

Math 241: Engineering Statistics

S DeRuiter (based on original by R Pruim)

Spring 2015

Contents

0	Where do numbers come from?	3
1	Graphical Summaries of Data	5
1.1	Getting Started With RStudio	5
1.2	Data in R	8
1.3	Graphing the Distribution of One Variable	11
1.4	Looking at Multiple Variables at Once	14
1.5	Exporting Plots	18
1.6	Reproducible Research	18
1.7	Customizing Graphics: A Few Bells and Whistles	21
1.8	Getting Help in RStudio	27
1.9	Graphical Summaries – Important Ideas	28
2	Numerical Summaries	33
2.1	Tabulating Data	33
2.2	Working with Pre-Tabulated Data	37
2.3	Summarizing Distributions of Quantitative Variables	38
2.4	Measures of Center	38
2.5	Measures of Spread	39
2.6	Summarizing Categorical Variables	44
2.7	Relationships Between Two Variables	44

0

Where do numbers come from?

Scientists and engineers work with numbers constantly. Physical constants, values for the specific heat index or measures of strength or flexibility of some material, resistance of some component in an electrical device, etc., etc.

Most of these numbers come from some process that generated data, often leading to a calculation that produced the number.

Thought experiment – How many coins?

Here's a thought experiment for you. Suppose a middle school class has collected a large number of coins in a sack. Before bringing the money to the bank, they would like to estimate how many coins they have (using tools and methods that 6th graders have at their disposal). You've been brought in to consult with them about how they should do this.

1. What method would you suggest? Why?
2. What other methods would be possible? What makes your proposed method better?
3. For your favorite method and others, identify factors that lead the resulting estimate to be different from the exact number of coins in the sack.

Some important terms

estimand/measureand The number we want to know. The “truth.” In our example this is the number of coins in the bag. Typically this will be a number that describes some process or population, and typically it will be impossible to know the value exactly.

estimate/measurement The value calculated from our data. This may be as simple as recording a value reported by some device, or it may involve recording multiple values, perhaps of multiple variables, maybe at multiple times, and making some computations with that data.

error The difference between the estimate and the estimand. Because we don't know the estimand exactly, we can't know the error exactly either. But thinking about what the error could be is a big part of understanding the statistical properties of an estimation method. Generally, we want methods where errors tend to be small (so our estimate is “likely to be close to the estimand”) and centered around 0 (so we're “right on average”).

systematic (component of) error a component of error that makes our estimate biased – in other words, leads the estimate to be either an over- or under-estimate. For example, neglecting the weight of the sack would lead us to overestimate the weight of the coins, and therefore overestimate the number of coins. Another way to express this idea is “a tendency to be off in a certain direction.”

random (component of) error a component of error that leads to variability in estimates (but not a particular tendency toward over- or under-estimation). If random errors are larger, there will be more variability in estimates, so we will be less confident that the estimand and estimate are close together – although some estimates may still be very close to the estimand, just by chance.

One of the big questions in statistics is this: *What does our estimate tell us about the estimand?* We will eventually learn techniques for quantifying (and attempting to reduce) the effects of error in our measurements.

1

Graphical Summaries of Data

1.1 Getting Started With RStudio

RStudio is an integrated development environment (IDE) for R, a freely available language and environment for statistical computing and graphics. Both R and RStudio are freely available for Mac, PC, and Linux.

We have set up an RStudio server on campus, which allows you to run R in a web browser on any computer without installing the software yourself. Your session is restored each time you log in, so you can work on multiple computers without losing your work when you move from one to the other. The RStudio server is the recommended interface for using R and RStudio for this course. You can access the RStudio server via a web browser. (For best results, avoid Internet Explorer.)

If you prefer to install R and RStudio directly on your own computer, you can get R at <http://cran.r-project.org> and RStudio at <http://rstudio.org/>.

To access the Calvin RStudio server, use the links from our course Moodle site, or connect directly at <http://rstudio.calvin.edu:8787>.

1.1.1 Logging in

When you navigate to the RStudio server, you will be prompted to login. Your login is your Calvin Gmail address, and your password is the corresponding password. Once you are logged in, you will see something like Figure 1.1.

1.1.2 Using R as a calculator

Notice that RStudio divides its world into four panels. Several of the panels are further subdivided into multiple tabs. The **Console** panel is where we type commands that R will execute.

R can be used as a calculator. Try typing the following commands in the console panel.

```
5 + 3

## [1] 8

15.3 * 23.4
```

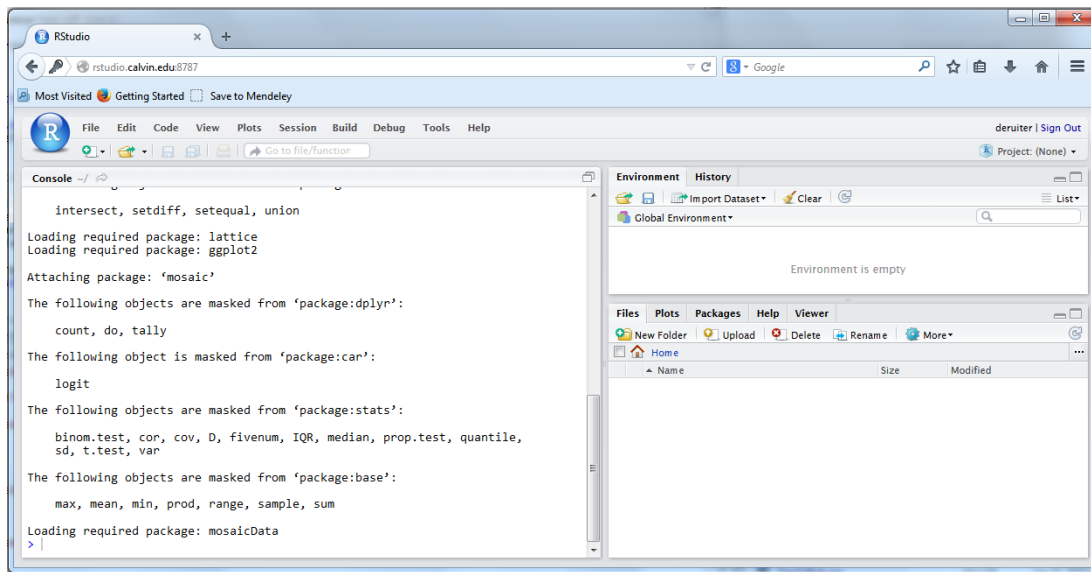


Figure 1.1: Welcome to RStudio.

```
## [1] 358.02
```

```
sqr(16)
```

```
## [1] 4
```

