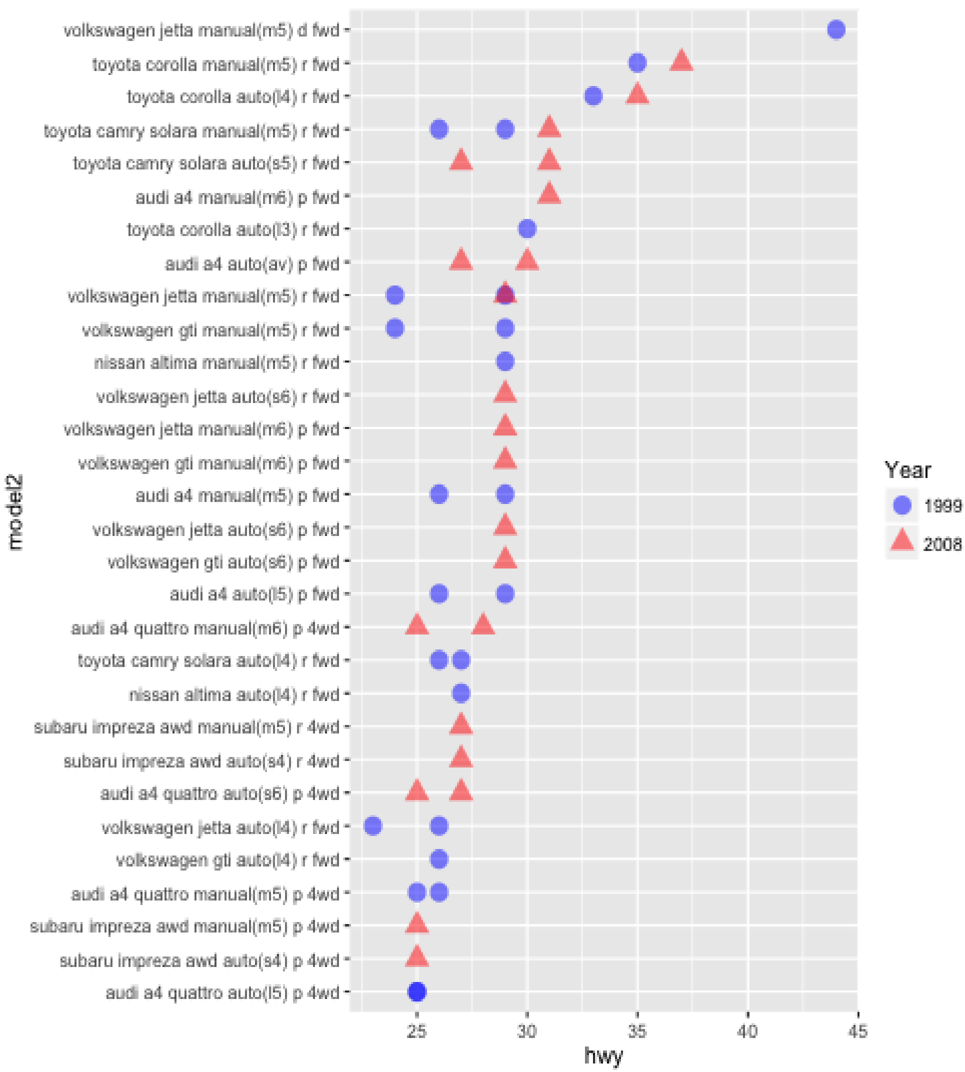


Figure 1: Fuel efficiency for each year and each model of car

```
## Source: local data frame [3 x 12]
##
##    year   sex  name     n       prop first  last length vowels consonants vowelFrac vcsplit
##   (dbl) (chr) (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)  (fctr)
## 1  1880     F  Mary  7065 0.07238359     m     y      4      2          2       0.5     2:2
## 2  1880     F  Anna  2604 0.02667896     a     a      4      2          2       0.5     2:2
## 3  1880     F  Emma  2003 0.02052149     e     a      4      2          2       0.5     2:2
```

Among the advantages of using `%>%` are:

1. Code can be written in the order in which things happen:

   Start with `Bnames`, then filter it to include only girls names that are more popular than 1%, then show me the first 3 lines.

2. Arguments remain near the functions they are arguments to.

3. It is easier to match parentheses.

Each of these advantages will become more important as we chain successively more operations.

15 Removing variables with `select()`

If we want to select only certain columns from our data frame (perhaps to make a data display easier to read or perhaps to reduce the size of our data set by removing unneeded variables from the data frame), we can use `select()`

```
AtLeast1PercentGirls %>% select(year, name, prop) %>% head

## Source: local data frame [6 x 3]
##
##    year      name       prop
##   (dbl)     (chr)      (dbl)
## 1  1880      Mary 0.07238359
## 2  1880      Anna 0.02667896
## 3  1880      Emma 0.02052149
## 4  1880 Elizabeth 0.01986579
## 5  1880    Minnie 0.01788843
## 6  1880  Margaret 0.01616720

# we can also select "negatively"
AtLeast1PercentGirls %>% select( -sex ) %>% head

## Source: local data frame [6 x 4]
##
##    year      name     n       prop
##   (dbl)     (chr) (int)      (dbl)
## 1  1880      Mary  7065 0.07238359
## 2  1880      Anna  2604 0.02667896
## 3  1880      Emma  2003 0.02052149
## 4  1880 Elizabeth  1939 0.01986579
## 5  1880    Minnie  1746 0.01788843
## 6  1880  Margaret  1578 0.01616720
```

16 Summarising a data frame with summarise()

summarise() is analogous to mutate(), but summarise() creates a new 1-row data frame instead of adding variables to an existing data frame. As with mutate(), we can create several variables at once if we like.

```
summarise(Bnames, avg.length = mean(length), max.pop = max(prop) )

## Source: local data frame [1 x 2]
##
##   avg.length    max.pop
##        (dbl)      (dbl)
## 1   6.174078 0.08154561
```

The real power of summarise() comes in combination with group_by(), the topic of our next section.

> Note: dplyr also includes a synonym, summarize(), 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.

17 Split-apply-combine using group_by()

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 subsets; and
3. **Combine** all the results into one data frame.

This is exactly what group_by() facilitates. Let's give it a try:

```
BnameSummariesByYearAndSex <-
Bnames %>%
  group_by(year,sex) %>%      # group by year and sex
  summarise(                  # summmarise each group/sex combination
      max.prop = max(prop),
      avg.length = mean(length),                      # per name
      avg.length2 = sum(length * prop) / sum(prop), # per child
      vowelFrac=mean(vowelFrac ),                     # per name
      vowelFrac2=sum(vowelFrac * prop) / sum(prop)  # per child
  )                           # the summaries are combined automatically
head(BnameSummariesByYearAndSex,3)

## Source: local data frame [3 x 7]
## Groups: year [2]
##
##    year   sex    max.prop avg.length avg.length2 vowelFrac vowelFrac2
##   (dbl) (chr)       (dbl)      (dbl)       (dbl)     (dbl)      (dbl)
## 1  1880     F 0.07238359   5.773885    5.408647 0.4934916  0.4934916
## 2  1880     M 0.08154561   5.634216    5.593460 0.4101043  0.4101043
## 3  1881     F 0.06998999   5.750533    5.398917 0.4955122  0.4955122
```

Notes:

