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  Once caveat: This book is now a bit out of date and is not completely consistent with the newest versions of `ggplot2`.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.

  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 treat
## 1 13 26  1     39          1  yes 25.112 58.41      0   black      no       4   cocaine   yes
## 2 56 62  2     43         NA <NA> 26.670 36.04      1   white      no       7   alcohol   yes
## 3  0  0  3     41          0   no  6.763 74.81     13   black      no       2    heroin    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 sexrisk
## 1        13        26  1     39          1  yes 25.11 58.41      0   black      no       4
## 2        56        62  2     43         NA <NA> 26.67 36.04      1   white      no       7
##   substance treat
## 1   cocaine   yes
## 2   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
## 1 1978-01-01   7701         1
## 2 1978-01-02   7527         2
## 3 1978-01-03   8825         3
## 4 1978-01-04   8859         4
## 5 1978-01-05   9043         5
## 6 1978-01-06   9208         6
```

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.

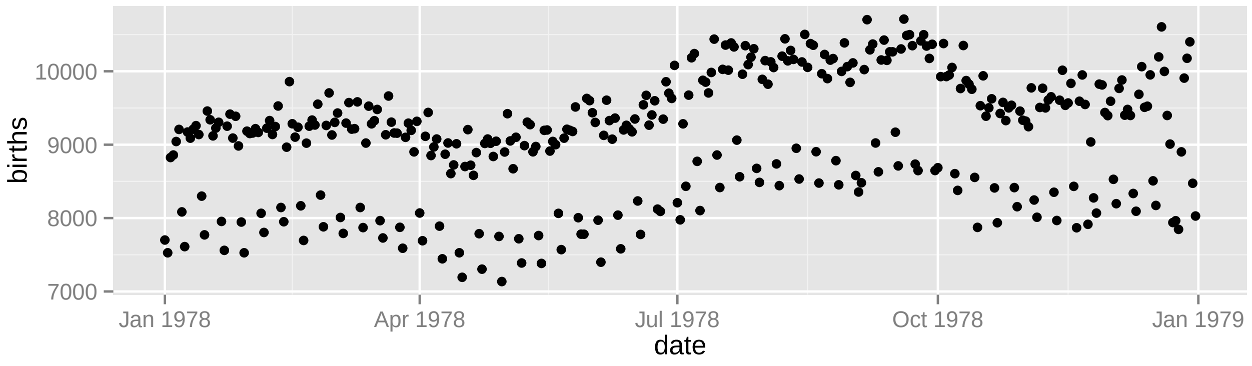
guide helps user convert visual data back into raw data

- legends, axes

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

1.1 Describing a plot

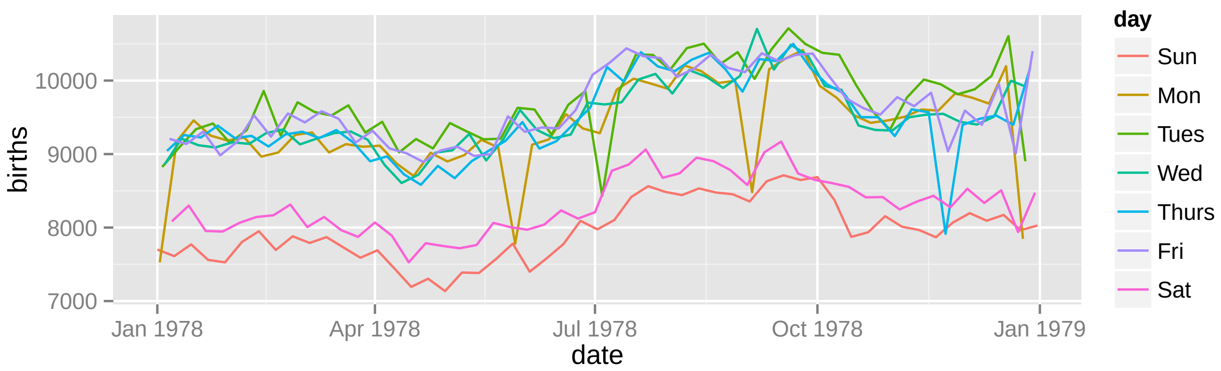
One key to successfully creating plots with ggplot2 is learning to describe plots using the grammar outlined above. For example, consider the following plot.



- **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.)

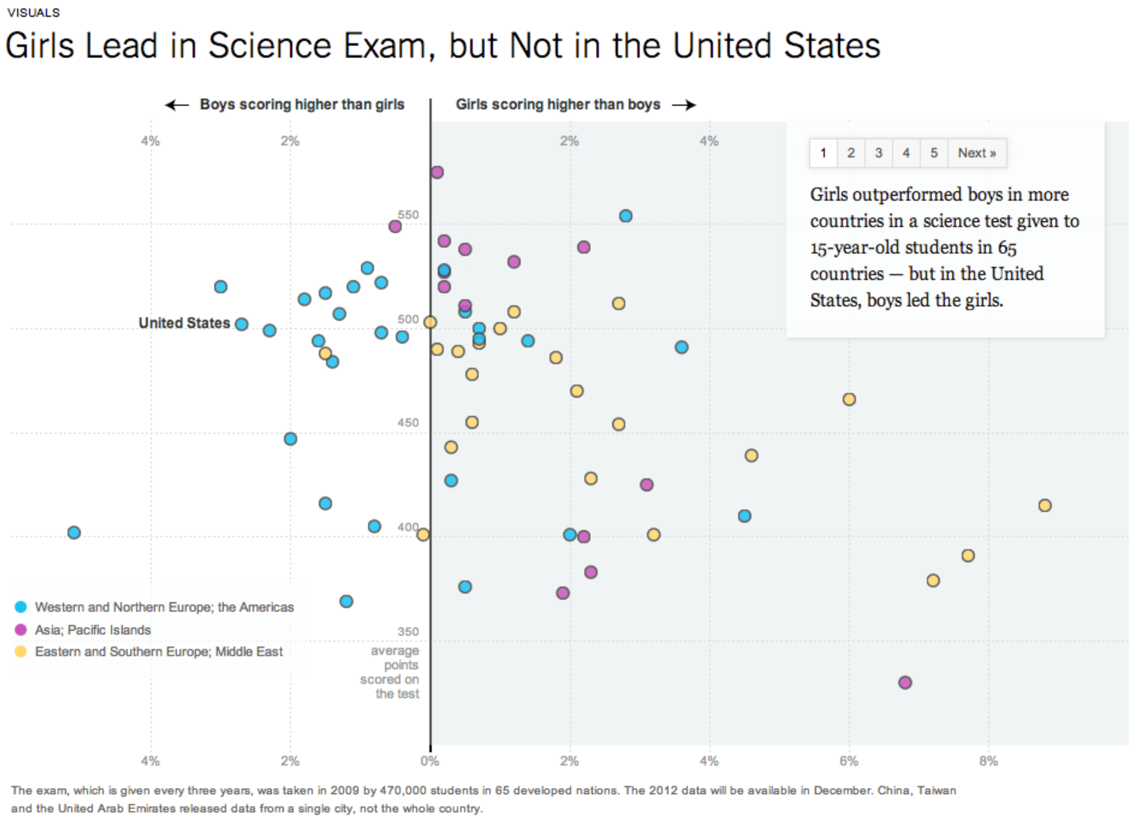
- We are now **mapping** the day of the week to the color **aesthetic**.
- An additional **guide** has beeen added to show which days are mapped to which colors. (The mapping takes days to colors for displaying and the guide helps us go from colors to days for interpreting the result.)

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) )

## Error:  No layers in plot
```

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.

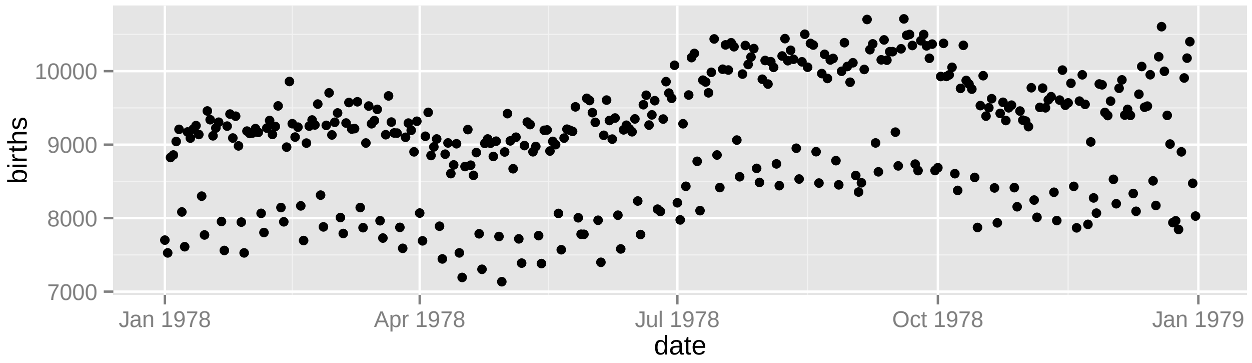
2.2 Layers, geoms, and stats

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:

[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.

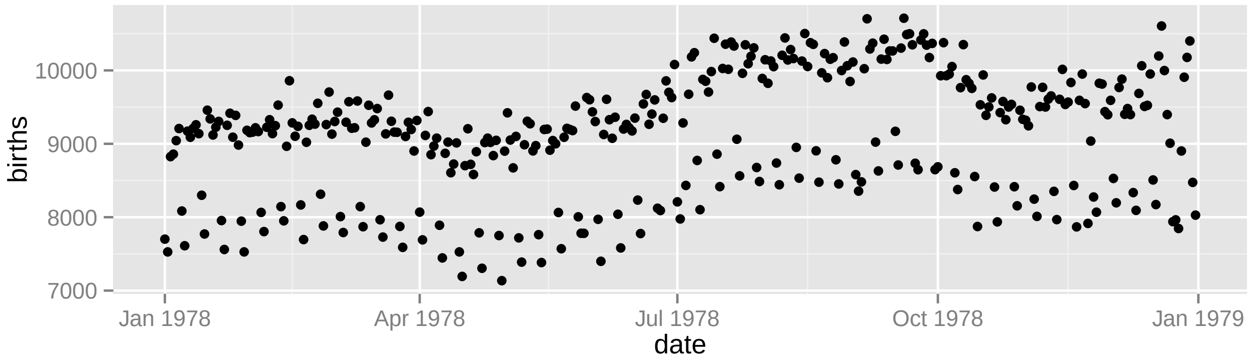
[5]The situation is actually a little bit more complicated. The mapping establishes the link between the data and an aesthetic, but a **scale** determines precisely which values of the data are mapped to which aesthetic values. For now, we will use the default scales, but eventually we will want to know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

```
# 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")
```



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

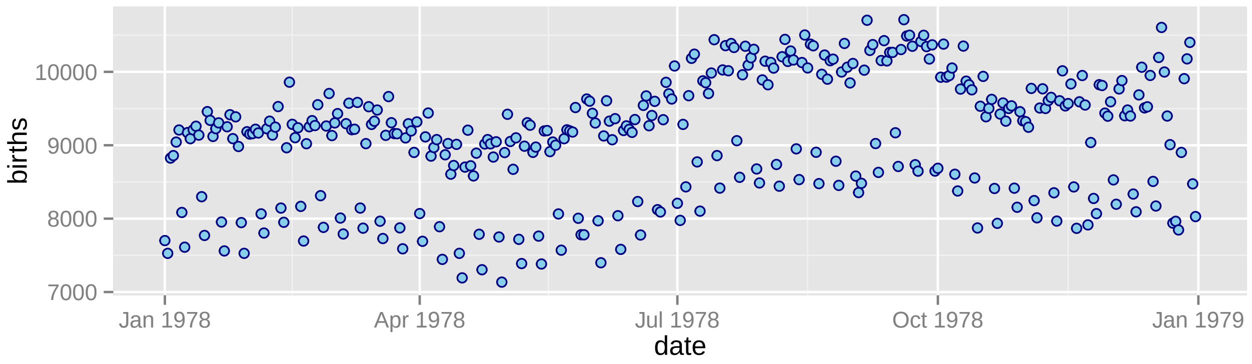
```
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point()   # stat_identity is the default
```



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.