You can save values to named variables for later reuse

```
product = 15.3 * 23.4      # save result
product                    # show the result
```

```
## [1] 358.02
```

```
product <- 15.3 * 23.4     # <- is assignment operator, same as =
product
```

```
## [1] 358.02
```

```
15.3 * 23.4 -> newproduct  # -> assigns to the right
newproduct
```

```
## [1] 358.02
```

```
.5 * product              # half of the product
```

```
## [1] 179.01
```

```
log(product)              # (natural) log of the product
```

```
## [1] 5.880589

log10(product)           # base 10 log of the product

## [1] 2.553907

log(product,base=2)      # base 2 log of the product

## [1] 8.483896
```

The semi-colon can be used to place multiple commands on one line. One frequent use of the semi-colon is to save and print a value all in one line of code:

```
15.3 * 23.4 -> product; product    # save result and show it

## [1] 358.02
```

1.1.3 Loading packages

R is divided up into packages. You can think of the packages as software toolkits designed to do particular jobs. A few of these, known as “base R”, are loaded every time you run R, but most have to be selected. This way you only have as much of R as you need. There are two steps to follow before you can use a package in R:

1. Install the package. This operation downloads the relevant files to your computer, and lets R know where they are located. It does *not* give the current R session permission to use the tools contained in the package! The packages you will need for work in this course have already been installed on the Calvin RStudio server. For this course, you will probably not need to install any packages yourself, unless you are using a local copy of R and RStudio installed on your own computer. If you need to install packages, an easy way to do it is to use the **Packages** tab in the lower right panel of RStudio. Just click on **Install** (upper left corner of the **Packages** tab) and then type the name of the package.
2. Load the package. This operation gives the current R session permission to access and use the tools contained in the package. Even if you are using the RStudio server, you will often need to load required packages at the beginning of each R session. There are several ways to load packages, as detailed below.

In the Packages tab, check the boxes next to the following packages to load them:

- **mosaic** (a package from Project MOSAIC; this should already be loaded)
- **DAAG** (a package that goes with the book *Data Analysis and Graphics*; probably not loaded, check the box to load it.)

You can also load packages by typing, for example

```
require(DAAG) # loads the DAAG package if it is not already loaded
```

1.1.4 Four Things to Know About R

1. R is case-sensitive

If you mis-capitalize something in R, it won't do what you want.

2. Functions in R use the following syntax:

```
functionname(argument1, argument2, ...)
```

- The arguments are always *surrounded by (round) parentheses* and *separated by commas*. Some functions (like `data()`) have no required arguments, but you still need the parentheses.
- If you type a function name without the parentheses, you will see the *code* for that function printed out to the console window – which probably isn't what you want at this point.

3. TAB completion and arrows can improve typing speed and accuracy.

If you begin typing a command and hit the TAB key, R will show you a list of possible ways to complete the command. If you hit TAB after the opening parenthesis of a function, it will show you the list of arguments it expects. The up and down arrows can be used to retrieve past commands.

4. Hit ESCAPE to break out of a mess.

If you get into some sort of mess typing (usually indicated by extra '+' signs along the left edge, indicating that R is waiting for more input – perhaps because you have some sort of error in what has gone before), you can hit the escape key to get back to a clean prompt.

1.2 Data in R

1.2.1 Data Frames

Most often, data sets in R are stored in a structure called a **data frame**. A data frame is designed to hold “rectangular data”. The people or things being measured or observed are called **observational units** (or subjects or cases when they are people). For measurements collected over time, the observational units would be the individual time-points at which data points were collected. Each observational unit is represented by one row in the data frame. The different pieces of information recorded for each observational unit are stored in separate columns, called **variables**.

1.2.2 Data in Packages

There are a number of data sets built into R and many more that come in various add-on packages.

You can see a list of data sets in a particular package like this:

```
data(mosaic)
```

You can find a longer list of all data sets available in any loaded package using

```
data()
```


1.2.3 The HELPrct data set

The `HELPrct` data frame from the `mosaic` package contains data from the Health Evaluation and Linkage to Primary Care randomized clinical trial. You can find out more about the study and the data in this data frame by typing

```
?HELPrct
```

Among other things, this will tell us something about the subjects (observational units) in this study:

Eligible subjects were adults, who spoke Spanish or English, reported alcohol, heroin or cocaine as their first or second drug of choice, resided in proximity to the primary care clinic to which they would be referred or were homeless. Patients with established primary care relationships they planned to continue, significant dementia, specific plans to leave the Boston area that would prevent research participation, failure to provide contact information for tracking purposes, or pregnancy were excluded.

Subjects were interviewed at baseline during their detoxification stay and follow-up interviews were undertaken every 6 months for 2 years.

It is often handy to look at the first few rows of a data frame. It will show you the names of the variables and the kind of data in them:

```
head(HELPrct)
```

```
##   age anysubststatus  anysub  cesd  d1  daysanysub  dayslink  drugrisk  e2b  female    sex  g1b
## 1  37                1    yes   49   3        177       225         0  NA      0    male  yes
## 2  37                1    yes   30  22         2        NA         0  NA      0    male  yes
## 3  26                1    yes   39   0         3       365        20  NA      0    male  no
## 4  39                1    yes   15   2       189       343         0   1      1 female  no
## 5  32                1    yes   39  12         2        57         0   1      0    male  no
## 6  47                1    yes    6   1        31       365         0  NA      1 female  no
##   homeless i1 i2 id indtot linkstatus link      mcs      pcs pss_fr racegrp satreat
## 1   housed 13 26  1    39          1  yes 25.111990 58.41369      0   black     no
## 2 homeless 56 62  2    43          NA <NA> 26.670307 36.03694      1   white     no
## 3   housed  0  0  3    41          0  no  6.762923 74.80633     13   black     no
## 4   housed  5  5  4    28          0  no 43.967880 61.93168     11   white    yes
## 5 homeless 10 13  5    38          1  yes 21.675755 37.34558     10   black     no
## 6   housed  4  4  6    29          0  no 55.508991 46.47521      5   black     no
##   sexrisk substance treat  binnedAge
## 1      4   cocaine   yes   [30,40)
## 2      7  alcohol   yes   [30,40)
## 3      2   heroin   no    [20,30)
## 4      4   heroin   no    [30,40)
## 5      6  cocaine   no    [30,40)
## 6      5  cocaine   yes   [40,50)
```

```
dim(HELPrct)
```

```
## [1] 453  28
```

The commands and R output above tell us that there are 453 observational units in this data set and 28 variables. That's plenty of variables to get us started with exploration of data.