1. The `group_by()` function adds information to the data describing the groups. These groups are then used by downstream applications of functions like `summarise()` and `mutate()`. Sometimes we will want to turn off the grouping, this is done with `ungroup()`. Neither of the options change the data itself, only the grouping information associated with the data.

```
Bnames %>% group_by(year, sex)


## Source: local data frame [1,792,091 x 12]
## Groups: year, sex [268]
##
##     year   sex      name     n       prop first  last length vowels consonants vowelFrac
##    (dbl) (chr)     (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)
## 1   1880     F      Mary  7065 0.07238359     m     y      4      2          2 0.5000000
## 2   1880     F      Anna  2604 0.02667896     a     a      4      2          2 0.5000000
## 3   1880     F      Emma  2003 0.02052149     e     a      4      2          2 0.5000000
## 4   1880     F Elizabeth  1939 0.01986579     e     h      9      4          5 0.4444444
## 5   1880     F    Minnie  1746 0.01788843     m     e      6      3          3 0.5000000
## 6   1880     F  Margaret  1578 0.01616720     m     t      8      3          5 0.3750000
## 7   1880     F       Ida  1472 0.01508119     i     a      3      2          1 0.6666667
## 8   1880     F     Alice  1414 0.01448696     a     e      5      3          2 0.6000000
## 9   1880     F    Bertha  1320 0.01352390     b     a      6      2          4 0.3333333
## 10  1880     F     Sarah  1288 0.01319605     s     h      5      2          3 0.4000000
## ..   ...   ...       ...   ...        ...   ...   ...    ...    ...        ...       ...
## Variables not shown: vcsplit (fctr)


Bnames %>% group_by(year, sex) %>% ungroup()


## Source: local data frame [1,792,091 x 12]
##
##     year   sex      name     n       prop first  last length vowels consonants vowelFrac
##    (dbl) (chr)     (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)
## 1   1880     F      Mary  7065 0.07238359     m     y      4      2          2 0.5000000
## 2   1880     F      Anna  2604 0.02667896     a     a      4      2          2 0.5000000
## 3   1880     F      Emma  2003 0.02052149     e     a      4      2          2 0.5000000
## 4   1880     F Elizabeth  1939 0.01986579     e     h      9      4          5 0.4444444
## 5   1880     F    Minnie  1746 0.01788843     m     e      6      3          3 0.5000000
## 6   1880     F  Margaret  1578 0.01616720     m     t      8      3          5 0.3750000
## 7   1880     F       Ida  1472 0.01508119     i     a      3      2          1 0.6666667
## 8   1880     F     Alice  1414 0.01448696     a     e      5      3          2 0.6000000
## 9   1880     F    Bertha  1320 0.01352390     b     a      6      2          4 0.3333333
## 10  1880     F     Sarah  1288 0.01319605     s     h      5      2          3 0.4000000
## ..   ...   ...       ...   ...        ...   ...   ...    ...    ...        ...       ...
## Variables not shown: vcsplit (fctr)
```

2. 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.

```
summary(Bnames)
```

```
##       year          sex                name                  n                 prop
##  Min.   :1880   Length:1792091     Length:1792091     Min.   :    5.0   Min.   :2.260e-06
##  1st Qu.:1948   Class :character   Class :character   1st Qu.:    7.0   1st Qu.:3.930e-06
##  Median :1981   Mode  :character   Mode  :character   Median :   12.0   Median :7.430e-06
##  Mean   :1972                                         Mean   :  186.1   Mean   :1.422e-04
##  3rd Qu.:2000                                         3rd Qu.:   32.0   3rd Qu.:2.366e-05
##  Max.   :2013                                         Max.   :99674.0   Max.   :8.155e-02
##
##     first               last               length           vowels         consonants
##  Length:1792091     Length:1792091     Min.   : 2.000   Min.   :0.000   Min.   : 0.000
##  Class :character   Class :character   1st Qu.: 5.000   1st Qu.:2.000   1st Qu.: 3.000
##  Mode  :character   Mode  :character   Median : 6.000   Median :3.000   Median : 3.000
##                                        Mean   : 6.174   Mean   :2.802   Mean   : 3.372
##                                        3rd Qu.: 7.000   3rd Qu.:3.000   3rd Qu.: 4.000
##                                        Max.   :15.000   Max.   :8.000   Max.   :11.000
##
##    vowelFrac         vcsplit
##  Min.   :0.0000   3:3    :283068
##  1st Qu.:0.4000   2:3    :221104
##  Median :0.4444   3:4    :211776
##  Mean   :0.4580   2:4    :169867
##  3rd Qu.:0.5000   3:2    :138706
##  Max.   :1.0000   2:2    :119605
##                   (Other):647965
```

```
require(mosaic)  # to get dfapply()
# dfapply() applies a function to selected variables in a data frame.
dfapply(Bnames, FUN=function(x) {sum(is.na(x))}, select=TRUE)
```

```
## $year
## [1] 0
##
## $sex
## [1] 0
##
## $name
## [1] 0
##
## $n
## [1] 0
##
## $prop
## [1] 0
##
## $first
## [1] 0
##
## $last
```

```
## [1] 0
##
## $length
## [1] 0
##
## $vowels
## [1] 0
##
## $consonants
## [1] 0
##
## $vowelFrac
## [1] 0
##
## $vcsplit
## [1] 0
```

3. When computing the averages, we need to think about what we are averaging over. Do we want to average over *names* or over *kids*? If the latter, we need to weight things by how many kids have each name. (In either case, using this data we don't have information about the really rare names.) Notice that we have computed averages two ways above.

17.1 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.

```
Male2000 <- filter(Bnames, year==2000 & sex=="M")
head(Male2000,3)


## Source: local data frame [3 x 12]
##
##    year   sex    name     n       prop first  last length vowels consonants vowelFrac vcsplit
##   (dbl) (chr)   (chr) (int)      (dbl) (chr) (chr)  (int)  (int)      (int)     (dbl)  (fctr)
## 1  2000     M   Jacob 34462 0.01651547     j     b      5      2          3 0.4000000     2:3
## 2  2000     M Michael 32025 0.01534757     m     l      7      3          4 0.4285714     3:4
## 3  2000     M Matthew 28566 0.01368989     m     w      7      2          5 0.2857143     2:5
```

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

Apply

Now we can apply our function to one subset.

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

```
## Source: local data frame [1 x 2]
##
##     max.prop avg.length
##        (dbl)      (dbl)
## 1 0.01651547   6.123359
```

When chained with `group_by()` we can apply this function to each of the subsets in turn.

Combine

In the combine step, the results of applying a function to each of the subset data frames are combined together into one data frame. If you are having trouble getting this process 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.

18 Reordering with `arrange()`

Sometimes we want the rows of our data frame to appear in a particular order. We can order the rows according to any variable in the data frame. (Additional variables can be specified and used to break ties.)

```
t <- table(Bnames$vcsplit)
t[tail(order(t),5)]  # 5 most common vowel/consonant splits
```

```
##
##    3:2    2:4    3:4    2:3    3:3
## 138706 169867 211776 221104 283068
```

```
Bnames %>%
  mutate(vcsplit=interaction(vowels, consonants, sep=":", drop=TRUE)) %>%
  group_by(vcsplit) %>%
  summarise(n=n()) %>%
  arrange( -n ) %>%
  head(5)
```

```
## Source: local data frame [5 x 2]
##
##   vcsplit      n
##    (fctr)  (int)
```

```
## 1     3:3 283068
## 2     2:3 221104
## 3     3:4 211776
## 4     2:4 169867
## 5     3:2 138706
```

The `n()` function in this example is special. It only works in the context of `summarise()` and returns the number of rows in a subsetted data frame.

19 Plots

Once our new data frame has been created using these tools, 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.

```
ggplot(aes(year, avg.length, color=sex), data=BnameSummariesByYearAndSex) +
  geom_point()
ggplot(aes(year, max.prop, color=sex), data=BnameSummariesByYearAndSex) +
  geom_point()
ggplot(aes(year, vowelFrac, color=sex), data=BnameSummariesByYearAndSex) +
  geom_point()
ggplot(aes(year, vowelFrac2, color=sex), data=BnameSummariesByYearAndSex) +
  geom_point()
```

19.1 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.

```
ggplot(aes(colour=sex), data=BnameSummariesByYearAndSex) +   # stuff common to all layers
  geom_point(aes(x=year, y=avg.length),
             data=mutate(BnameSummariesByYearAndSex, plot="ave. length")) +
  geom_point(aes(x=year, y=max.prop),
             data=mutate(BnameSummariesByYearAndSex, plot="max. proprtion")) +
  geom_point(aes(x=year, y=vowelFrac),
             data=mutate(BnameSummariesByYearAndSex, plot="vowel fraction (by name)")) +
  geom_point(aes(x=year, y=vowelFrac2),
             data=mutate(BnameSummariesByYearAndSex, plot="vowel fraction (by kid)")) +
  facet_wrap(~plot, scale="free_y") +
  labs(title="Several plots as facets", y="")
```

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.

19.2 How many girls have popular names?

Now we can return to 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.