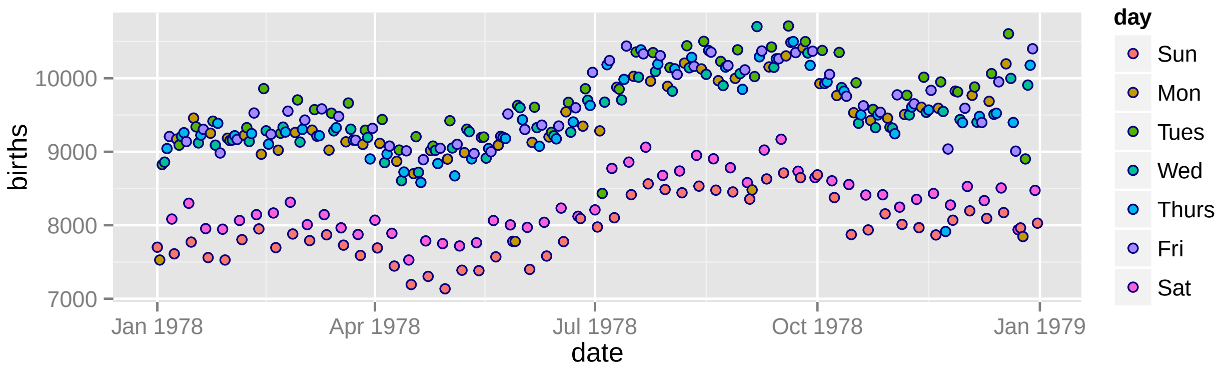
```
# shape = 21 is a circle with separate color and fill aesthetics.
ggplot( Births78, aes(x=date, y=births) ) +
  geom_point(color="navy", fill="skyblue", shape=21)
```



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.

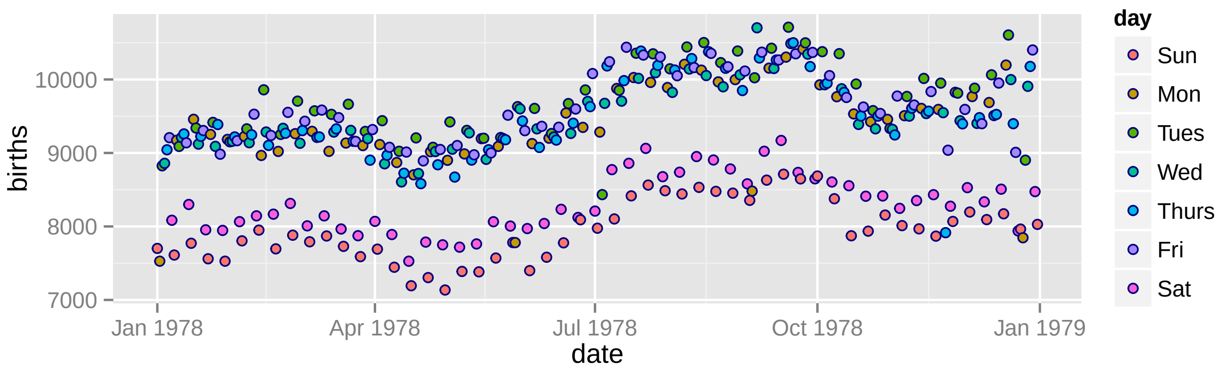
If we want the fill to show the days of the week, we must use **mapping** rather than **setting**. To make things easier, let's add a variable to our data that stores the day of the week. The `wday()` function from the `lubridate` package will do the computation of weekday from date and `mutate()` creates 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 day
## 1 1978-01-01   7701         1 Sun
## 2 1978-01-02   7527         2 Mon


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



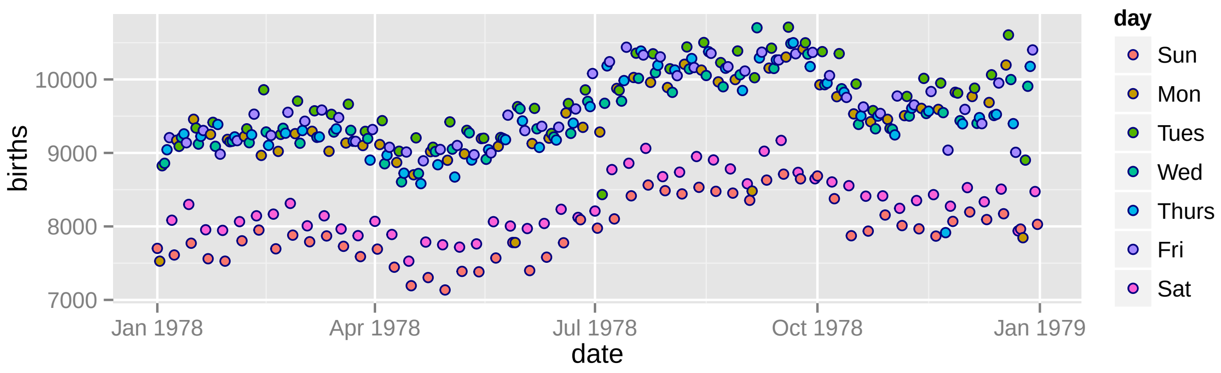
Notice that `fill=` is now inside `aes()` in the call to `ggplot()` rather than an argument to `geom_point()`. This is how we distinguish between **mapping** and **setting**. We could also have made the plot this way:

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



or a bit more verbosely using

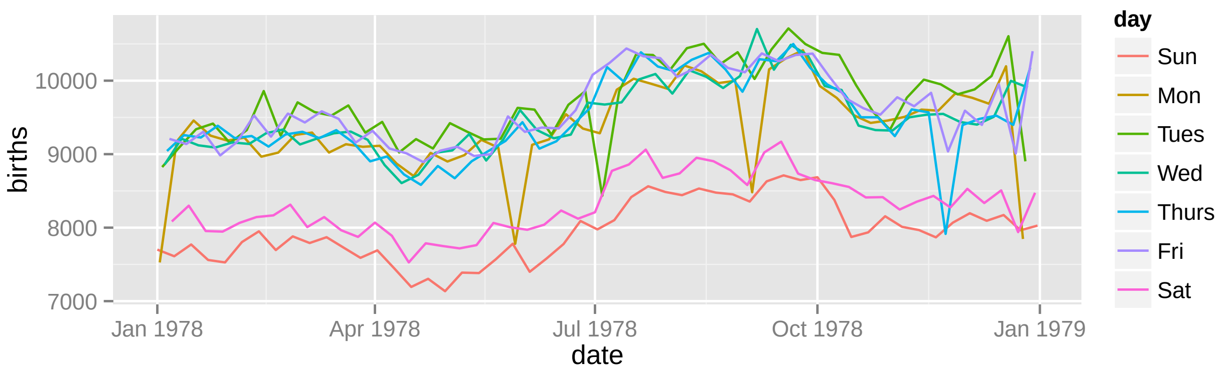
```
ggplot( data=Births78, mapping=aes(x=date, y=births) ) +
  geom_point( mapping=aes(fill=day), color="navy", shape=21 )
```



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 multliple 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.

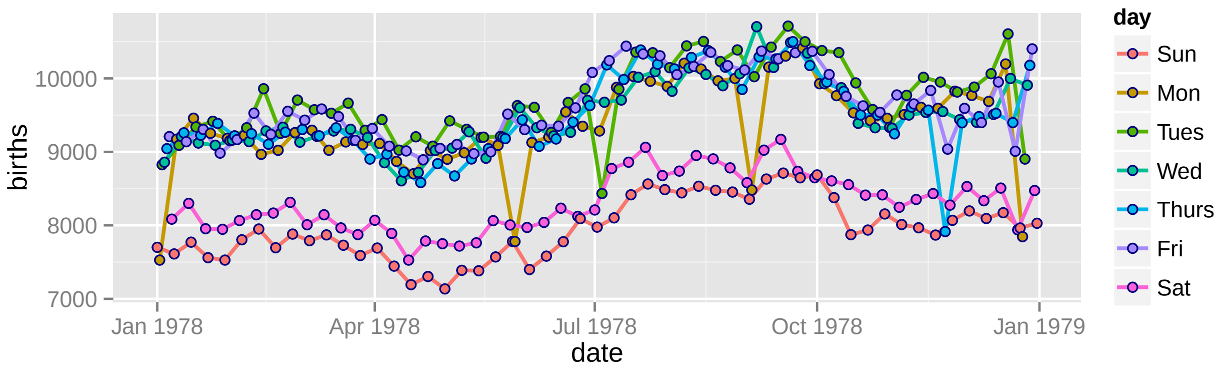
If we would prefer to use lines, rather than points, we can easily change the geom:

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



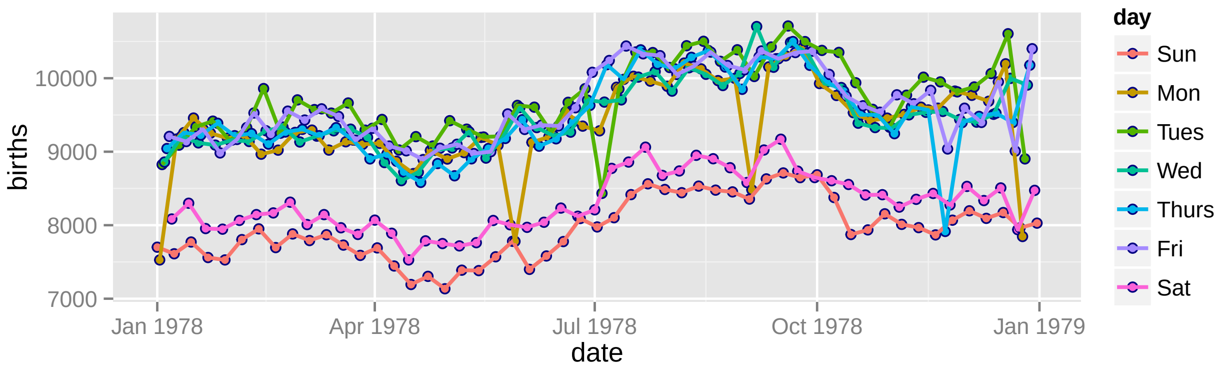
We can even choose to do both points and lines by creating two layers:

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



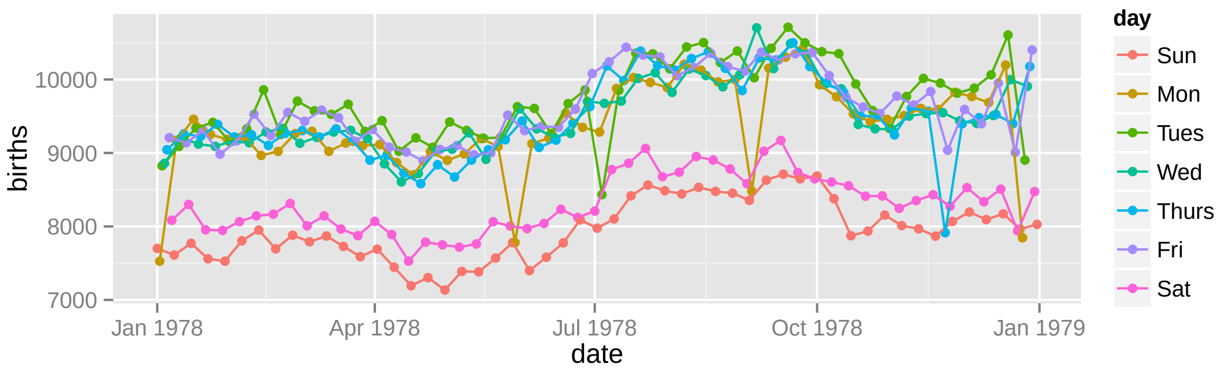
The order of the layers matters. Each layer is created *on top* of the previous layers.

```
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 )
```

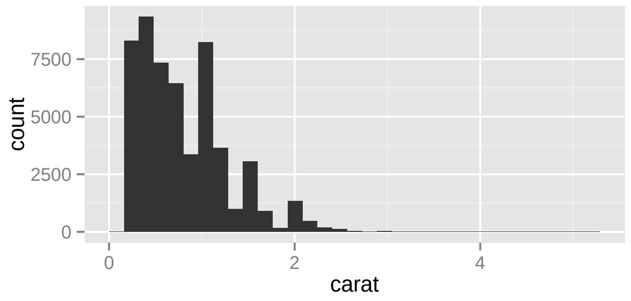


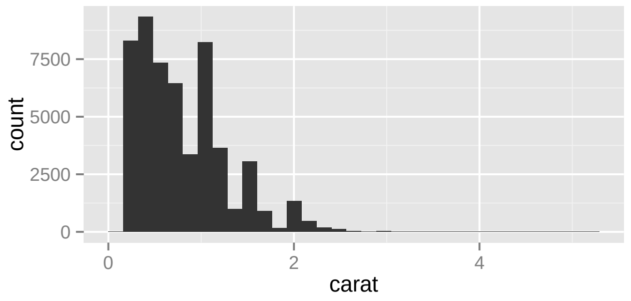
2.4 Stats

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)


##   carat     cut color clarity depth table price    x    y    z
## 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()