1.2.4 The KidsFeet data set

Here is another data set in the `mosaic` package:

```
head(KidsFeet)

##      name birthmonth birthyear length width sex biggerfoot domhand
## 1 David          5        88   24.4   8.4  B           L          R
## 2 Lars          10        87   25.4   8.8  B           L          L
## 3 Zach          12        87   24.5   9.7  B           R          R
## 4 Josh           1        88   25.2   9.8  B           L          R
## 5 Lang           2        88   25.1   8.9  B           L          R
## 6 Scotty         3        88   25.7   9.7  B           R          R
```

1.2.5 The oldfaith data set

A final example data set comes from the `alr3` package. This package is probably not loaded (unless you already loaded it). You can load it from the **Packages** tab or by typing the command

```
require(alr3)
```

Once you have done that, you will have access to the data set containing information about eruptions of Old Faithful, a geyser in Yellowstone National Park.

```
head(oldfaith)

##      Duration Interval
## 1         216        79
## 2         108        54
## 3         200        74
## 4         137        62
## 5         272        85
## 6         173        55
```

If you want to know the size of your data set, you can ask it how many rows and columns it has with `nrow()`, `ncol()`, or `dim()`:

```
nrow(oldfaith)

## [1] 270

ncol(oldfaith)

## [1] 2

dim(oldfaith)

## [1] 270  2
```

In this case we have 270 observations of each of two variables. In a data frame, the observational units are always in the rows and the variables are always in the columns. If you create data for use in R (or most other statistical packages), you need to make sure your data are also in this shape.

1.2.6 Using your own data

We will postpone for now a discussion about getting your own data into RStudio, but any data you can get into a reasonable format (like csv) can be imported into RStudio pretty easily.

1.3 Graphing the Distribution of One Variable

A **distribution** tells which values a variable takes on, and with what frequency. That is, the distribution answers two questions:

- What values?
- How often?

Several standard statistical graphs can help us see distributions visually.

The general syntax for making a graph or numerical summary of one variable in a data frame is

```
plotname(~variable, data = dataName)
```

In other words, there are three pieces of information we must provide to R in order to get the plot we want:

- The kind of plot (`histogram()`, `bargraph()`, `densityplot()`, `bwplot()`, etc.)
- The name of the variable
- The name of the data frame this variable is a part of.

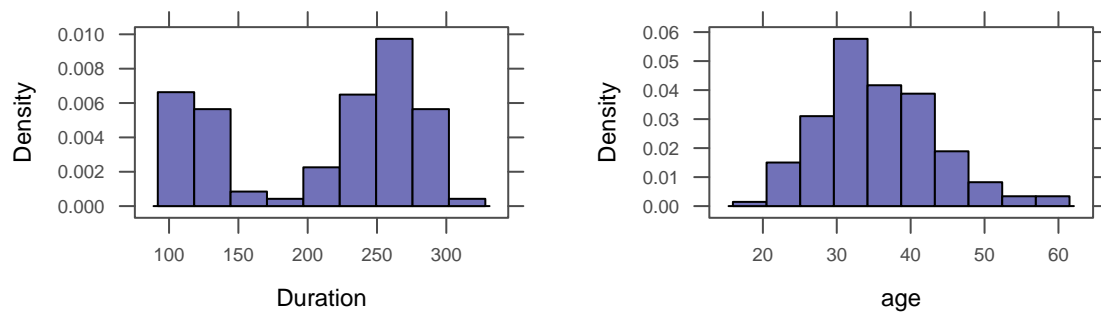
Note: The same syntax works for numerical summaries as well – thanks to the `mosaic` package we can apply the same syntax for `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, etc. Later we will use this syntax again to fit linear and nonlinear models to data.

1.3.1 Histograms (and density plots) for quantitative variables

Histograms are a way of displaying the distribution of a quantitative variable.

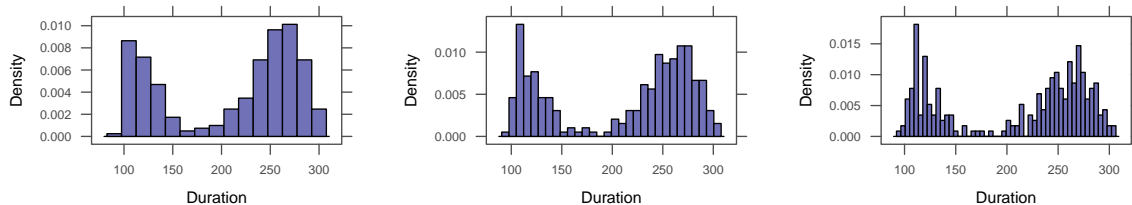
Here are a couple examples:

```
histogram(~Duration, data = oldfaith)
histogram(~age, data = HELPrct)
```



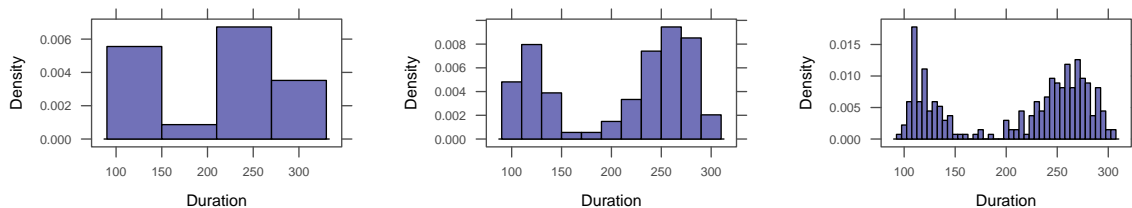
We can control the (approximate) number of bins using the `nint` argument, which may be abbreviated as `n`. The number of bins (and to a lesser extent the positions of the bins) can make a histogram look quite different.

```
histogram(~Duration, data = oldfaith, n = 15)
histogram(~Duration, data = oldfaith, n = 30)
histogram(~Duration, data = oldfaith, n = 50)
```



The `histogram()`¹ function in the `mosaic` package lets you describe the bins in terms of center and width instead of in terms of the number of bins. This is especially nice for count data.