In the example below, we include `ggplot()` in our `%>%` chaining and then switch to using `+` to add additional elements to the plot. This works because `data` is the first argument to `ggplot()`.

```
require(scales)    # for percent()
AtLeast1PercentGirls %>%
  group_by(year) %>%
  summarise(popular=sum(prop)) %>%
  ggplot( aes(x=year, y=popular) ) +
```

```
  geom_line() +
  scale_y_continuous(labels=percent)
```

As we see, since about 1950 it has become steadily less popular to give girls "popular" names.

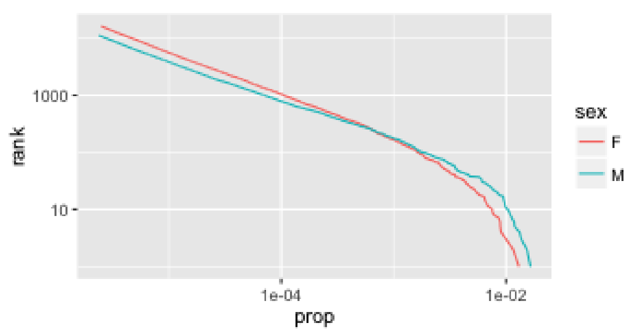
19.3 group_by() and mutate()

We've been using group_by() with summarise(), but it can be used with other functions as well. In particular, we can use it with mutate() 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.

```
Bnames %>%
  group_by(year,sex) %>%
  mutate(rank=rank(-prop)) ->
  BnamesWithRanks
```

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

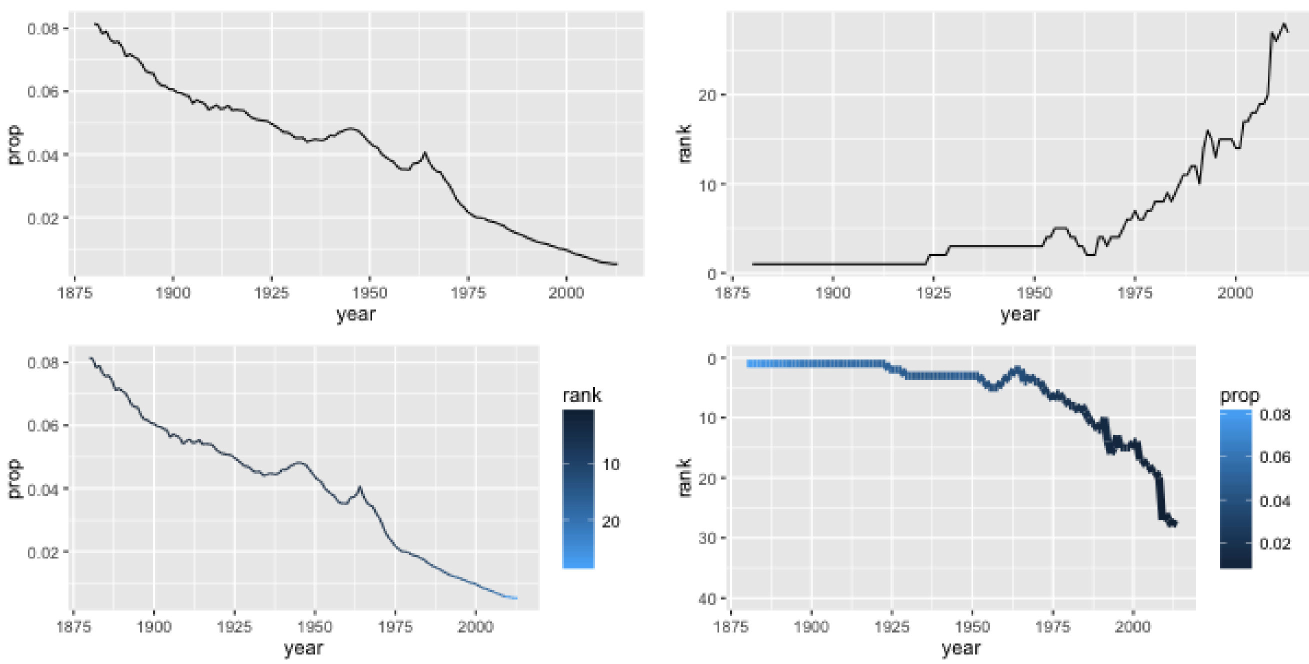
```
# make sure the relationship between prop and rank is monotonic -- at least for one year
qplot(x=prop, y=rank, data=filter(BnamesWithRanks, year==2000), colour=sex,
      geom="line", log="xy" )
```



Now we can see how a name's rank changes over time. Here are several ways to look at rank and/or porportion of Johns over time.

```
BnamesWithRanks %>%
  filter(sex=="M" & name=="John") %>%
  ggplot(aes(x=year, y=prop)) +
  geom_line()

BnamesWithRanks %>%
```

```
  filter(sex=="M" & name=="John") %>%
  ggplot(aes(x=year, y=rank)) +
  geom_line()

BnamesWithRanks %>%
  filter(sex=="M" & name=="John") %>%
  ggplot(aes(x=year, y=prop, colour=rank)) +
  geom_line() +
  guides(colour=guide_colourbar(reverse=TRUE))

BnamesWithRanks %>%
  filter(sex=="M" & name=="John") %>%
  ggplot(aes(x=year, y=rank, colour=prop)) +
  geom_line(size=2) +
  ylim(40,0)
```



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 30.

20 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.

```
require(fastR)

## Loading required package:  fastR
##
## Attaching package:  'fastR'
##
```

```
## The following object is masked from 'package:graphics':
##
##     panel.smooth

head(fusion1,3)
```

```
head(pheno,3)
```

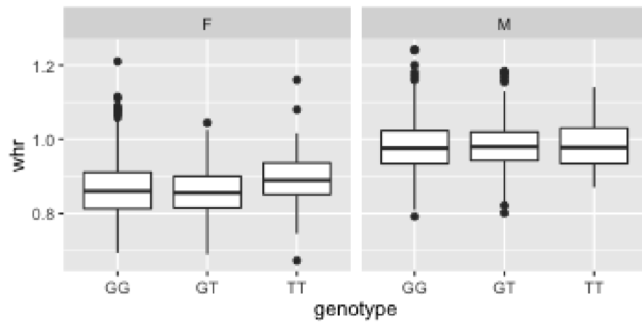
The `id` variable in each data frame contains an identifier that allows us to combine the data sets if we want to compare genotype and phenotype information.

```
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
## 1 1002 RS12255372        1       3       3       GG     0     0     2     0    case 32.85994
## 2 1009 RS12255372        1       3       3       GG     0     0     2     0    case 27.39085
## 3 1012 RS12255372        1       3       3       GG     0     0     2     0 control 30.47048
##   sex      age smoker chol waist weight height       whr sbp dbp
## 1   F 70.76438 former 4.57 112.0   85.6  161.4 0.9867841 135  77
## 2   F 53.91896  never 7.32  93.5   77.4  168.1 0.9396985 158  88
## 3   M 53.86161 former 5.02 104.0   94.6  176.2 0.9327354 143  89
```

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.

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

## Warning:  Removed 79 rows containing non-finite values (stat_boxplot).
```



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.

```
gwa %>%
  group_by(sex) %>%
  summarise(missing.whr=sum(is.na(whr)))

## Source: local data frame [2 x 2]
##
##      sex missing.whr
##   (fctr)       (int)
## 1      F          45
## 2      M          34
```

It is also possible to apply arbitrary functions to each subsetted data frame using `do()`. The only restriction is that the results for each subset must be compatible data frames.