## stat_bin:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.

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

## stat_bin:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.
```

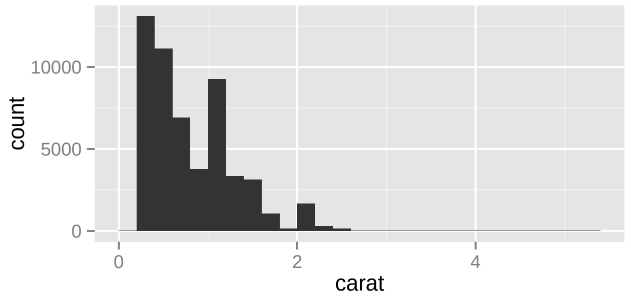


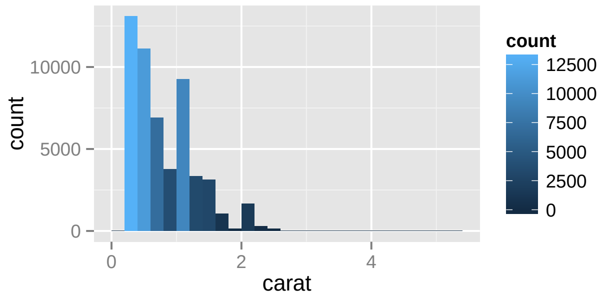


This is mostly straightforward. We have mapped `carat` to the x-axis and selected the histogram geom. We are even warned that it would be better if we selected the width of the bins ourselves rather than using the default value. We'll do that in our next histogram.

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:

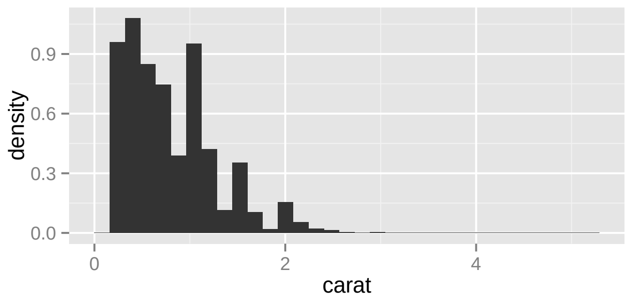
```
ggplot(data=diamonds, aes(x=carat, y=..count..)) +
  geom_histogram(binwidth=.2)
ggplot(data=diamonds, aes(x=carat, fill=..count..)) +
  geom_histogram(binwidth=.2)
```

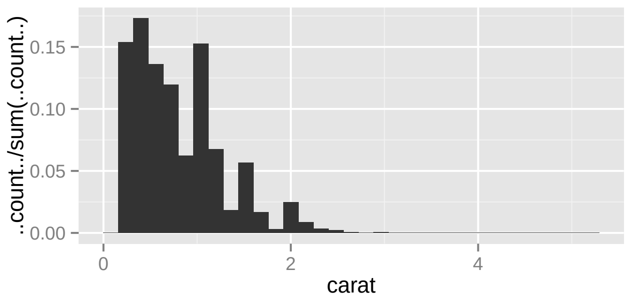




Other variables in the statified data frame allow us to create other types of histograms.

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

## stat_bin:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.

# 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:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.
```



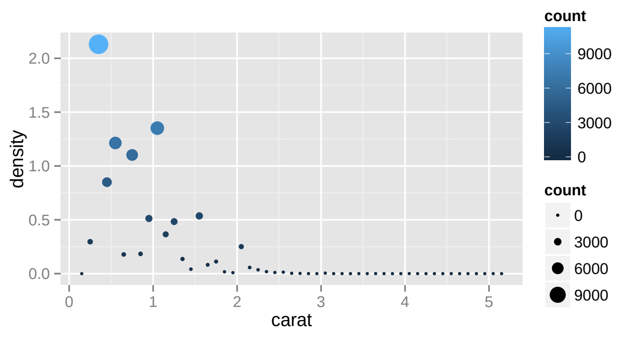


The list of additional variables computed by `stat_bin()` can be found with

```
?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.

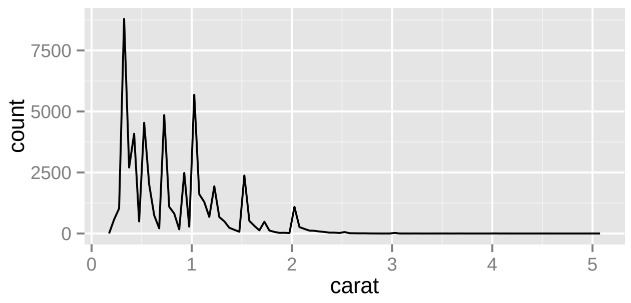
2.6 Some more geoms and stats

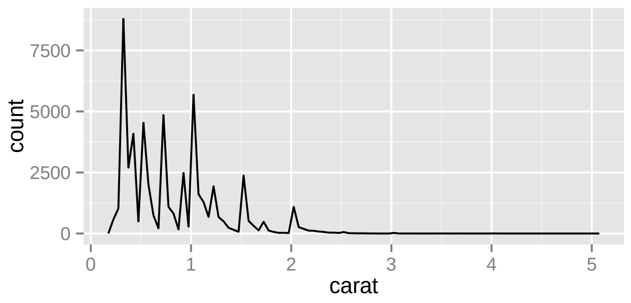
2.6.1 Frequency polygons

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)

## ymax not defined:  adjusting position using y instead
```

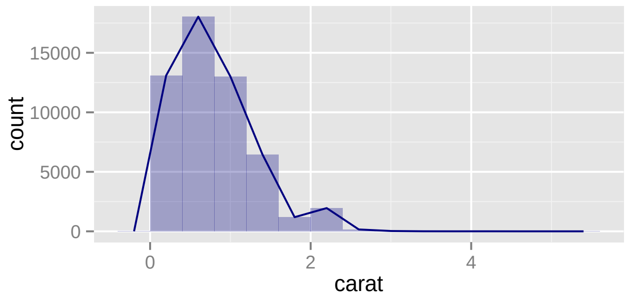
[6]This is a bit of an oversimplification. Actually, the aesthetics get computed twice, once before the stat and again after. For a histogram, for example, we need to look at the aesthetics to figure out which variable to bin (that's the stats job), but it isn't until after the binning that we will know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

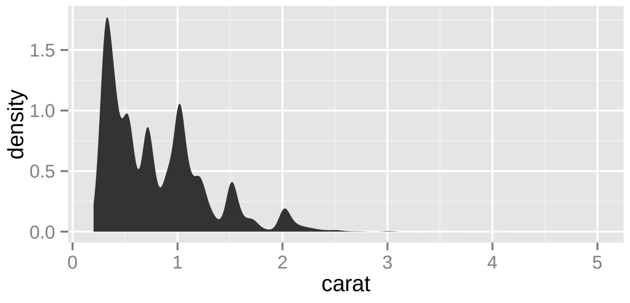
```
ggplot(data=diamonds, aes(x=carat)) +
  geom_histogram(alpha=.3, binwidth=0.4, fill="navy") +
  geom_line(stat="bin", binwidth=0.4, color="navy")
```



2.6.2 Density plots

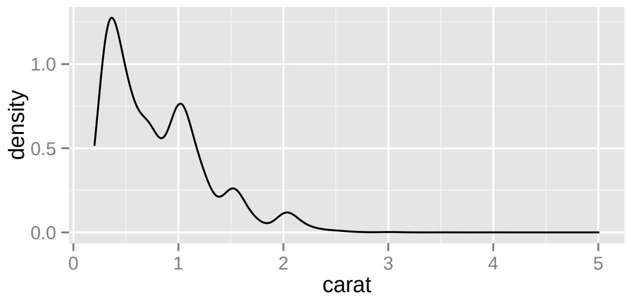
Another common representation of the distribution of a quantitative variable is a density plot. This requires a new stat: `stat_density()`.

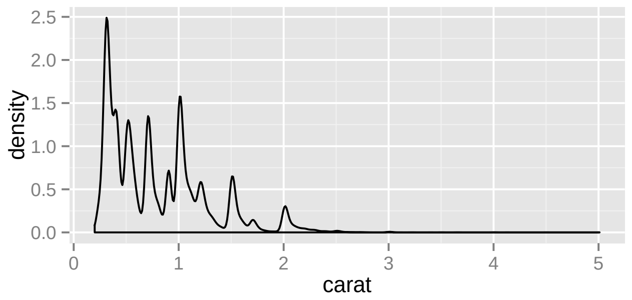
```
ggplot(data=diamonds, aes(x=carat)) +
  stat_density()
```



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.

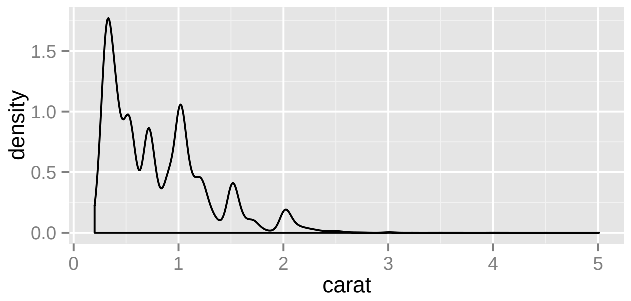
```
ggplot(data=diamonds, aes(x=carat)) +
  stat_density(geom="line", adjust=2)

## ymax not defined:  adjusting position using y instead

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

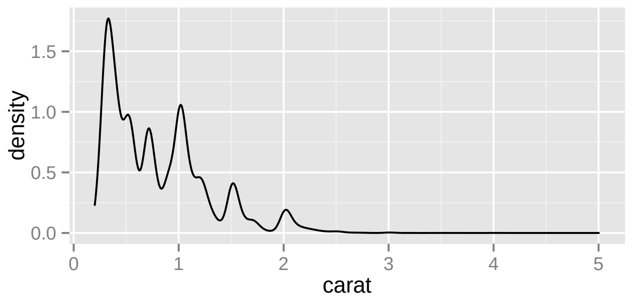


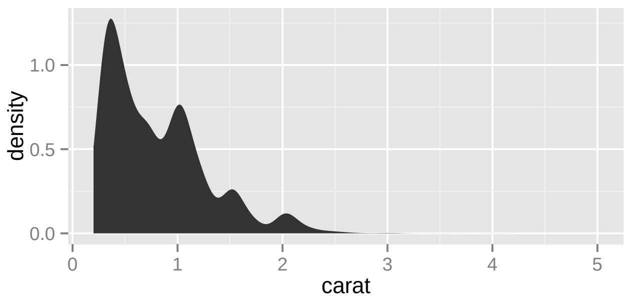


Alternatively, we could specify the geom and stat this way

```
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)
```







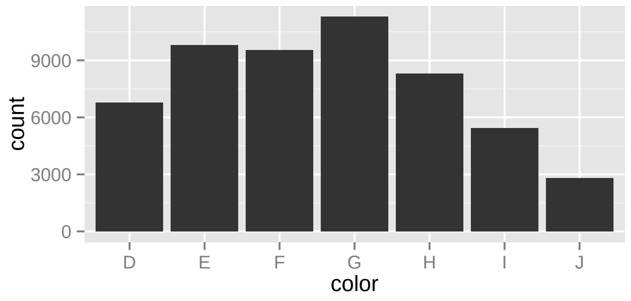
Whichever way we choose to do it, we are telling `ggplot()` both the stat and the geom that should be used.

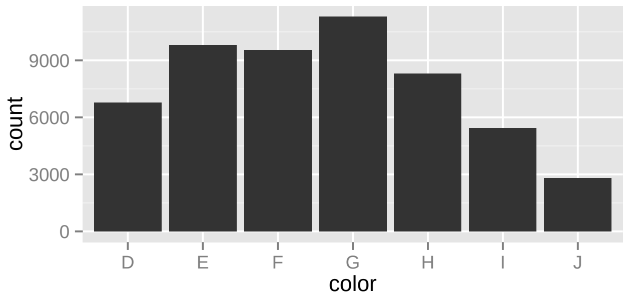
In this particular case, we need to be cautious not to oversmooth and lose part of the story. We have a lot of data (53940 rows), so the spikey appearance of these plots is likely a feature and not an artifact. In fact, there is a plausible explanation: the peaks coincide with round numbers. Likely the 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
ggplot(data=diamonds, aes(x=color)) +
  geom_bar()    # stat_bin is the default
```



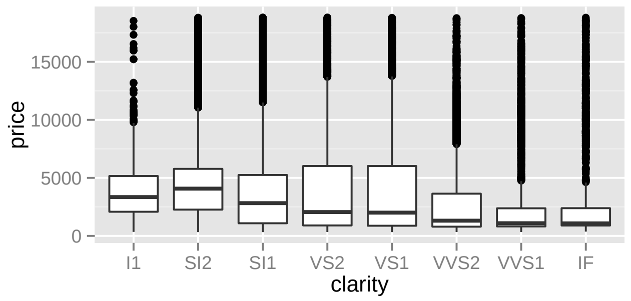


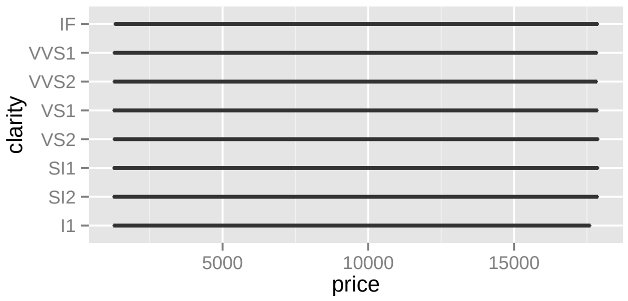
2.7 Boxplots

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.