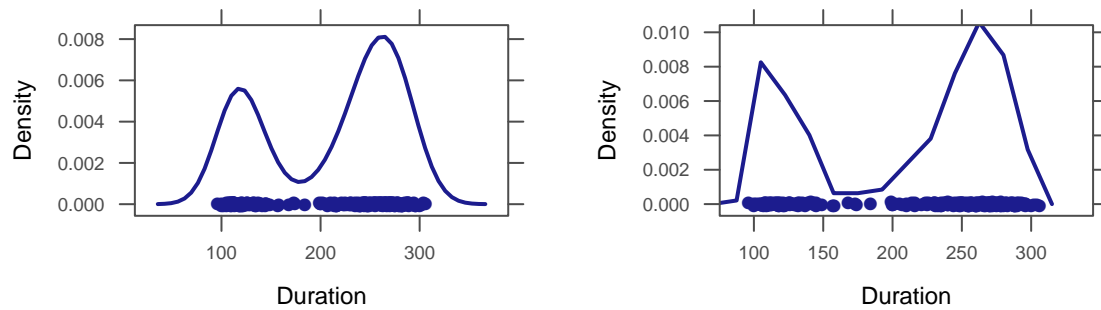
```
histogram(~Duration, data = oldfaith, width = 60)
histogram(~Duration, data = oldfaith, width = 20)
histogram(~Duration, data = oldfaith, width = 5)
```



R also provides a “smooth” version called a density plot and a triangular version called a frequency polygon; just change the function name from `histogram()` to `densityplot()` or `freqpolygon()`.

```
densityplot(~Duration, data = oldfaith)
freqpolygon(~Duration, data = oldfaith)
```

¹The `mosaic` version of the `histogram()` function has some extra features (like the `width` argument) but is otherwise is very similar to regular `histogram()` function in `lattice`.



1.3.2 The shape of a distribution

If we make a histogram of our data, we can describe the overall shape of the distribution. Keep in mind that the shape of a particular histogram may depend on the choice of bins. Choosing too many or too few bins can hide the true shape of the distribution. (When in doubt, compare several histograms with different bin settings before you decide which one provides the most informative summary of the data.)

Here are some words we use to describe shapes of distributions.

symmetric The left and right sides are mirror images of each other.

skewed The distribution stretches out farther in one direction than in the other. (We say the distribution is skewed toward the long tail. So right-skewed (also known as positive-skewed) data have a “fat right tail” – more observations of larger values than of small ones.)

uniform The heights of all the bars are (roughly) the same. (So the data are equally likely to be anywhere within some range.)

unimodal There is one major “bump” where there is a lot of data.

bimodal There are two “bumps”.

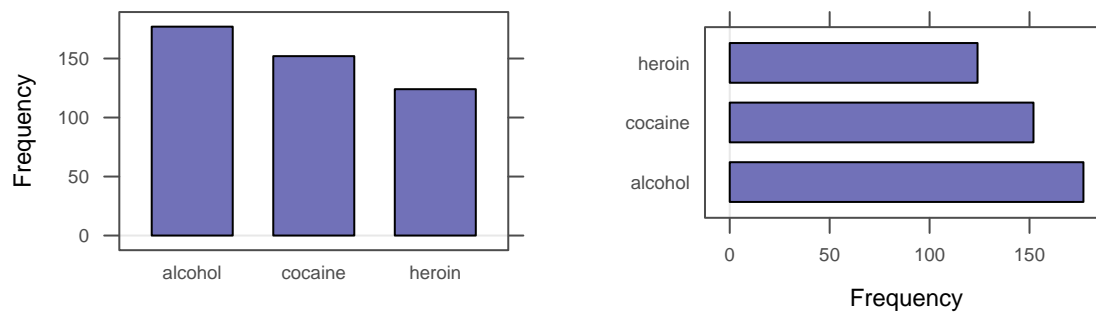
outlier An observation that does not fit the overall pattern of the rest of the data.

We’ll learn about another graph used for quantitative variables (a boxplot, `bwplot()` in R) soon.

1.3.3 Bar graphs for categorical variables

Bar graphs are a way of displaying the distribution of a categorical variable.

```
bargraph(~substance, data = HELPrct)
bargraph(~substance, data = HELPrct, horizontal = TRUE)
```



A side note: we will be unlikely to use pie charts in this course. Many data analysts argue that pie charts are difficult to read and interpret, and often use space ineffectively, especially if they are divided into more than two slices. Unless you are *sure* there is a good reason to use one, don't.

1.4 Looking at Multiple Variables at Once

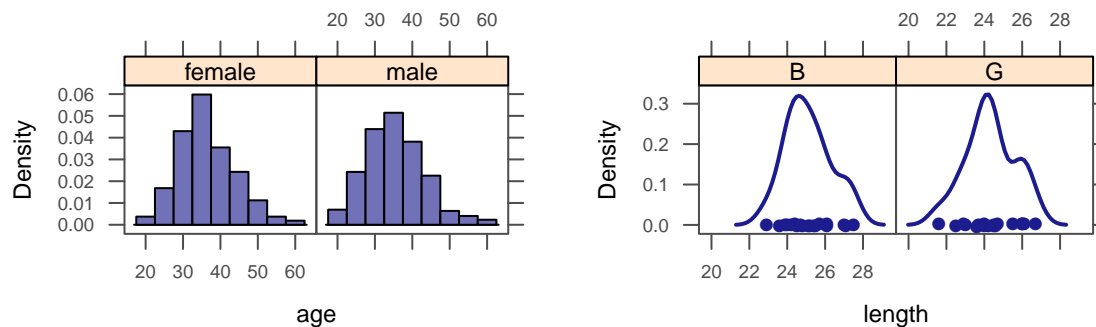
1.4.1 Conditional plots

The formula for a `lattice` plot can be extended to create multiple panels based on a “condition”, often given by another variable. The general syntax for this becomes

```
plotname(~variable | condition, data = dataName)
```

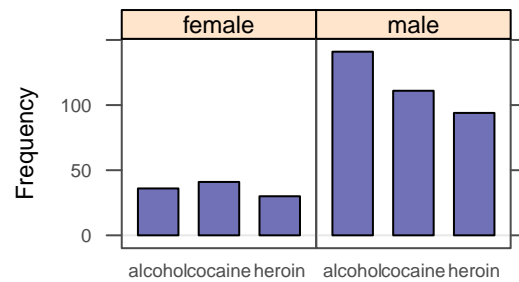
For example, we might like to see how the ages of men and women compare in the HELP study, or whether the distribution of weights of male mosquitoes is different from the distribution for females.

```
histogram(~age | sex, HELPrct, width = 5)
densityplot(~length | sex, KidsFeet)
```