```
gwa %>%
  group_by(sex) %>%
  dplyr::do( favstats(~whr, data=.) )    # . is placehoder for subsetted data frame

## Source: local data frame [2 x 10]
## Groups: sex [2]
##
##      sex       min        Q1    median       Q3      max      mean         sd     n missing
##   (fctr)     (dbl)     (dbl)     (dbl)    (dbl)    (dbl)     (dbl)      (dbl) (int)   (int)
## 1      F 0.6728972 0.8140496 0.8600000 0.910000 1.210000 0.8642272 0.07095933  1061      45
## 2      M 0.7918782 0.9366185 0.9769585 1.020204 1.242424 0.9800165 0.06341068  1191      34
```

In this particular case, `favstats()` would have been capable of producing the same summary in another (and simpler) way:

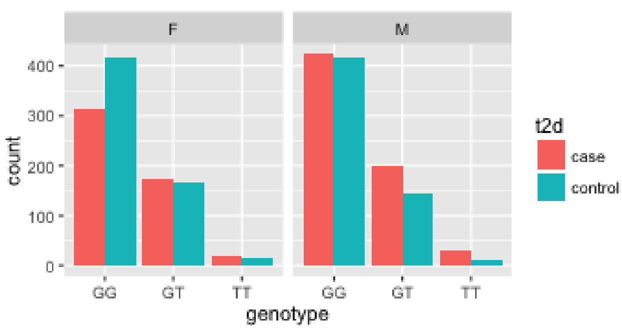
```
favstats( whr ~ sex, data=gwa )

##   sex       min        Q1    median       Q3      max      mean         sd    n missing
## 1   F 0.6728972 0.8140496 0.8600000 0.910000 1.210000 0.8642272 0.07095933 1061      45
## 2   M 0.7918782 0.9366185 0.9769585 1.020204 1.242424 0.9800165 0.06341068 1191      34
```

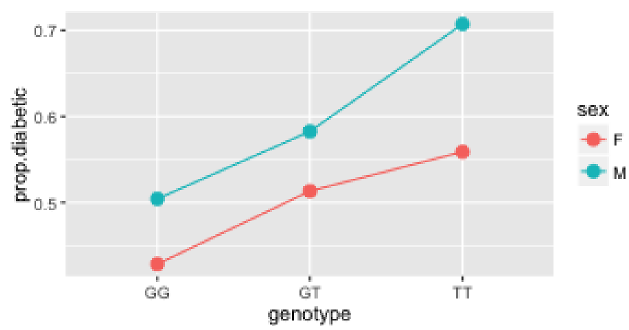
The following table shows that the risk for type 2 diabetes appears to be higher for individuals with more copies of the T allele at this marker:

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

```
## Source: local data frame [6 x 5]
## Groups: sex [?]
##
##      sex genotype diabetic nondiabetic prop.diabetic
##   (fctr)   (fctr)    (int)       (int)         (dbl)
## 1      F       GG      314         419     0.4283765
## 2      F       GT      174         165     0.5132743
## 3      F       TT       19          15     0.5588235
## 4      M       GG      423         416     0.5041716
## 5      M       GT      201         144     0.5826087
## 6      M       TT       29          12     0.7073171
```

```
ggplot(data=gwa, aes(x=genotype, fill=t2d)) +
  geom_bar(position="dodge") + facet_grid(~sex)
ggplot(data=gwa2, aes(x=genotype, y=prop.diabetic, color=sex)) +
  geom_point(size=3) +
  geom_line(aes(group=sex))
```





21 Converting Between Wide and Long Formats

Often it is useful to convert data between "wide" and "long" formats. For example, for names that are given to both sexes, we might prefer to have a wider format with a single line for each year listing both proportions instead of longer format that has a separate row for males and females (for each name and each year). We can use `spread()` to make the format wider and `gather()` to make it longer. Both functions are in the `tidyr` package.

```
require(tidyr)

## Loading required package:  tidyr

Bnames2 <-
  Bnames %>%
  select(year, name, sex, prop) %>%
  group_by(year, name) %>%
  spread(key = sex, value=prop)
head(Bnames2)


## Source: local data frame [6 x 4]
##
##    year   name            F            M
##   (dbl)  (chr)        (dbl)        (dbl)
## 1  1880  Aaron           NA 8.614865e-04
## 2  1880     Ab           NA 4.222973e-05
## 3  1880  Abbie 7.274218e-04           NA
## 4  1880 Abbott           NA 4.222973e-05
## 5  1880   Abby 6.147226e-05           NA
## 6  1880    Abe           NA 4.222973e-04
```

Alternatively, we can gather variables into key-value pairs with `gather()`.

```
Bnames3 <-
  Bnames2 %>%
  gather(key=sex, value=prop, M, F)
head(Bnames3)


## Source: local data frame [6 x 4]
##
##    year   name    sex         prop
##   (dbl)  (chr) (fctr)        (dbl)
## 1  1880  Aaron      M 8.614865e-04
## 2  1880     Ab      M 4.222973e-05
## 3  1880  Abbie      M           NA
## 4  1880 Abbott      M 4.222973e-05
## 5  1880   Abby      M           NA
## 6  1880    Abe      M 4.222973e-04
```

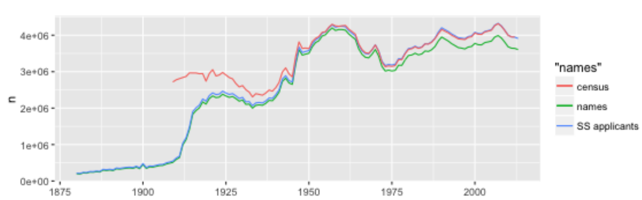
Notice that `Bnames3` is not quite the same as `Bnames`. First, we have dropped some columns to focus our attention. Second, there are now rows for name/year combinations for which there are no children. This is because the NA's created with `spread()` are retained when we `gather()`. This sort of spreading and regathering is sometimes useful when one wants to make missing data explicit rather than implicit.

22 Other Examples

22.1 Under-reporting

The (Bnames) data comes from the US Social Security records. Two other data sets in the babynames package contain data from other sources: applicants contains information on number of social security applicants, and births has the total number of births each year according to the US Census Bureau (rounded to the nearest 1000). It is interesting to compare these different estimates of the number of children born each year.

```
BabiesPerYear <- Bnames %>%
  group_by(year) %>%
  summarise(n=sum(n)) %>%
  merge(births, by="year", all=TRUE) %>%
  merge(applicants %>% group_by(year) %>% summarise(applicants=sum(applicants)),
        by="year", all=TRUE)
BabiesPerYear %>% head


##   year      n births applicants
## 1 1880 201484     NA     216005
## 2 1881 192700     NA     207142
## 3 1882 221537     NA     237730
## 4 1883 216952     NA     232546
## 5 1884 243468     NA     260330
## 6 1885 240856     NA     257898


ggplot(aes(x=year), data=BabiesPerYear) +
  geom_line(aes(y=n, color="names")) +
  geom_line(aes(y=applicants, color="SS applicants")) +
  geom_line(aes(y=births, color="census"))

## Warning:  Removed 30 rows containing missing values (geom_path).
```



We can see that early on it appears that many children were not registered with the SSA. After about 1950, the census numbers and SSA numbers track together. Prior to that, it appears that quite a few people didn't obtain a social security number. More recently the gap between the number of babies in names and applicants has been growing – presumably because a larger fraction of children have names given to fewer than 5 children the year they were born.

22.2 Issues with names

One difficulty in analysing baby names is that there are many spellings for some names and many names that are quite similar.

```
Bnames %>%
  filter( subword(name,1,3) == "ann" ) %>%
  group_by(name,sex) %>%
  summarise(n = sum(n)) %>%
  arrange(-n)