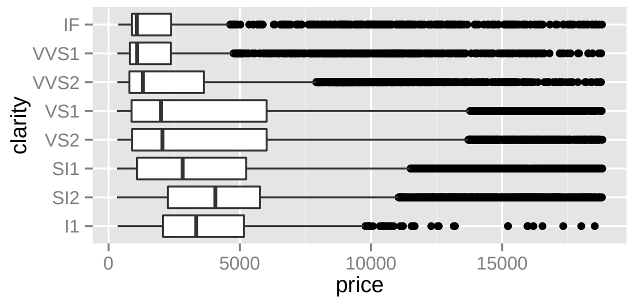
```
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_boxplot()
# this doesnt do what we might have expected.
ggplot( data=diamonds, aes(y=clarity, x=price) ) +
  geom_boxplot()

## Warning:  position_dodge requires constant width:  output may be incorrect
## Warning:  position_dodge requires non-overlapping x intervals
```





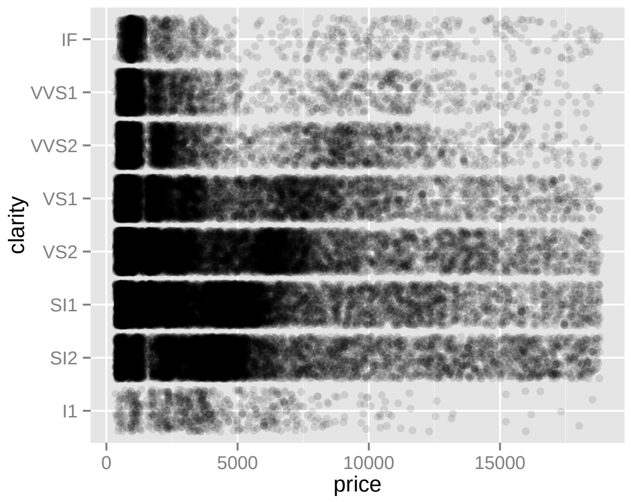
If we want horizontal boxplots, we need to flip them by using a different coordinate system.

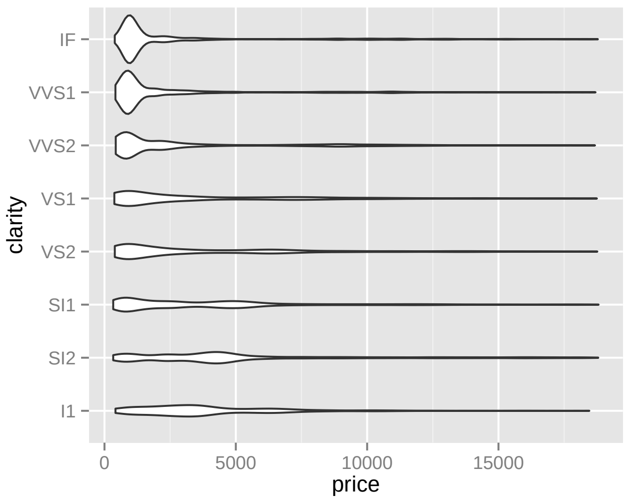
```
ggplot( data=diamonds, aes(x=clarity, y=price) ) +
  geom_boxplot() +
  coord_flip()
```



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_crossbar"  "geom_density"   "geom_density2d"
## [11] "geom_dotplot"   "geom_errorbar"  "geom_errorbarh" "geom_freqpoly"  "geom_hex"
## [16] "geom_histogram" "geom_hline"     "geom_jitter"    "geom_line"      "geom_linerange"
## [21] "geom_map"        "geom_path"       "geom_point"      "geom_pointrange" "geom_polygon"
## [26] "geom_quantile"   "geom_raster"     "geom_rect"       "geom_ribbon"     "geom_rug"
## [31] "geom_segment"    "geom_smooth"     "geom_step"       "geom_text"       "geom_tile"
## [36] "geom_violin"     "geom_vline"
```

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

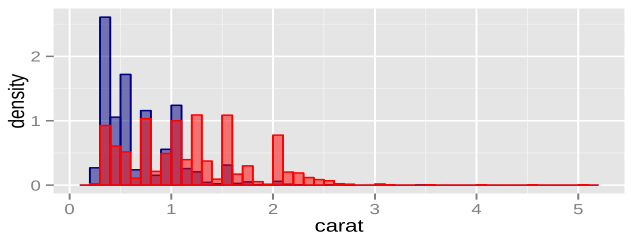
```
##  [1] "stat_abline"      "stat_bin"         "stat_bin2d"       "stat_bindot"
##  [5] "stat_binhex"      "stat_boxplot"     "stat_contour"     "stat_density"
##  [9] "stat_density2d"   "stat_ecdf"        "stat_ellipse"     "stat_function"
## [13] "stat_hline"       "stat_identity"    "stat_qq"          "stat_quantile"
## [17] "stat_smooth"      "stat_spoke"       "stat_sum"         "stat_summary"
## [21] "stat_summary_hex" "stat_summary2d"   "stat_unique"      "stat_vline"
## [25] "stat_ydensity"
```

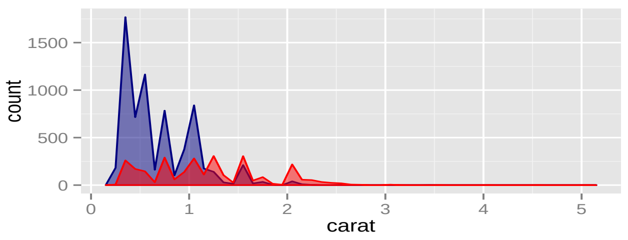
The documentation for these functions can be consulted to learn the options available for each.

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)
```

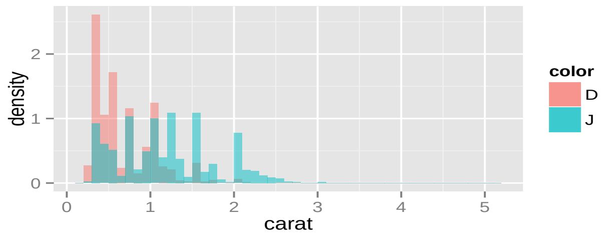


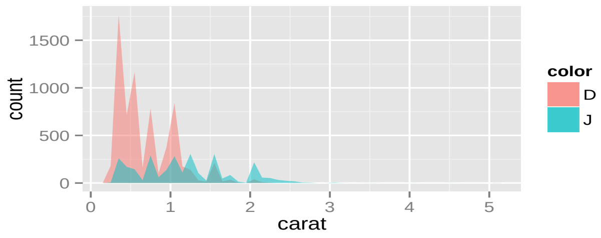


Notice that the plots above do not have a legend for fill. This is because fill is set, not mapped. We could use mapping instead, like this:

```
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)
```

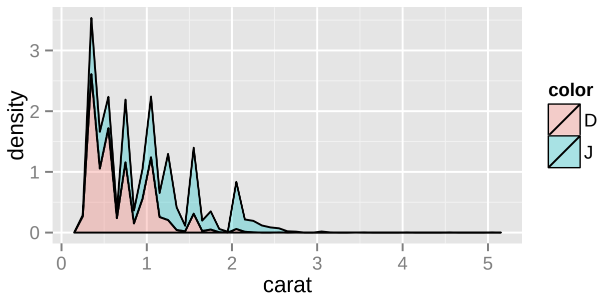
```
ggplot(mapping=aes(x=carat, ymax=..density..)) +
  stat_bin(data=D, geom="polygon", aes(fill=color), alpha=0.5, binwidth=0.1) +
  stat_bin(data=J, geom="polygon", aes(fill=color), alpha=0.5, binwidth=0.1)
```





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")

## ymax not defined:  adjusting position using y instead
```



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()`

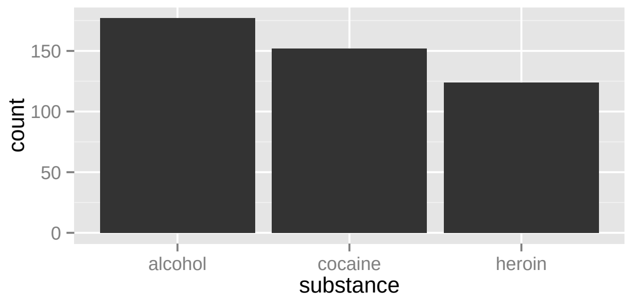
ggplot2 provides a short-cut that makes it easy to describe certain simple plots using the `qplot()` function. In it's simplest form, `qplot()` requires only two or three pieces of information:

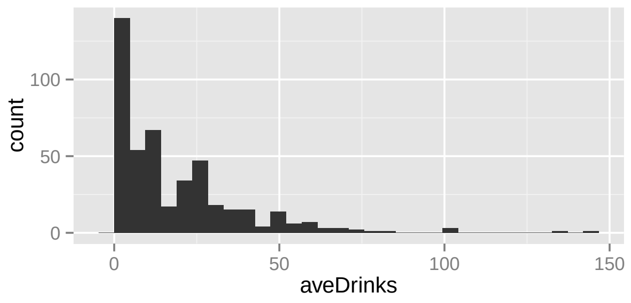
- the name(s) of one or two variable(s) providing the data to be displayed (`x=` and `y=`), and
- the name of a data frame containing those variables (`data=`).

`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:  binwidth defaulted to range/30.  Use 'binwidth = x' to adjust this.
```

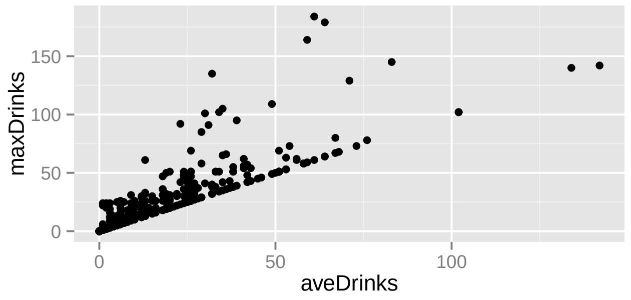


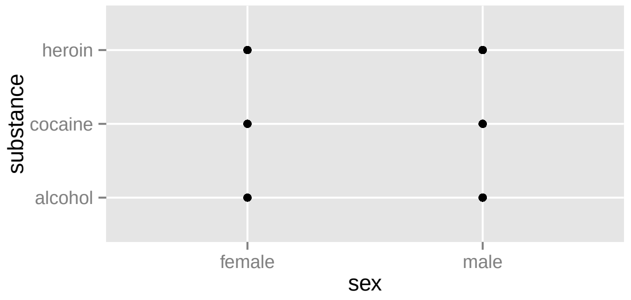


3.2 Two variable plots

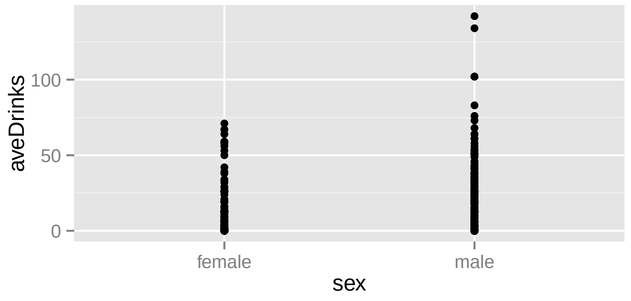
When two variables are provided, the result is a scatter plot. The first variable goes on the horizontal axis and the second on the vertical axis.