We can do the same thing for bar graphs.

```
bargraph(~substance | sex, data = HELPrct)
```

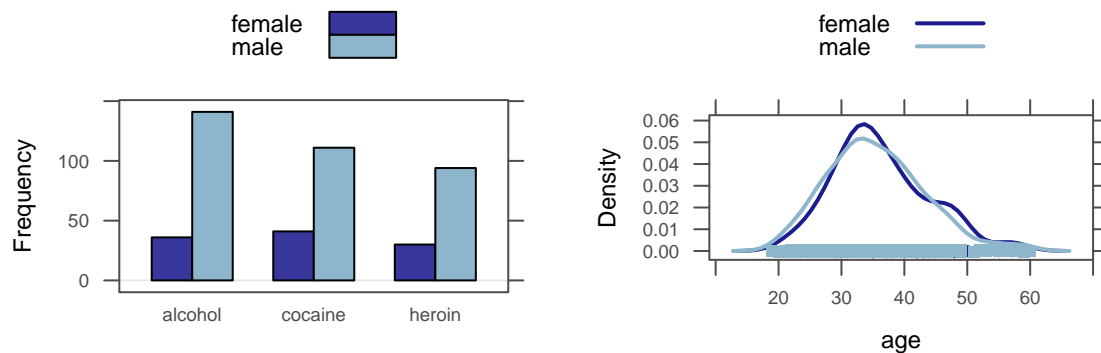


1.4.2 Grouping

Another way to look at multiple groups simultaneously is by using the `groups` argument. What `groups` does depends a bit on the type of graph, but it will put the information in one panel rather than multiple panels. Using `groups` with `histogram()` doesn't work so well because it is difficult to overlay histograms.² Density plots work better when you wish to look at the shapes of several distributions in a single plot panel.

Here are some examples. We use `auto.key=TRUE` to build a simple legend so we can tell which groups are which.

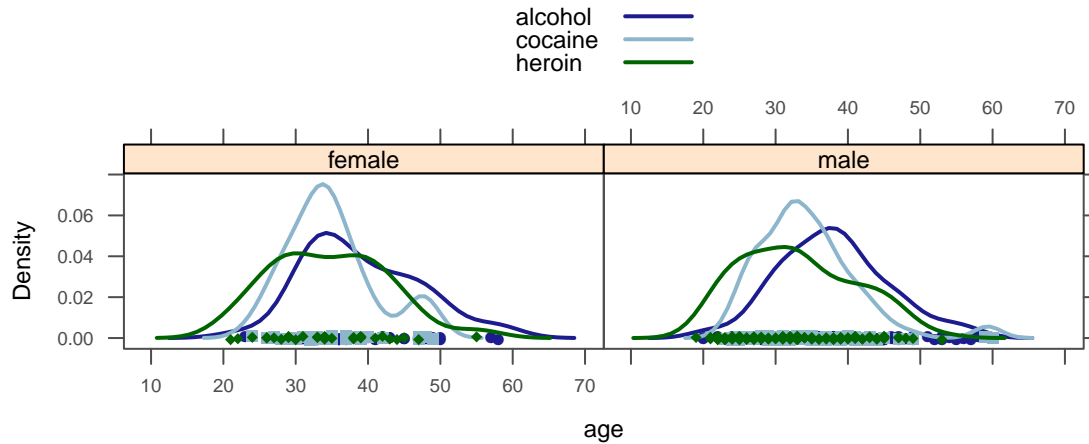
```
bargraph(~substance, groups = sex, data = HELPrct, auto.key = TRUE)
densityplot(~age, groups = sex, data = HELPrct, auto.key = TRUE)
```



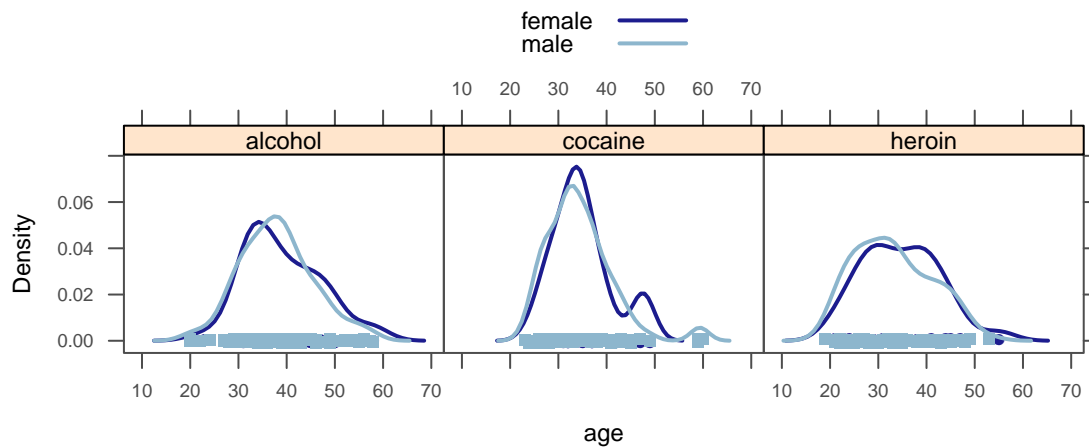
We can even combine grouping and conditioning in the same plot.

```
densityplot(~age | sex, groups = substance, data = HELPrct, auto.key = TRUE)
```

²The `mosaic` function `histogram()` does do something meaningful with `groups` in some situations.



```
densityplot(~age | substance, groups = sex, data = HELPrct, auto.key = TRUE, layout = c(3, 1))
```



This plot shows that for each substance, the age distributions of men and women are quite similar, but that the distributions differ from substance to substance.

1.4.3 Scatterplots

The most common way to look at two quantitative variables is with a scatter plot. The `lattice` function for this is `xyplot()`, and the basic syntax is

```
xyplot(yvar ~ xvar, data = dataName)
```

Notice that now we have something on both sides of the `~` since we need to tell R about two variables.