## Source: local data frame [312 x 3]
## Groups: name [303]
##
##          name   sex      n
##         (chr) (chr)  (int)
## 1         Ann     F 467213
## 2         Ann     M   1365
## 3        Anna     F 868027
## 4        Anna     M   2722
## 5  Annaalicia     F     25
## 6     Annabel     F  11694
## 7    Annabela     F      6
## 8    Annabele     F      6
## 9    Annabell     F   5798
## 10  Annabella     F   9157
## ..        ...   ...    ...
```

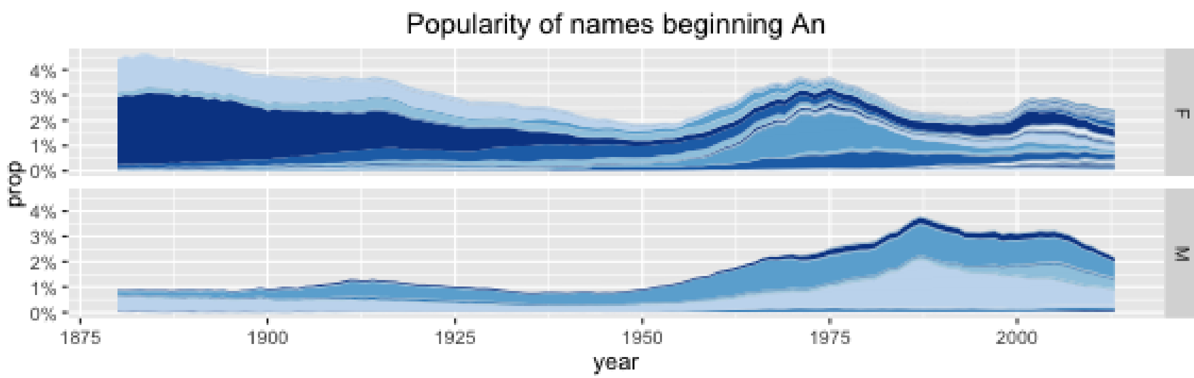
One solution to this is to group names, for example by some prefix. Here we create a function that shows the popularity of all names with a specified prefix.[13] Creating functions that make it easy to create many similar plots is a good illustration of the DRY pinciple (Don't Repeat Yourself) and is 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.

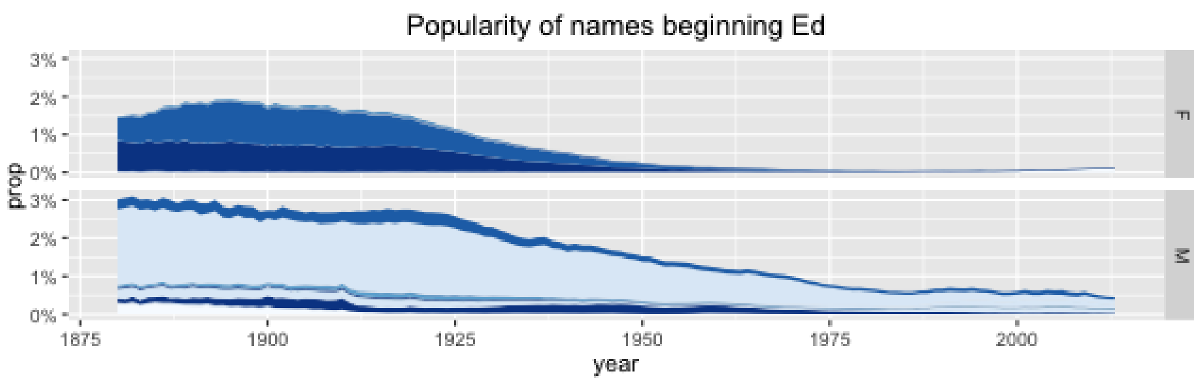
```
beginningNamesPlot <- function( data, prefix, legend=FALSE ) {
  Names <- data %>%
    filter( subword(name, 1, nchar(prefix) ) == tolower(prefix) ) %>%
    group_by(year, name, sex) %>%
    summarise( prop=sum(prop) )

  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),ceiling(numNames/8))) +
  facet_grid(sex~.) +
  labs(title=paste("Popularity of names beginning", prefix)) +
  guides(fill=legend)
}
```

[13]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 now available and makes creating interactive plots much easier for people who know ggplot2 since it is being designed and written by Hadley Wickham using basically the same grammar of graphics approach (but with syntax more like dplyr). 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.

```
beginningNamesPlot(Bnames, "An")

## Warning:  ‘stat‘ is deprecated

beginningNamesPlot(Bnames, "Ed")

## Warning:  ‘stat‘ is deprecated
```





22.3 A ggvis example

Here is an example of a plot using ggvis to provide tooltips. More work is required to compute the stacking manually, faceting is not supported yet, and the syntax is a bit different, but otherwise, the basic approach is the same as was used with ggplot2. This plot cannot be displayed in a PDF document, but it should be executable in an RStudio console.

```
require(ggvis)

display_name <- function(x) {
    if(is.null(x)) return(NULL)
    # paste0(names(x), ": ", format(x, digits=4), collapse = "<br />")
    paste0("<i>", x$name, "</i>")
}

beginningNamesPlot2 <- function( data, prefix, sexes="M", legend=FALSE ) {
  Names <- data %>%
    filter(
```

```
      subword(name, 1, nchar(prefix) ) == tolower(prefix) &
              sex %in% sexes ) %>%
    group_by(year, name, sex) %>%
    summarise( prop=sum(prop) )

  numNames <- length(unique(Names$name))

  Names %>%
    group_by(year) %>%
    arrange(name) %>%
    mutate(to = cumsum(prop), from = c(0, to[-n()])) %>%
    ggvis(x = ~year, y = ~from, y2 = ~to, fill=~name) %>%
    group_by(name) %>%
    layer_ribbons() %>%
    hide_legend("fill") %>%
    add_tooltip(display_name)

}
beginningNamesPlot2(Bnames, "Ann", "F")
```

dope , *n.* information especially from a reliable source [the inside dope]; *v.* figure out – usually used with out; *adj.* excellent[14]

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.

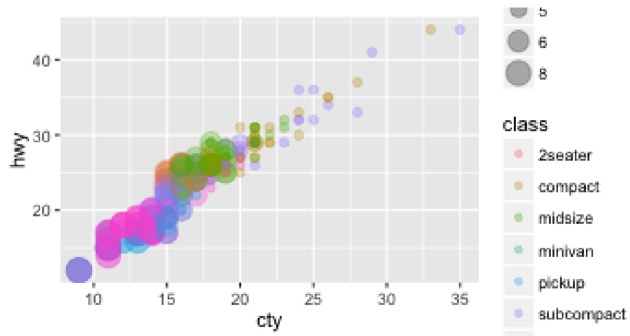
This Dope Sheet includes terse descriptions and examples of the main things covered this week. See the other course materials and the `ggplot2` book for more complete descriptions and additional examples. But note that there have been a number of changes to `ggplot2` since the book was published that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

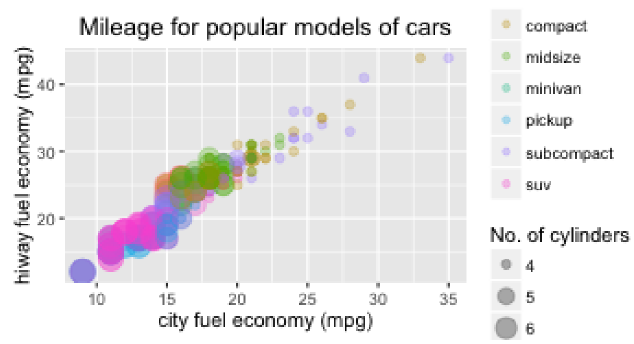
23 Modifying guides using `labs()`

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()`.

[14] definitions selected from Webster's online dictionary

```
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
```





24 Selecting a theme

Themes affect the way various plot elements look. Using themese to control these elements makes it easier to acheive 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.

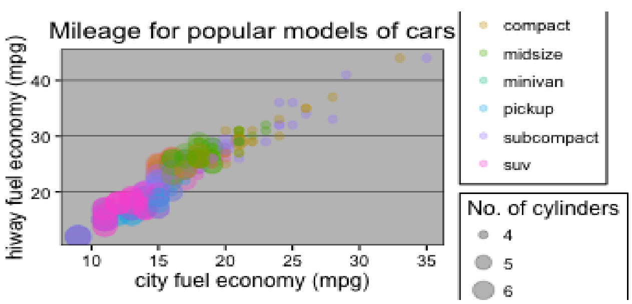
A theme is really just a list of settings.