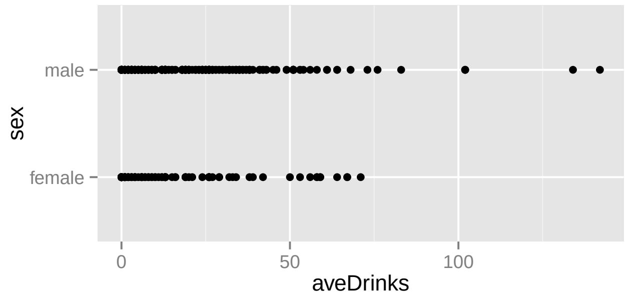
```
# two quantitative variables -> geom_point (+ stat_identity)
qplot( aveDrinks, maxDrinks, data=HELPrct)
# two categorical variables -> geom_point (+ stat_identity)
qplot( sex, substance, data=HELPrct)
```





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



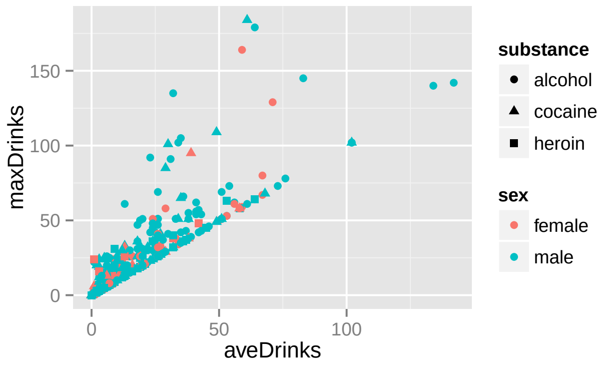


Overplotting of data or labels in these plots might render them less than ideal. This is a common problem in large data sets or when one or more variables are categorical. We'll learn some methods for dealing with this soon.

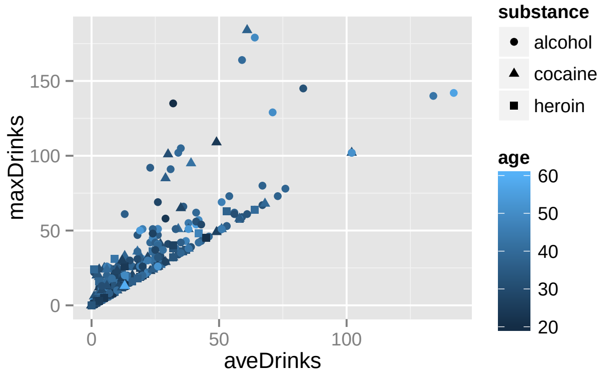
3.3 Mapping aesthetics with qplot()

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

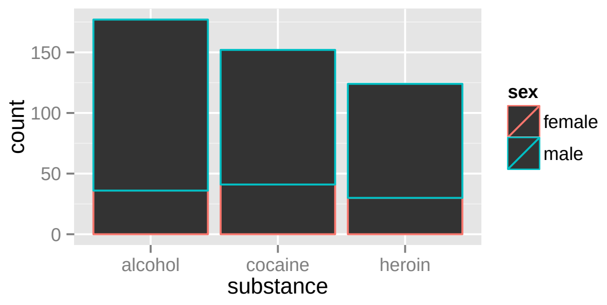
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=sex, shape=substance)
```

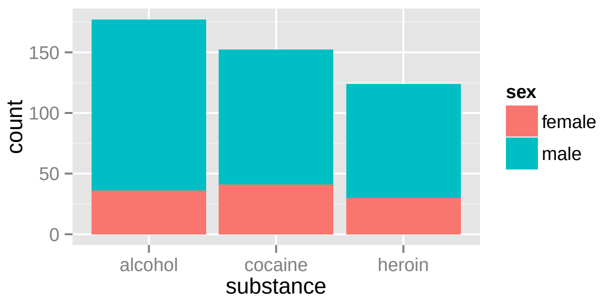


```
qplot( aveDrinks, maxDrinks, data=HELPrct, shape=substance, color=age)
```



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

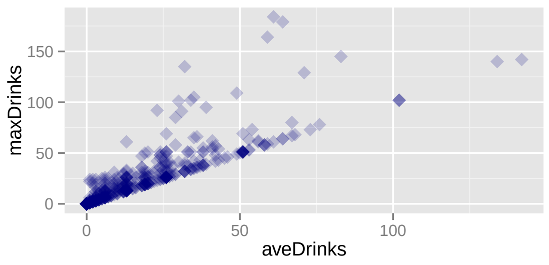




3.4 Setting aesthetics

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") )
```

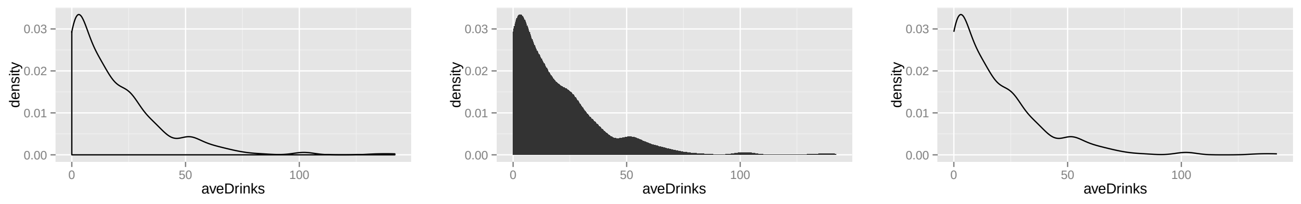


The use of alpha to make the points quite transparent illustrates one way to deal with overplotting. Where there is more data, the points (diamond-shaped in this plot) become darker.

3.5 Choosing non-default geoms and stats

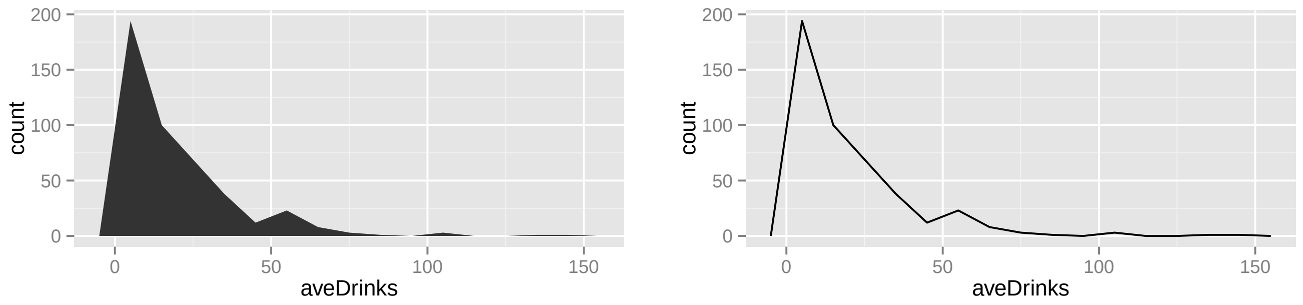
We can also select non-default geoms and stats.

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



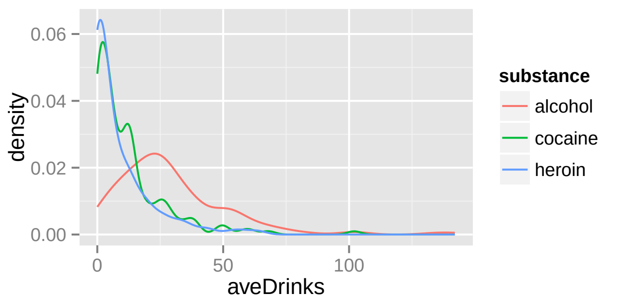
Similarly, we can create frequency polygons by using the `"bin"` stat used to create histograms with the `"polygon"` or `"line"` geoms.

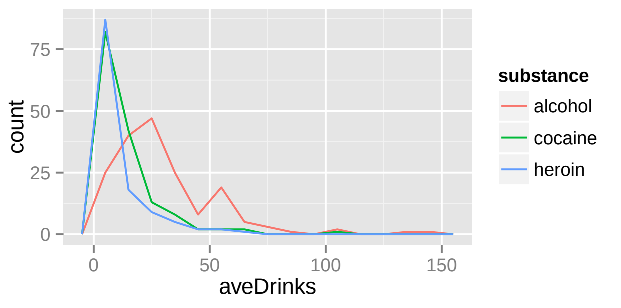
```
qplot( aveDrinks, data=HELPrct, geom="polygon", stat="bin", binwidth=10)
qplot( aveDrinks, data=HELPrct, geom="line", stat="bin", binwidth=10)
```



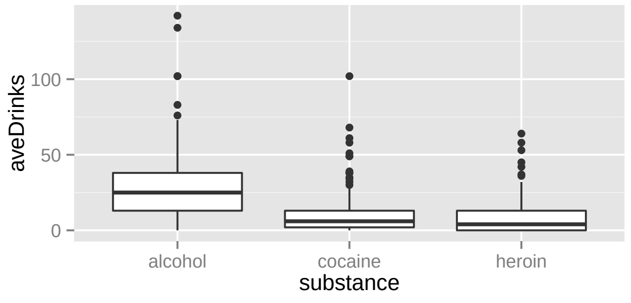
Here are some additional examples.

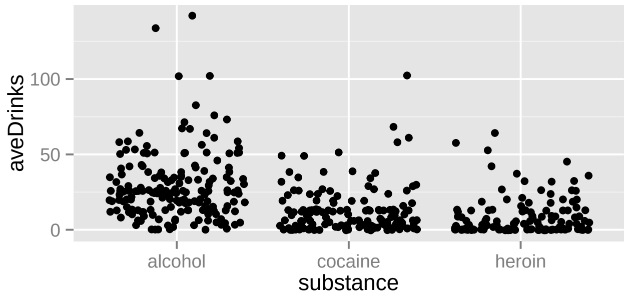
```
qplot( aveDrinks, data=HELPrct, geom="line", stat="density", color=substance)
qplot( aveDrinks, data=HELPrct, geom="line", stat="bin", binwidth=10, color=substance)
```





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

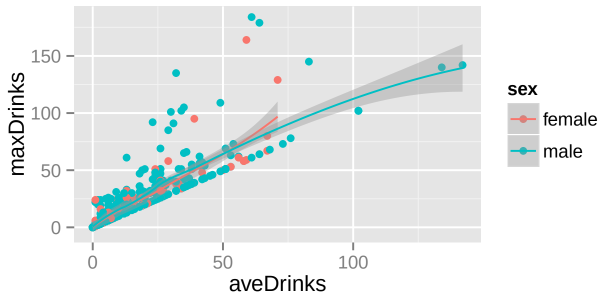


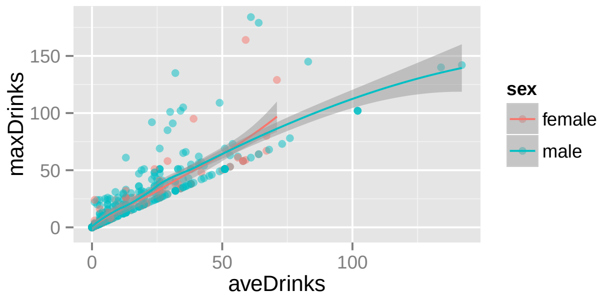


3.6 More than one geom

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

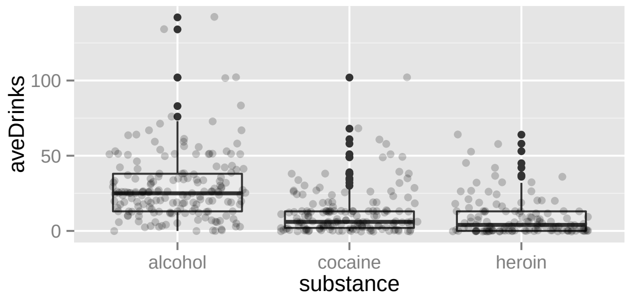
```
qplot( aveDrinks, maxDrinks, data=HELPrct, geom=c("point","smooth"), color=sex )
qplot( aveDrinks, maxDrinks, data=HELPrct, geom=c("point","smooth"), color=sex, alpha=I(.5) )
```

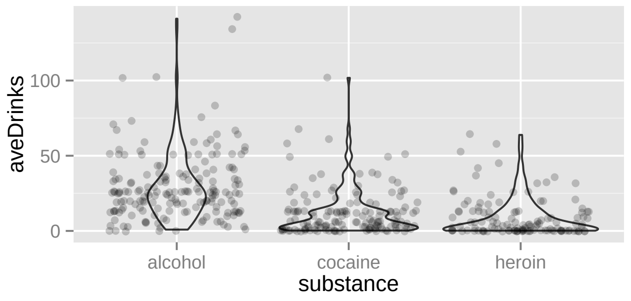




Many other combinations are possible as well.