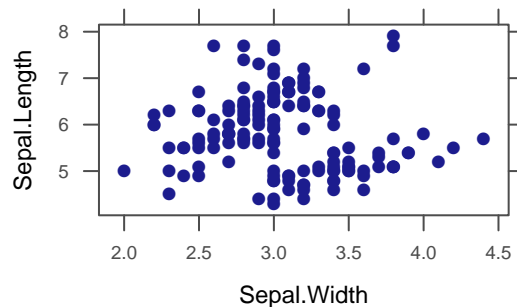
```
head(iris) # data on iris plants
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
```



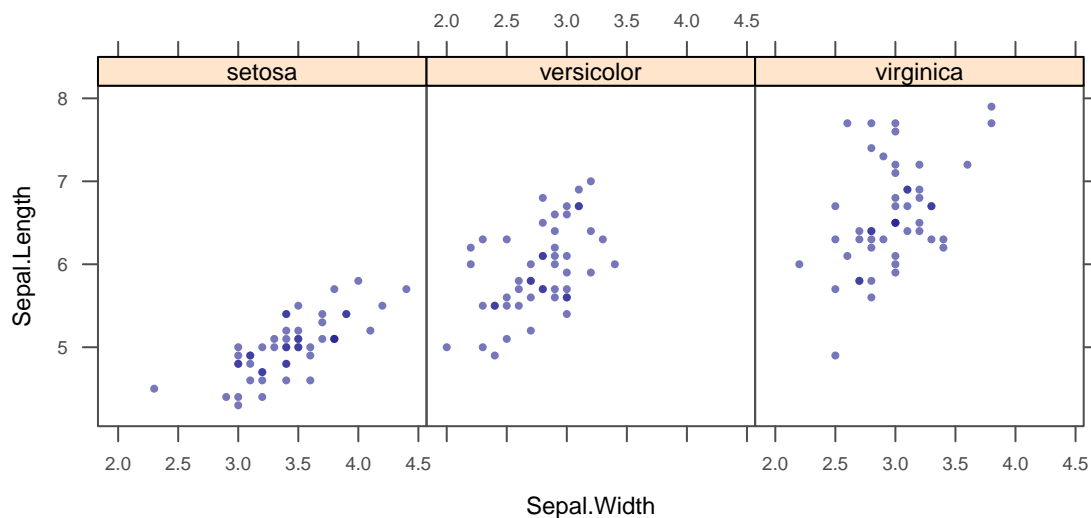
```
## 3      4.7      3.2      1.3      0.2 setosa
## 4      4.6      3.1      1.5      0.2 setosa
## 5      5.0      3.6      1.4      0.2 setosa
## 6      5.4      3.9      1.7      0.4 setosa
```

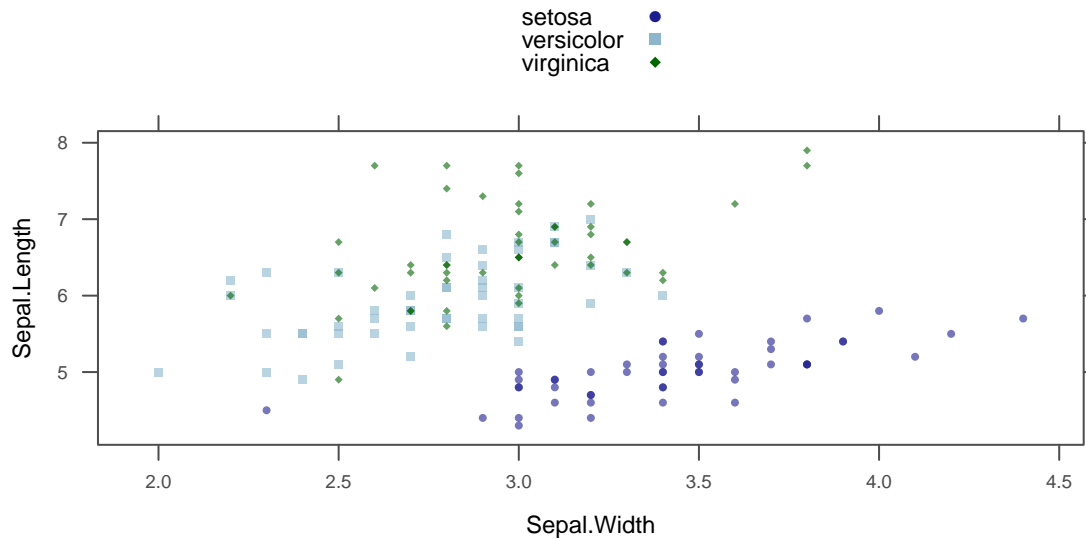
```
xyplot( Sepal.Length ~ Sepal.Width, data=iris )
```



Grouping and conditioning work just as before and can be used to see the relationship between sepal length and sepal width broken down by species of iris plant. With large data set, it can be helpful to make the dots semi-transparent so it is easier to see where there are overlaps. This is done with `alpha`. We can also make the dots smaller (or larger) using `cex`.

```
xyplot( Sepal.Length ~ Sepal.Width | Species, data=iris, alpha=.6, cex=.5 )
xyplot( Sepal.Length ~ Sepal.Width, groups = Species, data=iris, alpha=.6, cex=.5,
        auto.key=TRUE )
```





1.5 Exporting Plots

You can save plots to files or copy them to the clipboard using the **Export** menu in the **Plots** tab. It is quite simple to copy the plots to the clipboard and then paste them into a Word document, for example. You can even adjust the height and width of the plot first to get it the shape you want.

1.6 Reproducible Research

When starting to learn to use R for data analysis, it may be tempting to work by typing commands into the R console directly, or maybe by copying and pasting commands from some other source (for example, these notes, a website, etc.).

There are many reasons to avoid working this way, including:

- It is tedious, unless there is very little to type, or to copy and paste.
- It is error-prone – it's easy to copy too little or too much, or to grab the wrong thing, or to copy when you want to cut or cut when you want to copy.
- If something changes, you have to start all over.
- You have no record of what you did (unless you are an unusual person who takes detailed notes about everything you copied and pasted, or typed into the R console).

So while copy and paste seems easy and convenient at first, it is not *reproducible*. Reproducible, here, means something that can easily be repeated in exactly the same way (or with some desired modification), because the exact procedure that was followed has been clearly documented in a format that is simple to access. Reproducibility is important when projects are large, when it is important to have record of exactly what was done, or when the same analysis will be applied to multiple data sets (or a data set that is growing over time).