```
theme_gray

## function (base_size = 11, base_family = "")
## {
##     half_line <- base_size/2
##     theme(line = element_line(colour = "black", size = 0.5, linetype = 1,
##         lineend = "butt"), rect = element_rect(fill = "white",
##         colour = "black", size = 0.5, linetype = 1), text = element_text(family = base_family,
##         face = "plain", colour = "black", size = base_size, lineheight = 0.9,
##         hjust = 0.5, vjust = 0.5, angle = 0, margin = margin(),
##         debug = FALSE), axis.line = element_blank(), axis.text = element_text(size = rel(0.8),
##         colour = "grey30"), axis.text.x = element_text(margin = margin(t = 0.8 *
##         half_line/2), vjust = 1), axis.text.y = element_text(margin = margin(r = 0.8 *
##         half_line/2), hjust = 1), axis.ticks = element_line(colour = "grey20"),
##         axis.ticks.length = unit(half_line/2, "pt"), axis.title.x = element_text(margin = margin(t = 0.8 *
##             half_line, b = 0.8 * half_line/2)), axis.title.y = element_text(angle = 90,
##             margin = margin(r = 0.8 * half_line, l = 0.8 * half_line/2),
##             ), 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(hjust = 0),
##         legend.title.align = NULL, legend.position = "right",
##         legend.direction = NULL, legend.justification = "center",
##         legend.box = NULL, panel.background = element_rect(fill = "grey92",
```

```
##             colour = NA), panel.border = element_blank(), panel.grid.major = element_line(colour = "white")
##         panel.grid.minor = element_line(colour = "white", size = 0.25),
##         panel.margin = unit(half_line, "pt"), panel.margin.x = NULL,
##         panel.margin.y = NULL, panel.ontop = FALSE, strip.background = element_rect(fill = "grey85",
##             colour = NA), strip.text = element_text(colour = "grey10",
##             size = rel(0.8)), strip.text.x = element_text(margin = margin(t = half_line,
##             b = half_line)), strip.text.y = element_text(angle = -90,
##             margin = margin(l = half_line, r = half_line)), strip.switch.pad.grid = unit(0.1,
##             "cm"), strip.switch.pad.wrap = unit(0.1, "cm"), plot.background = element_rect(colour = "white"
##         plot.title = element_text(size = rel(1.2), margin = margin(b = half_line *
##             1.2)), plot.margin = margin(half_line, half_line,
##             half_line, half_line), 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.)

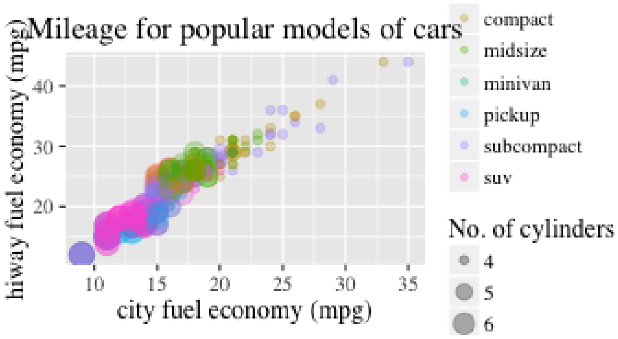
```
require(ggthemes)

## Loading required package:  ggthemes
## Warning:  replacing previous import by 'grid::arrow' when loading 'ggthemes'
## Warning:  replacing previous import by 'grid::unit' when loading 'ggthemes'
## Warning:  replacing previous import by 'scales::alpha' when loading 'ggthemes'
##
## Attaching package:  'ggthemes'
##
## The following object is masked from 'package:mosaic':
##
##     theme_map

q + theme_excel()
```

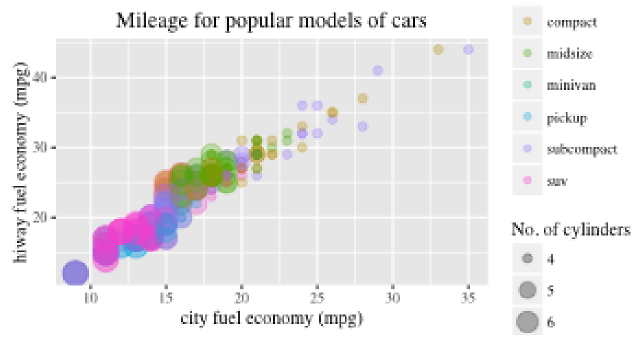



```
q + theme_economist()

## Warning:  'axis.ticks.margin' is deprecated.  Please set 'margin' property  of 'axis.text' instead
## Error in FUN(X[[i]], ...):  Theme element 'text' has NULL property:  margin, debug
```

The themes in ggplot2 take two arguments that control the size of the fonts and the font family used.

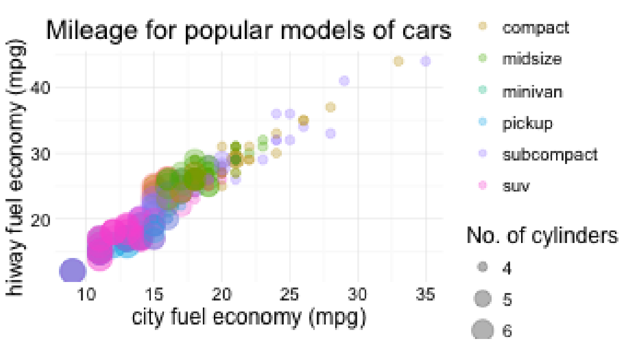
```
q + theme_gray(base_size=14, base_family="serif")
q + theme_gray(base_size=10, base_family="Times")
```

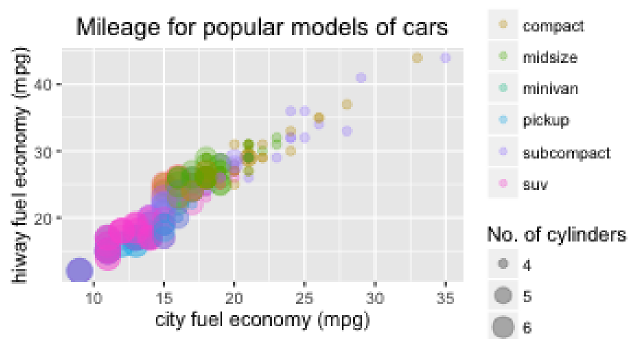




The default theme can be set using `theme_set()`, which returns the previous theme, in case you want to revert back to it later.

```
old_theme <- theme_set(theme_minimal())
q
theme_set( old_theme )
q
```

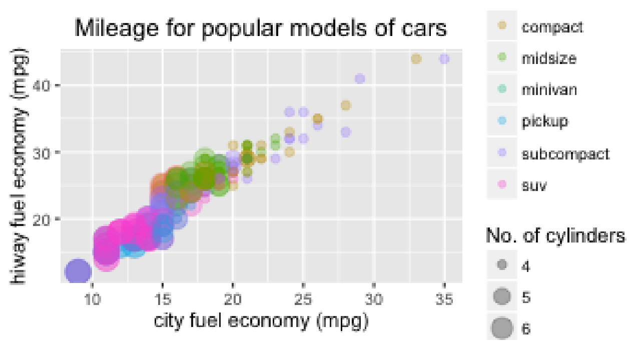


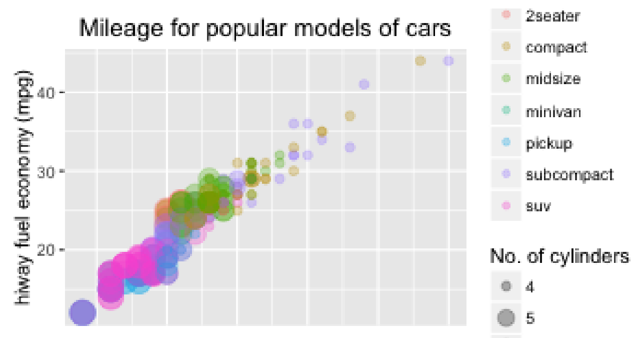


25 Modifying theme elements

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()`.

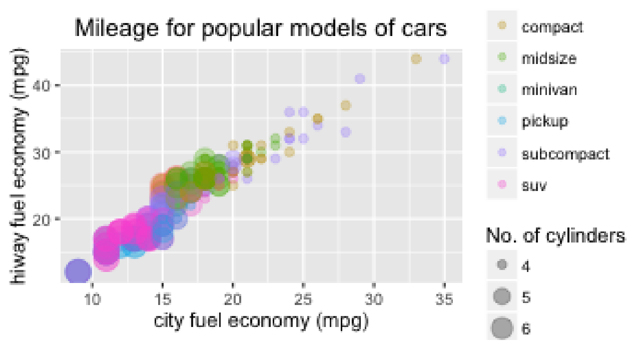
```
q
q + theme( axis.text.x = element_blank(),
               axis.title.x = element_blank(),
               axis.ticks.x = element_blank()
               )
```

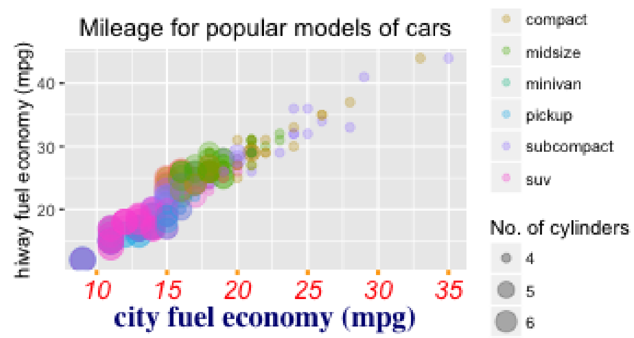




Alternatively, we might like to change the style of the axis labeling using `element_text()` and `element_line()`. In the example below different colors are used to make it clear which element is which.