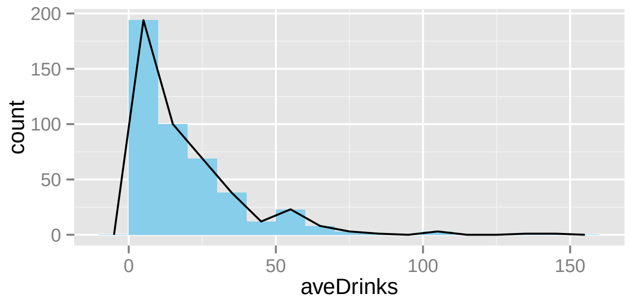
```
qplot(substance, aveDrinks, data=HELPrct, geom=c("boxplot","jitter"), alpha=I(0.2))
qplot(substance, aveDrinks, data=HELPrct, geom=c("violin","jitter"), alpha=I(0.2))
```

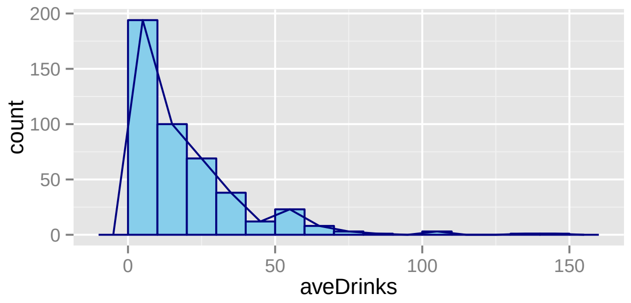




This method works as long as all the geoms involved use the same information. Notice how setting the color below sets the color for both the lines and the bars. Similarly, binwidth is shared by both geoms.

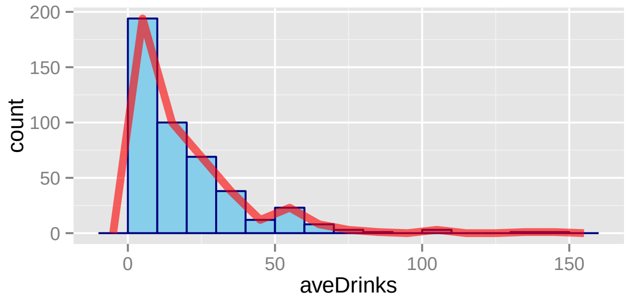
```
qplot( aveDrinks, data=HELPrct, geom=c("bar","line"), stat="bin", fill=I("skyblue"), binwidth=10 )
qplot( aveDrinks, data=HELPrct, geom=c("bar","line"), stat="bin",
    fill=I("skyblue"), color=I("navy"), binwidth=10 )
```





To achieve finer control, we need to work with geoms 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)
```



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.

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

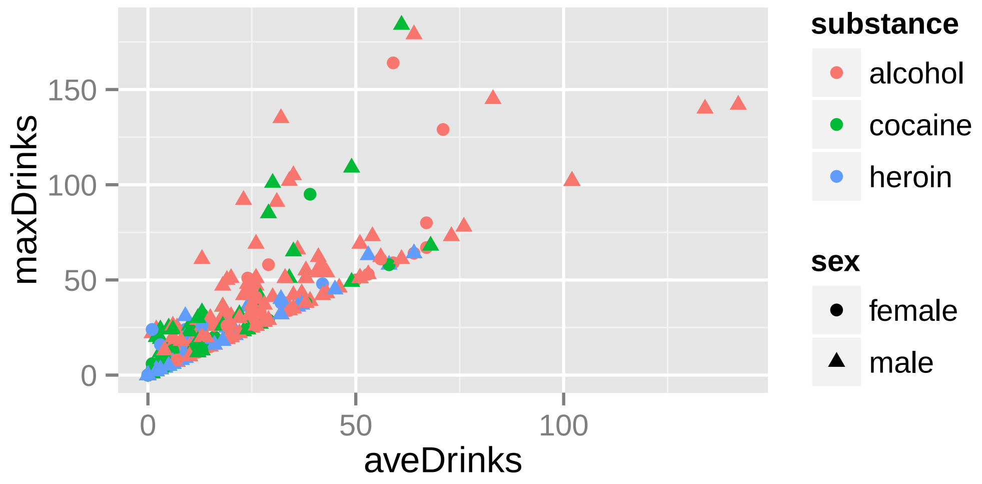
1. Use **faceting** to create multi-panel plots
2. Use **position** controls (jitter, dodge, and stack)
3. Customize the **scales** used when mapping variables to aesthetics
4. Set the limits of the viewing window for a plot.
5. Modify the labeling of axes and add titles to plots.

[7]definitions selected from Webster's online dictionary

5 Faceting

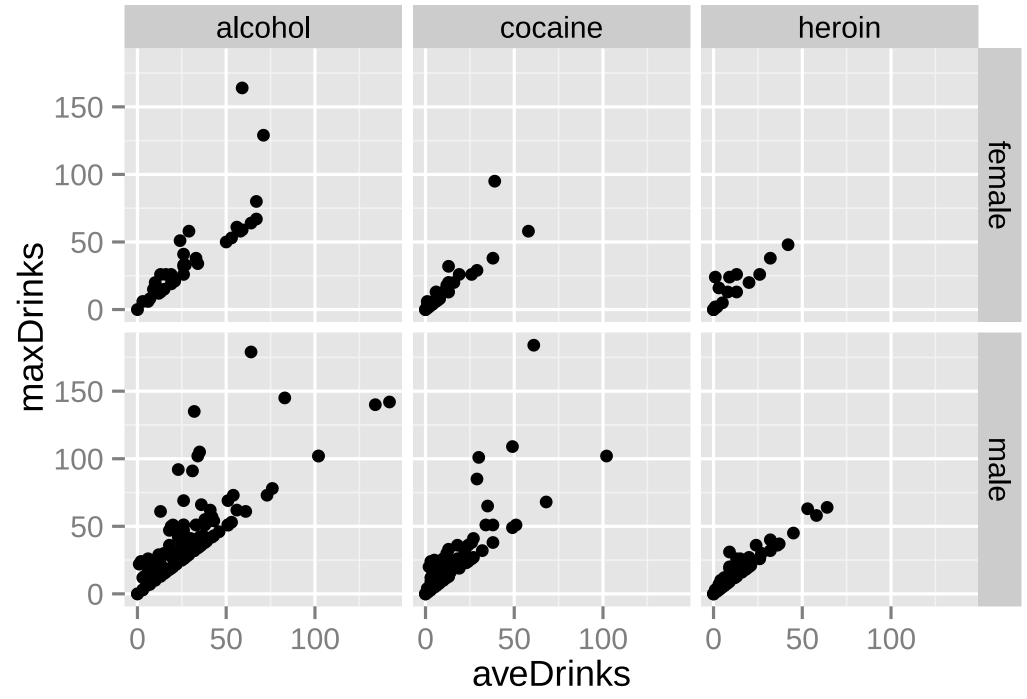
Sometimes overplotting can obscure patterns rather than reveal them.

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

## Error:  binding not found:  'i1'

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



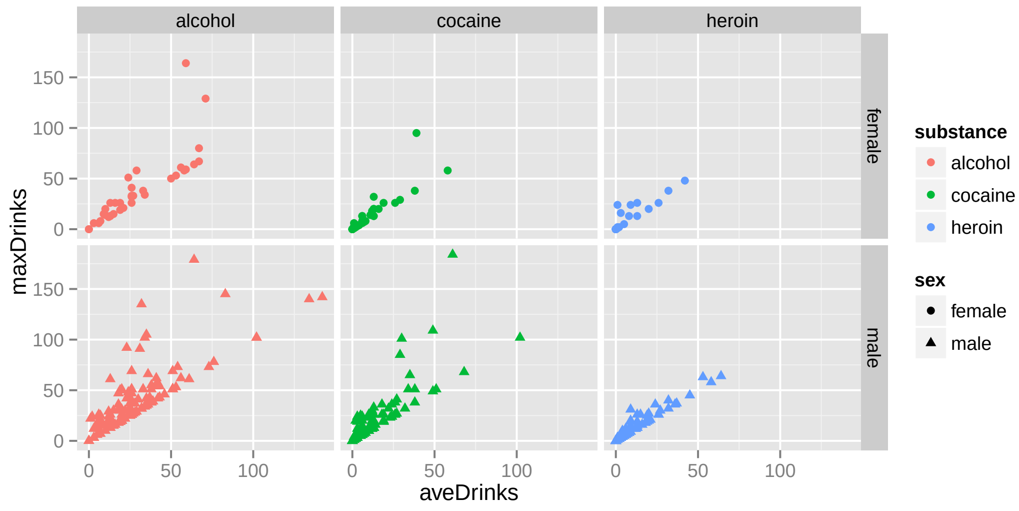
Faceting creates separate sub-plots for each subset. Faceting is described with a formula and works the same way whether we create our plot with `ggplot()` or with `qplot()`.

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



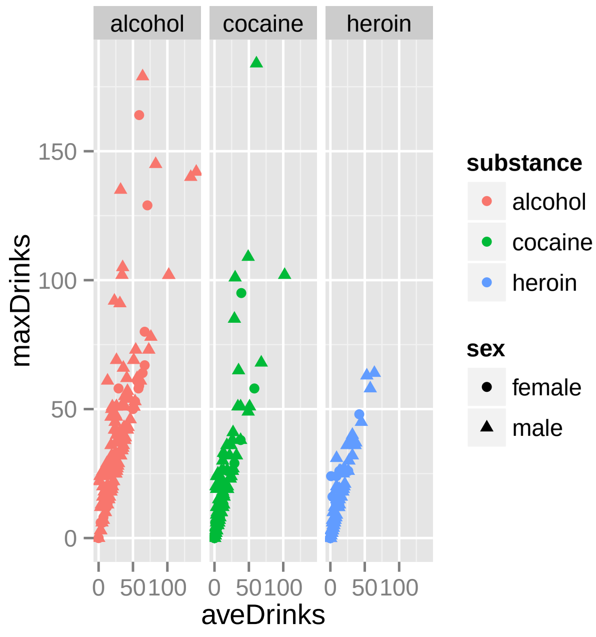
Redundant coding is sometimes useful even when faceting.

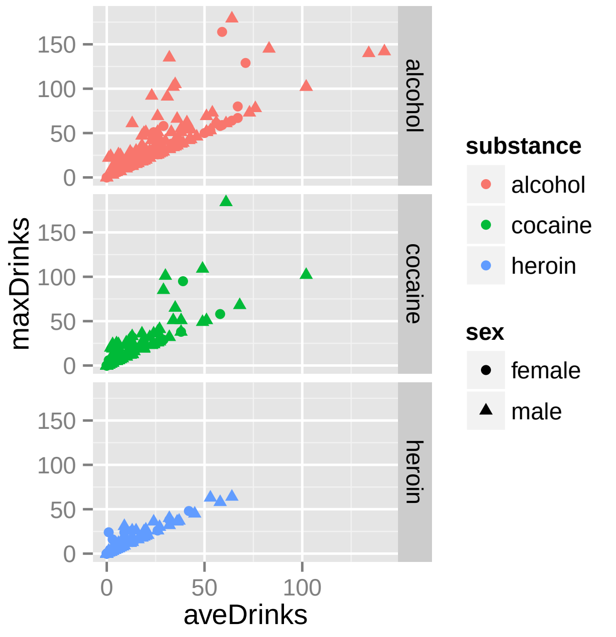
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid(sex ~ substance)
```



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

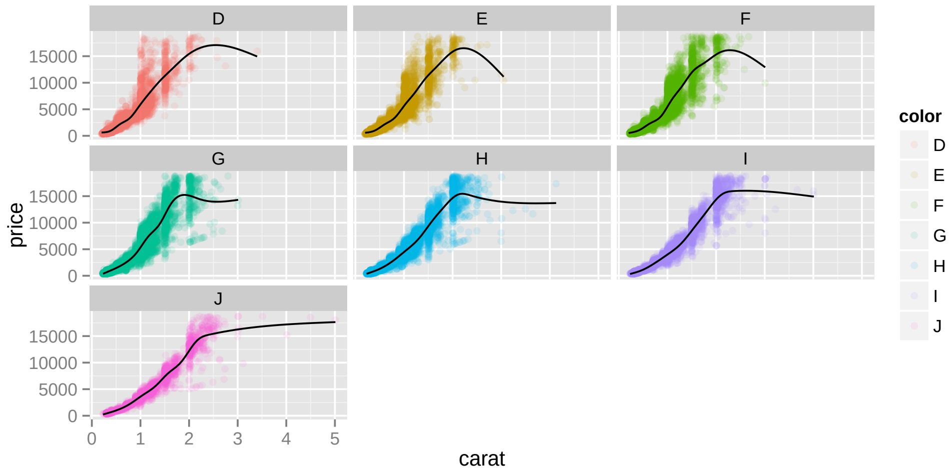
```
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid(. ~ substance)
qplot( aveDrinks, maxDrinks, data=HELPrct, color=substance, shape=sex) +
  facet_grid( substance ~ . )
```





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

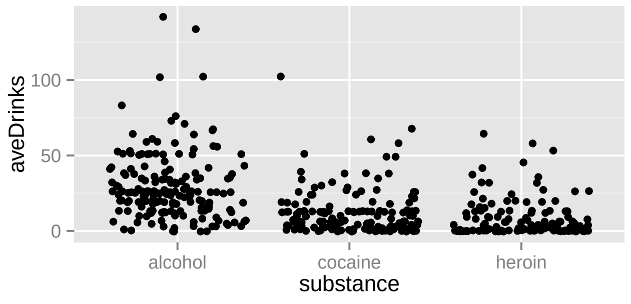
```
ggplot( data=diamonds, aes(x=carat, y=price, color=color) ) +
  geom_point( alpha=0.1 ) +
  geom_smooth( se=FALSE, color="black" ) +
  facet_wrap( ~ color)
```