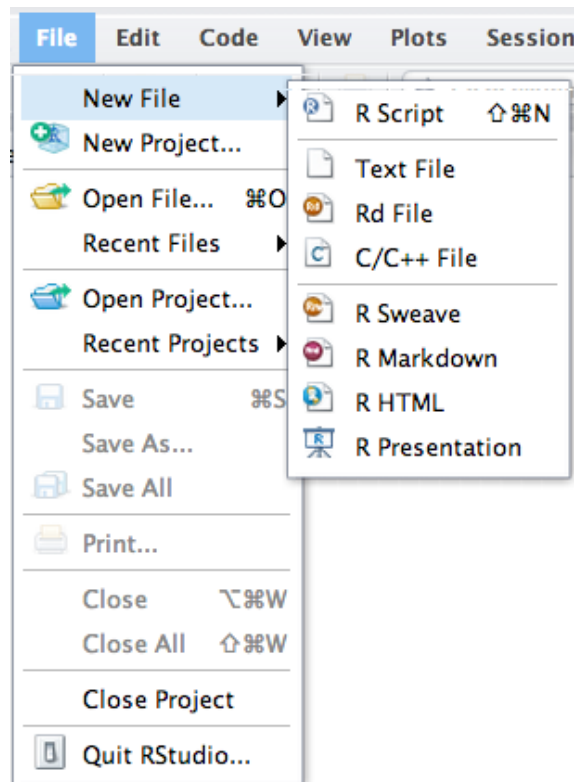
RStudio makes it easy to use techniques of reproducible research to create documents that include text, R commands, R output, and R graphics.

1.6.1 R Markdown

One simple way to do reproducible work is to use a format called R Markdown. Markdown is a simple markup language that allows for a few basic improvements on plain text (section headers, bulleted lists, numbered lists, bold, italics, etc.) R Markdown adds the ability to mix in the R stuff (R commands and output, including figures). The end product is an HTML file, so it is especially good for producing web documents.³

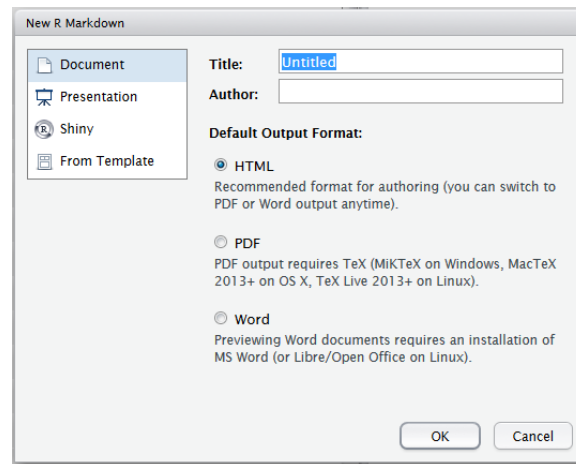
Creating a new document

To create a new R Markdown document in RStudio, go to “File”, “New File”, then “R Markdown”:



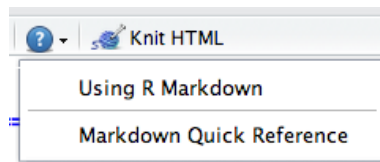
A small pop-up window will appear; for current purposes, you can select all the default options (a Document, with HTML output format, with the Title and Author blanks filled in or left blank, as you wish).

³You can actually mix in arbitrary HTML and even CSS, so if you are good at HTML, you can have quite a bit of control over how things look. Here we will focus on the basics.

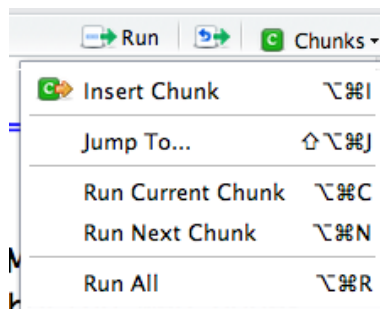


When you do this, a file editing pane will open with a template inserted. If you click on “Knit HTML”, RStudio will turn this into an HTML file and display it for you. Give it a try. You will be asked to name your file if you haven’t already done so. If you are using the RStudio server in a browser, then your file will live on the server (“in the cloud”) rather than on your computer.

If you look at the template file you will see that the file has two kinds of sections. Some of this file is just normal text (with some extra symbols to make things bold, add in headings, etc.) You can get a list of all of these mark up options by selecting the “Markdown Quick Reference” in the question mark menu (at the top of the Markdown document in the editing pane).



The second type of section is an R code chunk. These are colored differently to make them easier to see. You can insert a new code chunk by selecting “Insert Chunk” from the “Chunks” menu:



(You can also type ````{r}` to begin and ````` to end the code chunk if you would rather type.) You can put any R code in these code chunks and the results (text output or graphics) as well as the R code will be displayed in your HTML file.

In addition to knitting the document to HTML, you can do a number of other things that will make your work more efficient. In the “Chunk” menu, you can choose to run a single chunk or all the chunks. This will execute your commands in the console so you can make sure your R code is working one chunk at a time. There is also a “run” button that allows you to run just one line from within a chunk.

R Markdown files must be self-contained

R Markdown files do not have access to things you have done in your console. (This is good, else your document would change based on things not in the file.) Within each R Markdown file, you must explicitly load data, and require packages *in the R Markdown file* in order to use them. In this class, this means that most of your R Markdown files will have a chunk near the beginning that loads required packages and datasets.

Chunk options

R Markdown provides a number of chunk options that control how R code is processed. You can use them to do things like:

- run the code without displaying it (good for polished reports – your client doesn’t want to see the code)
- show the code without running it – mainly useful for demonstration purposes
- control the size and alignment of graphics

You can set default values for the chunk options and you can also override them in individual chunks. See the R Markdown help for more information about chunk options.

The default plots are often bigger than required. The following chunk options are a place to start. They can be adjusted as necessary.

```
require(knitr)
opts_chunk$set(fig.width = 5, fig.height = 2, fig.align = "center", fig.show = "hold")
```

1.6.2 knitr/latex

There is another system that produces PDFs by combining L^AT_EX and R, using the Rpackage **knitr**. This is the system used to create this document; it gives much more control over document formatting. The quality is good enough for professional publishing. If you already know L^AT_EX, it is very easy to learn. If you don’t know L^AT_EX, then you need to learn the basics of L^AT_EX to get going, which is not covered in detail here. If you are interested in learning more, consult the documentation for the **knitr** package, or the knitr website.