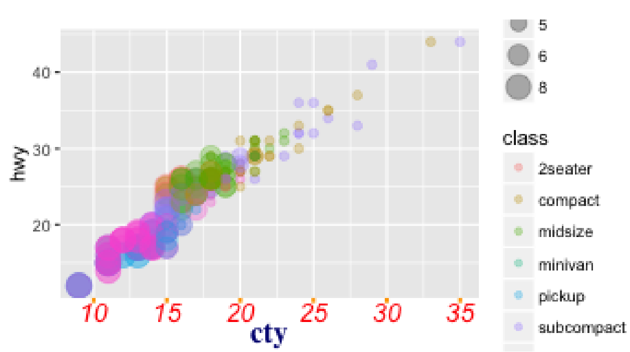
```
q
q + theme( axis.text.x = element_text(colour=red, size=14, face=italic),
                axis.title.x = element_text(colour=navy, size=16, face=bold, family=serif),
                axis.ticks.x = element_line(colour=orange, size=1)
                )
```

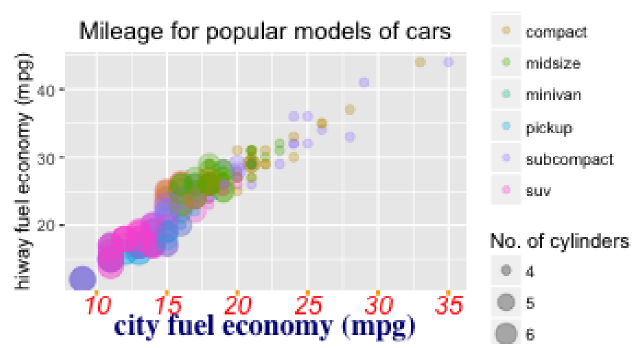




Changes to theme elements can be used to update the current default theme using `theme_update()`.

```
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)
                )
p
q
```





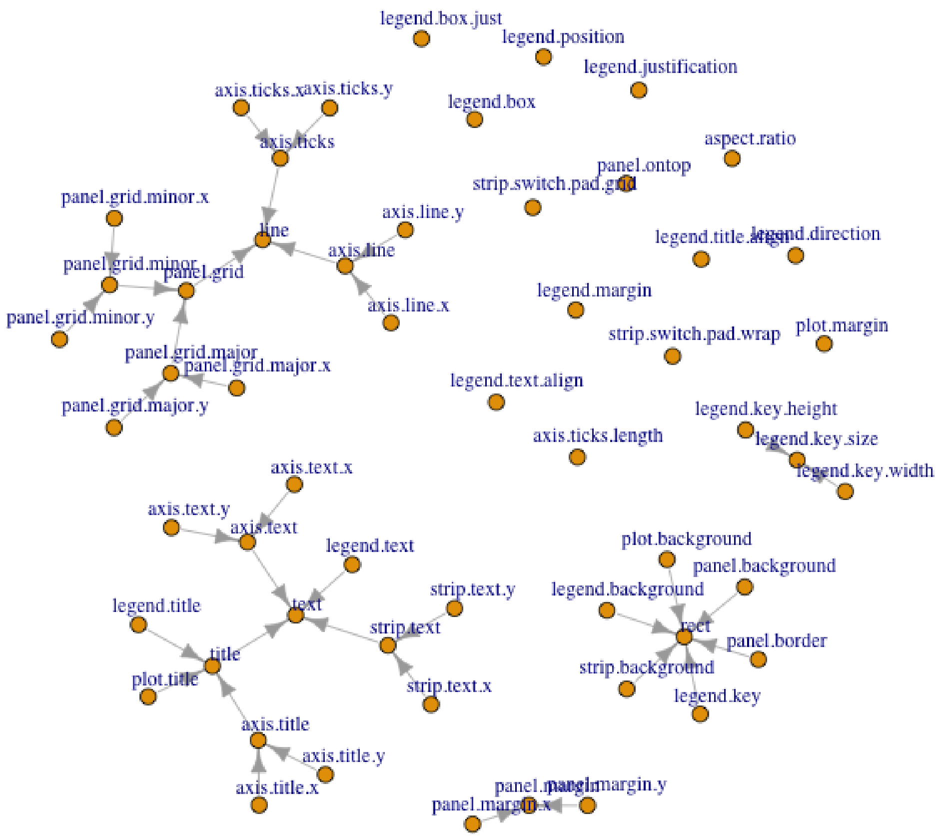
In addition to `element_blank()`, `element_line()`, and `element_text()`, there is also `element_rect()` for setting features of rectangular elements (like the plot background).

```r
apropos("element_")

## [1] "element_blank" "element_grob"  "element_line"  "element_rect"  "element_text"
```

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

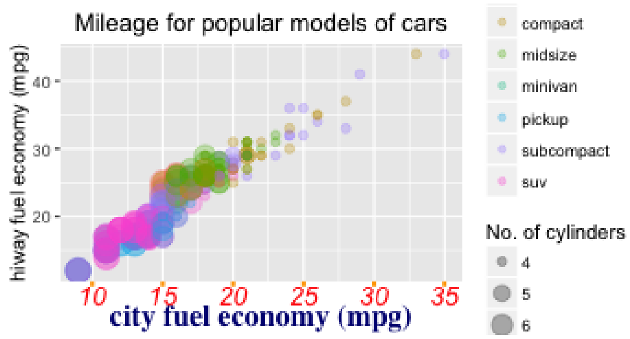
25.1 Taking advantage of the hiearchy

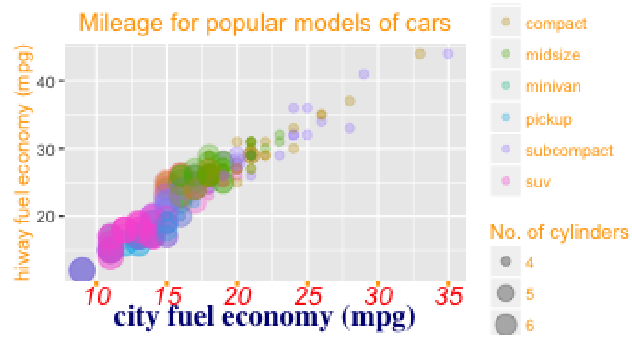
The hierarchy of theme element names is illustrated in this graph. (The R code that generates this graph can be found at https://github.com/wch/ggplot2/wiki/New-theme-system.)



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



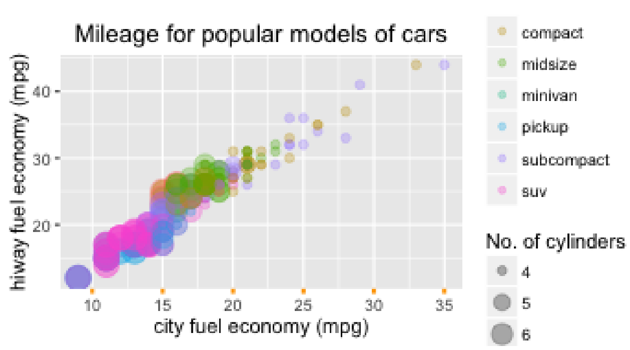


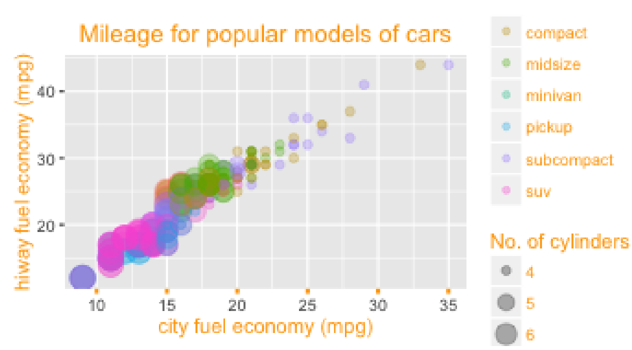
To understand why this first attempt didn't quite work, we need to look a little more closely at themes and inheritance.

```
current_theme <- theme_update()  # dont make any changes, just store the current theme
current_theme$axis.text.x


## List of 10
##  $ family    : NULL
##  $ face      : chr "italic"
##  $ colour    : chr "red"
##  $ size      : num 14
##  $ hjust     : NULL
##  $ vjust     : NULL
##  $ angle     : NULL
##  $ lineheight: NULL
##  $ margin    : NULL
##  $ debug     : NULL
##  - attr(*, "class")= chr [1:2] "element_text" "element"
```

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.