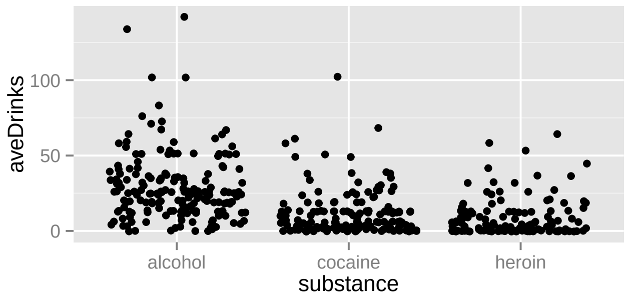


6 Position – jitter, stack, and dodge

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

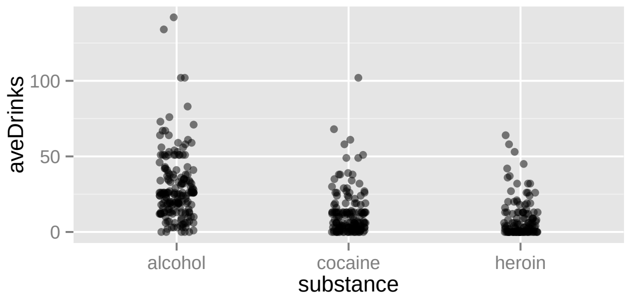
```
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_jitter()
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_point(position="jitter")
```





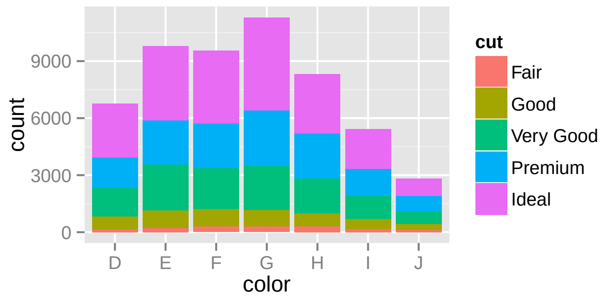
Jittering adds a bit of random noise to the data to help avoid collisions. We can exert more precise control over the jittering if we use `position_jitter()`. Since no information is lost when jittering (slightly) a categorical variable, it is often a good idea to jitter only in one direction when plotting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

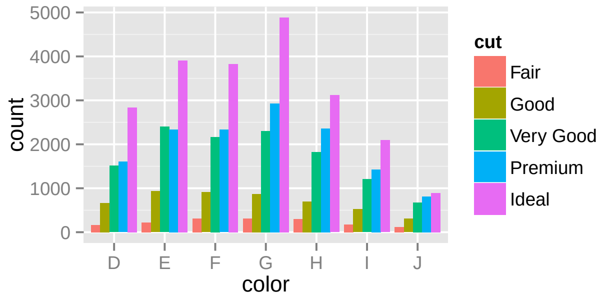
```
ggplot( data=HELPrct, aes(x=substance, y=aveDrinks)) +
  geom_point(position=position_jitter(width=0.1, height=0), alpha=0.5)
```



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")
```

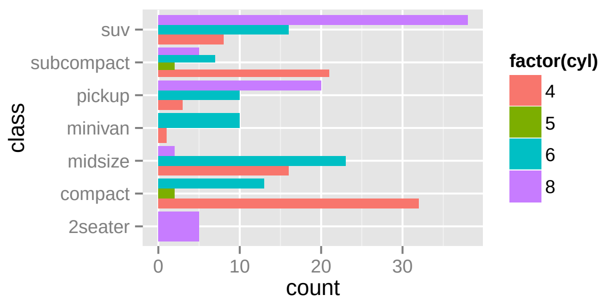


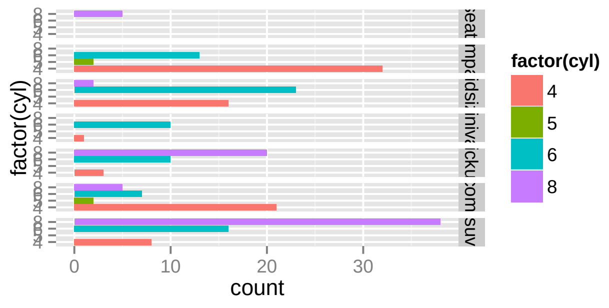


Dodging often yields results that are similar to faceting, but the labeling is different, and the results can be less than pleasing if some of the discrete categories are unpopulated.

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

```r
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

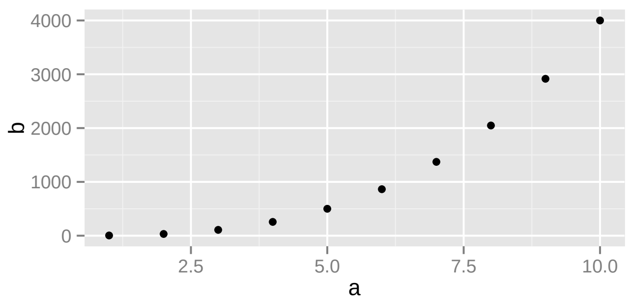
- `trans` a transformation to apply to the variable (e.g., `"log10"`)
- `breaks` manually set the break points for the axes.
- `label` the name of a emphfunction for processing the values appearing on the axes. Use `format=dollar` to format as US currency (`dollar()` is in the `scales` package).

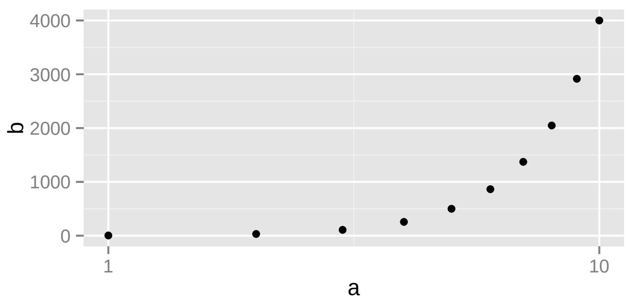
[8]This is a bit of an oversimplification. Actually, the aesthetics get computed twice, once before the stat and again after. For a histogram, for example, we need to look at the aesthetics to figure out which variable to bin (that's the stats job), but it isn't until after the binning that we will know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

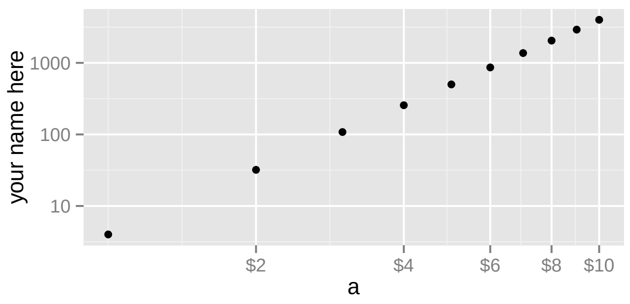
- name the name displayed on the guide. This can be useful if the variable name is not appropriate for this purpose.
- limits for limiting the data used to create the plot.

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
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")
```

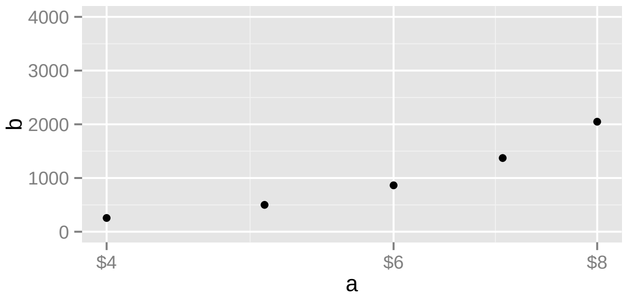






The limits argument of a position scale can be used to limit the *data* displayed.

```
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).
```

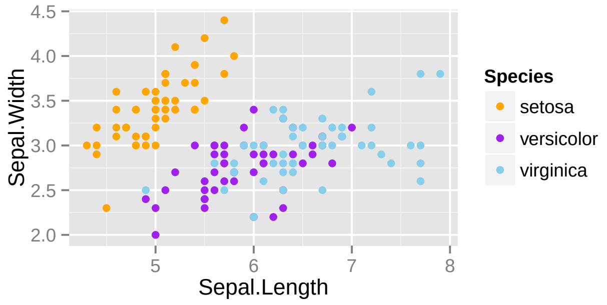


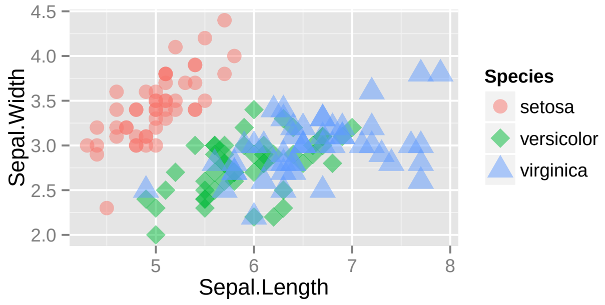
We will see another way to change the viewing window of a plot in Section 8.

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.

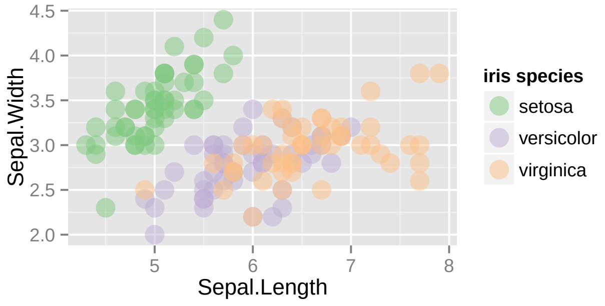
```
p <- ggplot( data=iris, aes(Sepal.Length, Sepal.Width) )
p +
  geom_point(aes(color=Species)) +
  scale_color_manual(
    values=c(setosa="orange", versicolor="purple", virginica="skyblue"))
p +
  geom_point(aes(shape=Species, color=Species), size=5, alpha=.5) +
  scale_shape_manual(values=c(20,18,17))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





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()`:

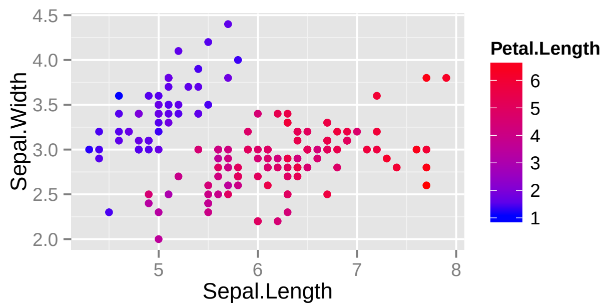
```
apropos("^scale_")
```

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

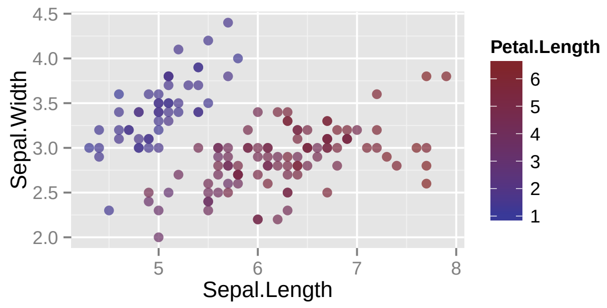
Continuous variables cannot be mapped to shape, but they can be mapped to color, in which case a spectrum of colors is used.