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





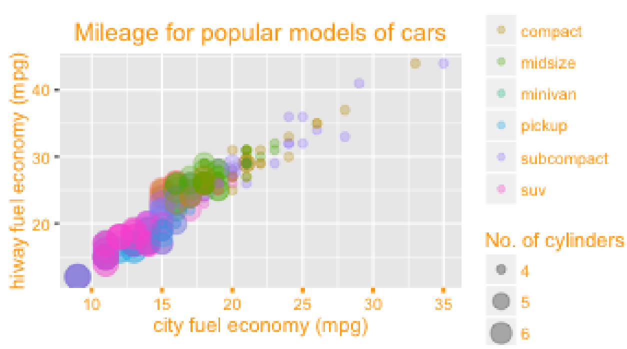
All the text is orange now except for `axis.text`. Inspecting `theme_gray()` shows us why.

```
theme_gray()$text$colour
```

```
## [1] "black"

theme_gray()$axis.text$colour

## [1] "grey30"

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.

```
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.

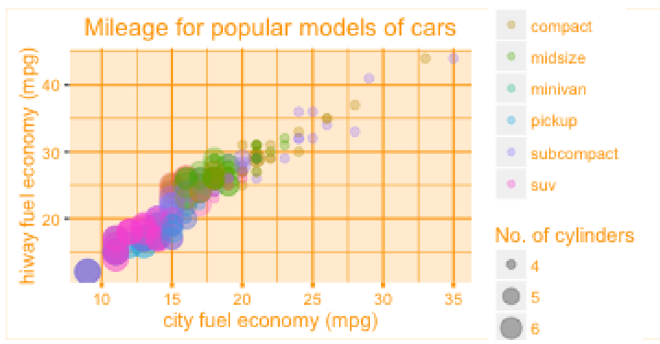
25.2 Saving your theme

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

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

The resulting theme can then be applied as the new default theme, or it can be added to individual plots.

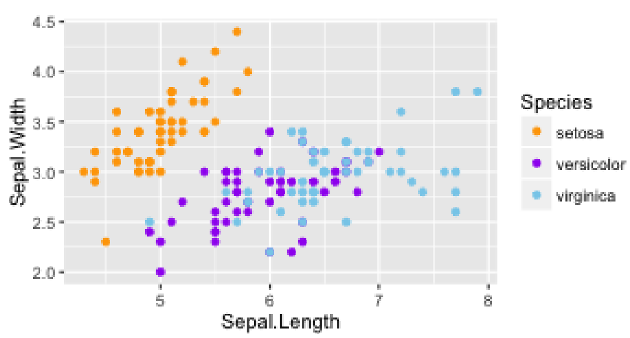
```
q + theme_orange
```

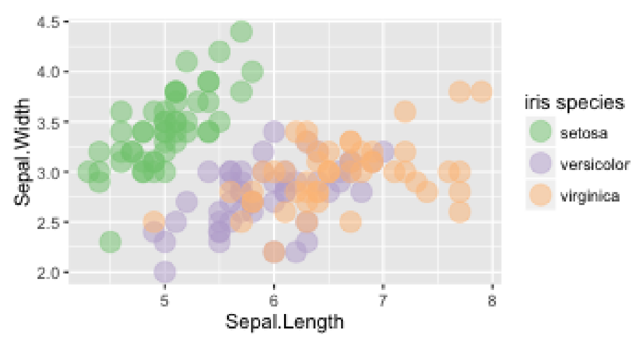


26 Controlling how aesthetics are mapped

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.

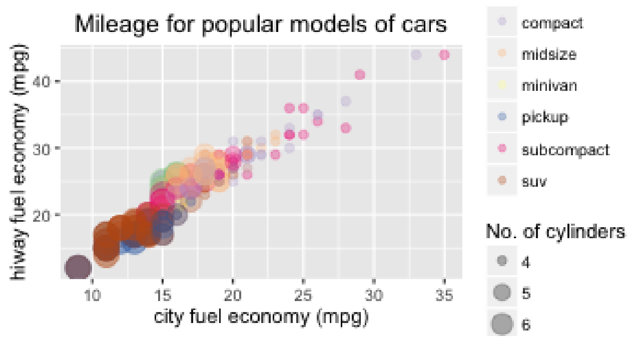
```
theme_set(theme_gray())
p2 <- ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species))
p2 + geom_point() + scale_color_manual(values = c("orange", "purple", "skyblue"))
p2 + geom_point(size = 5, alpha = 0.5) + scale_color_brewer(type = "qual",
    palette = 1, name = "iris species")
```

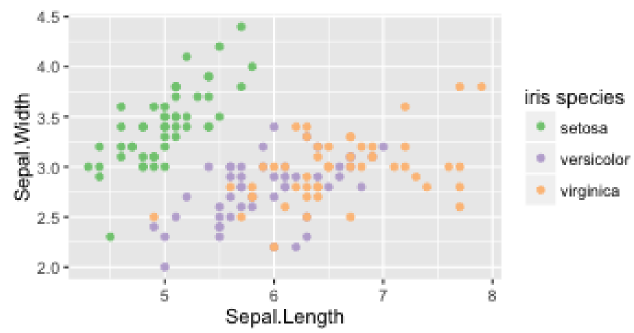




There is no mechanism in ggplot2 for setting default scales as part of a theme, but the scales can be saved for easier reuse in multiple plots.

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

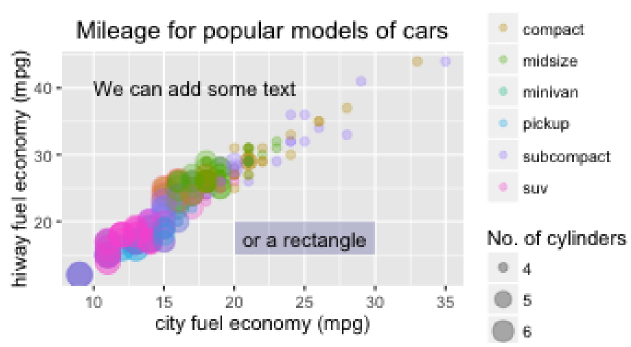




27 Annotation

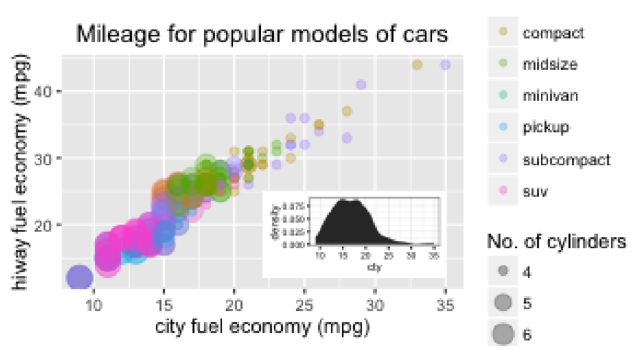
ggplot2 provides some additional utilities for adding small amounts of additional information to a plot. Like many other ggplot2 functions we have seen, annotate() adds a layer to the plot. Its first argument is a geom. Unlike the other ggplot2 functions, the annotate() function does not require that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

```
q + 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",.2)) +
    annotate("text", x=25, y=17.5, label="or a rectangle")
```



Inset plots can be added using annotation_custom().

```
inset <- ggplot(aes(cty), data=mpg) + stat_density() + theme_bw(6)
insetG <- ggplotGrob(inset)
q + 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.