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



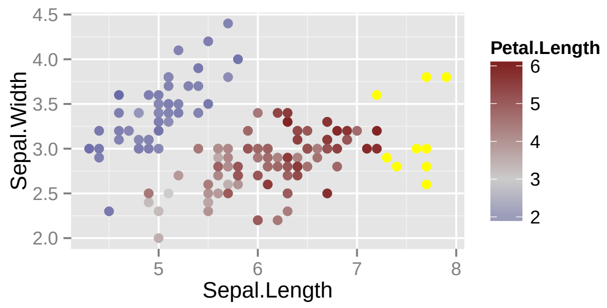
Those colors are bit hard to look at. Let's mute them a bit

```
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.

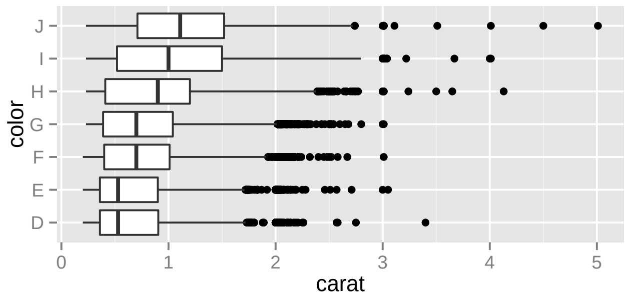
8 Coordinate systems

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.

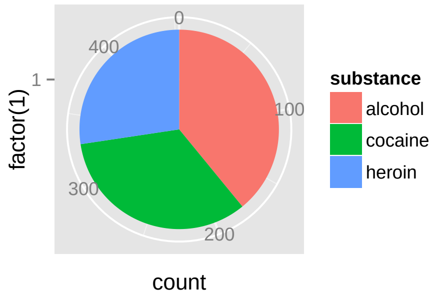
```
ggplot(diamonds, aes(x=color, y=carat)) +
  geom_boxplot() +
  coord_flip()
```



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.

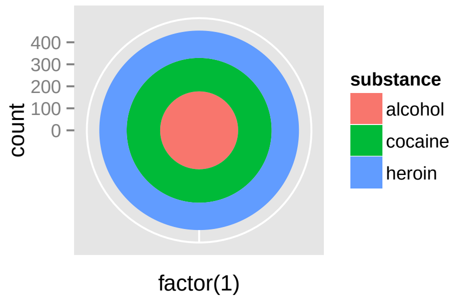
```
ggplot( data=HELPrct, aes(x=factor(1), fill=substance)) +
  geom_bar(width=1) +    # avoid space between "bars"
  coord_polar(theta="y")
```



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()
```



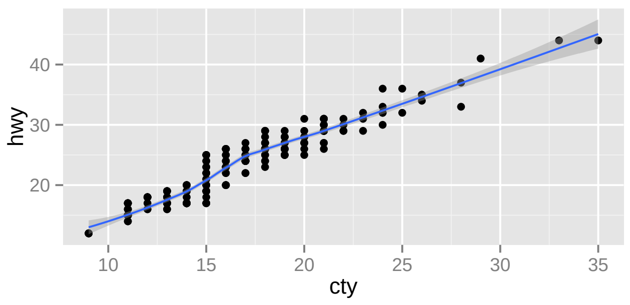
8.3 `coord_map()`

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.

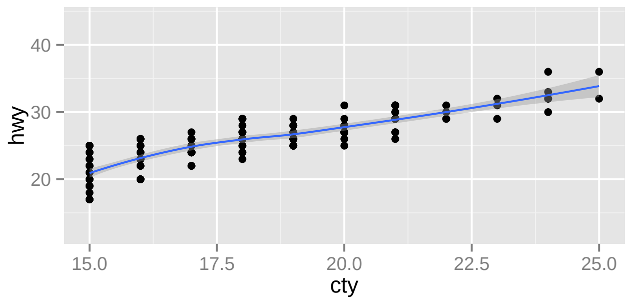
8.4 Using coords to zoom

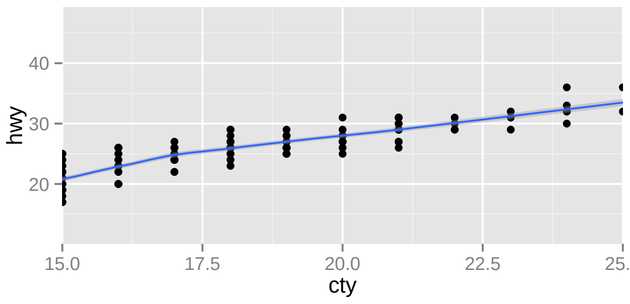
One additional use for a coordinate system is to zoom in on a portion of a plot using `xlim` and `ylim`. Setting limits in a scale removes all data outside those limits from the data frame used to generate the plot; setting the limits in a coord merely restricts the view – all the data are still used. 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))
```



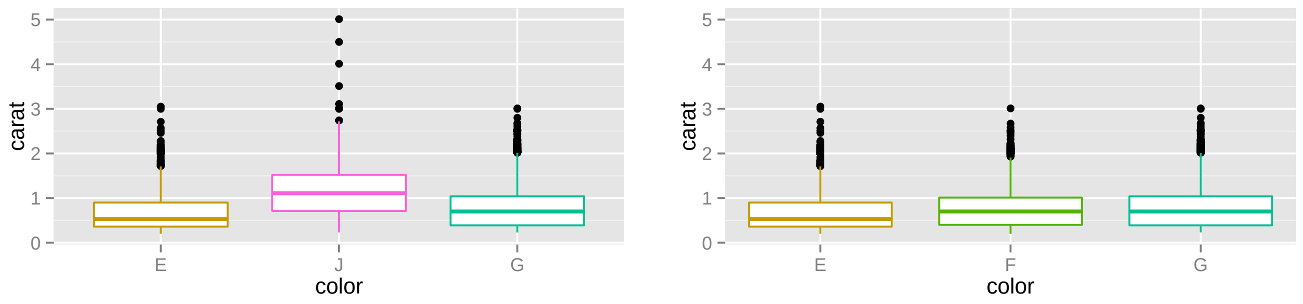


When working with discrete data, the limits are set as numerical values in the coord system but as natural values in the scale (and need not be contiguous or in the original order). There's really no reason for a legend on these plots, so let's turn it off.[9]

```
p <- ggplot(diamonds, aes(x=color, y=carat, color=color)) +
  geom_boxplot()
p +
  scale_x_discrete(limits=c("E","J", "G")) +
```

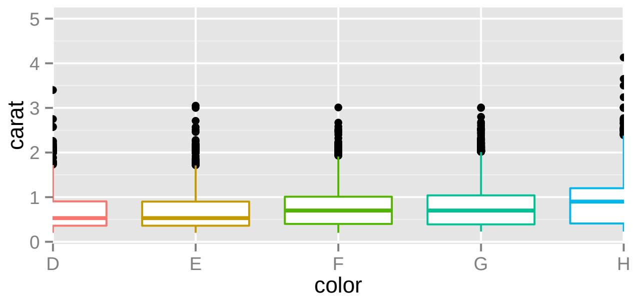
[9]We'll have more to say about `theme()` later. Note that `theme()` replaces `opt()` from older versions of `ggplot2`.

```
  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.

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

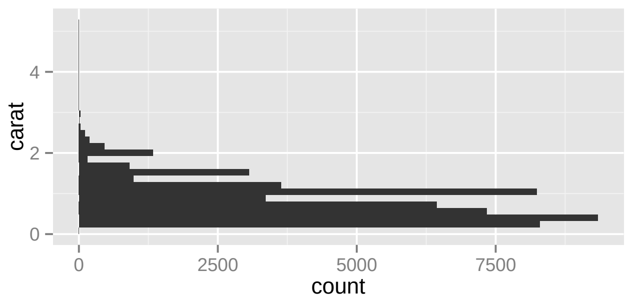


9 Some shortcuts

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()
```



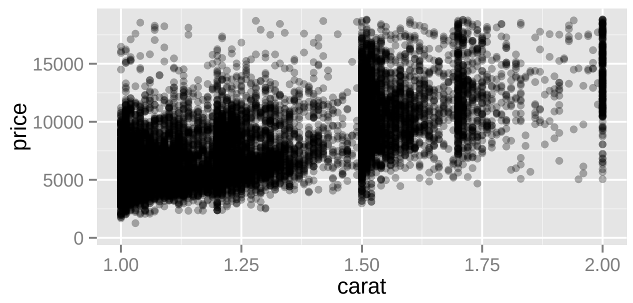
`qplot()` also provides quick access to some common plot modifications:

- `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.

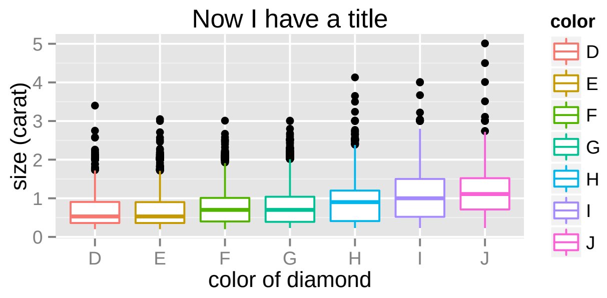
```
qplot( carat, price, data=diamonds, alpha=I(0.3))  + xlim(c(1,2))
```



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

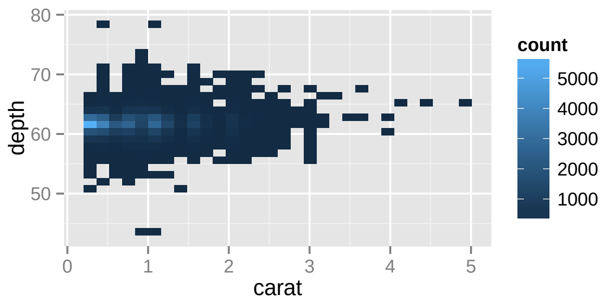
A similarly simple solution is to use transparency so that where there is more data, the plot becomes darker.

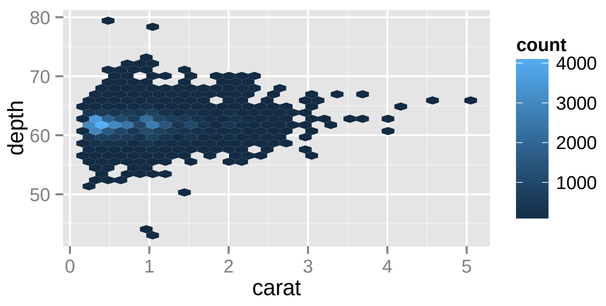
10.3 Use plots that rely on data reduction

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.

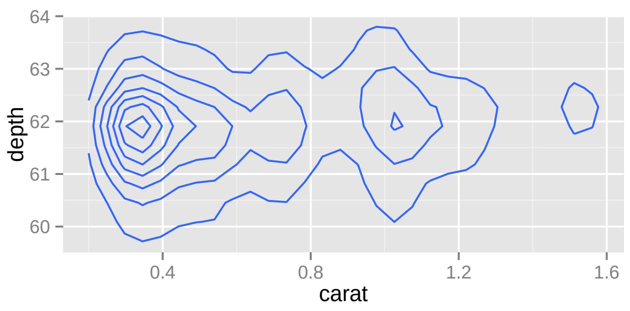
```
p <- ggplot( diamonds, aes(x = carat, y=depth) )
p + stat_bin2d()
require(hexbin)         # You may need to install this package from source

## Loading required package:  hexbin

p + stat_binhex()
```



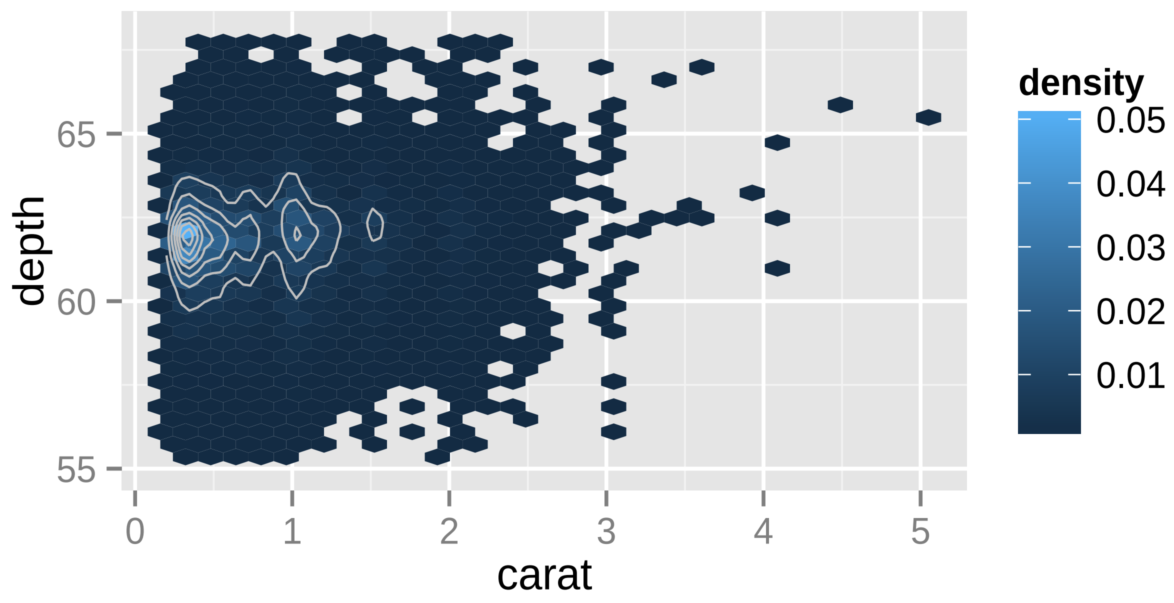


Alternatively, we can plot the contours of a 2d kernel density estimate.

```
p + stat_density2d()
```



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

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


depth
65
60
55
0
1
2
3
4
5
carat
density
0.05
0.04
0.03
0.02
0.01