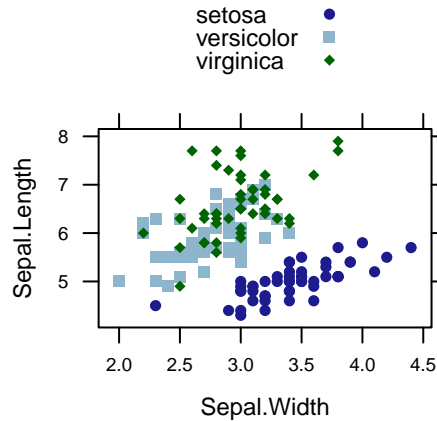
1.7 Customizing Graphics: A Few Bells and Whistles

There are lots of arguments that control the appearance of plots created in R. Here are just a few examples, some of which we have already seen.

1.7.1 auto.key

auto.key=TRUE turns on a simple legend for the grouping variable. (There are ways to have more control, if you need it.)

```
xyplot(Sepal.Length ~ Sepal.Width, groups = Species, data = iris, auto.key = TRUE)
```

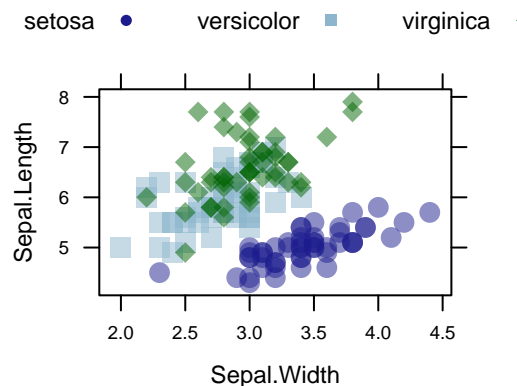


1.7.2 alpha, cex

Sometimes it is nice to have elements of a plot be partly transparent. When such elements overlap, they get darker, showing us where data are “piling up.” Setting the `alpha` argument to a value between 0 and 1 controls the degree of transparency: 1 is completely opaque, 0 is invisible. The `cex` argument controls “character expansion” and can be used to make the plotting “characters” larger or smaller by specifying the scaling ratio.

Here is another example using data on 150 iris plants of three species.

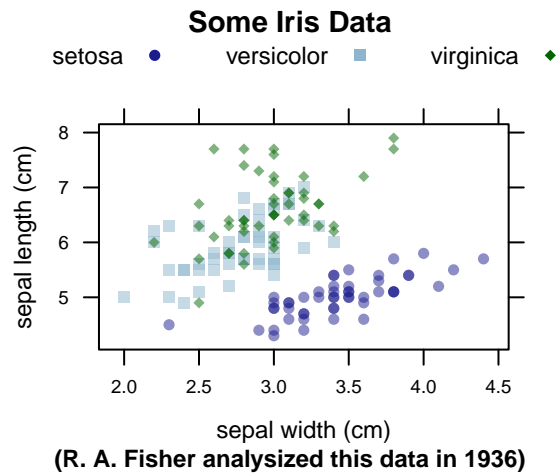
```
xyplot(Sepal.Length ~ Sepal.Width, groups = Species, data = iris, auto.key = list(columns = 3),
       alpha = 0.5, cex = 1.3)
```



main, sub, xlab, ylab

You can add a title or subtitle, or change the default labels of the axes.

```
xyplot(Sepal.Length ~ Sepal.Width, groups = Species, data = iris, main = "Some Iris Data",
       sub = "(R. A. Fisher analyzed this data in 1936)", xlab = "sepal width (cm)", ylab = "sepal length",
       alpha = 0.5, auto.key = list(columns = 3))
```



layout

`layout` can be used to control the arrangement of panels in a multi-panel plot. The format is

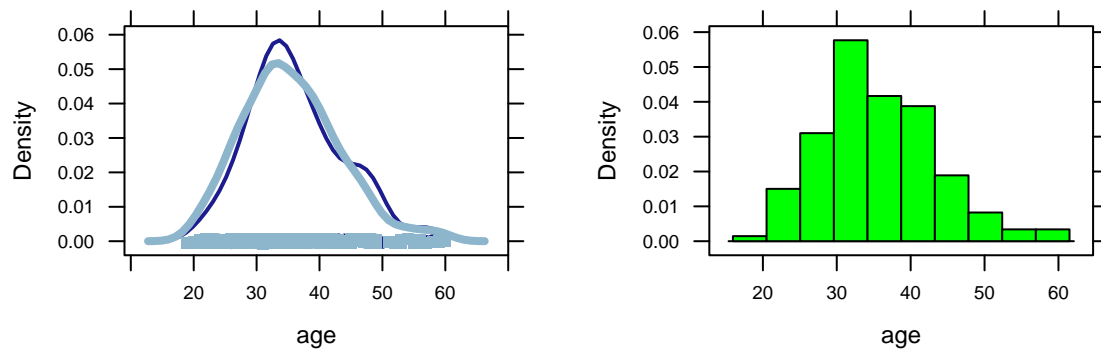
```
layout = c(cols, rows)
```

where `cols` is the number of columns and `rows` is the number of rows. (Columns first because that is the x -coordinate of the plot.)

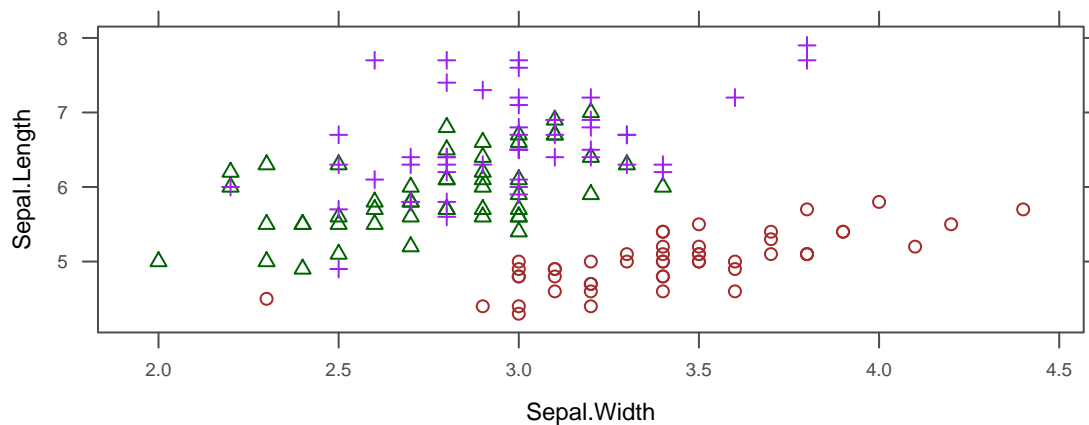
`lty`, `lwd`, `pch`, `col`

These can be used to change the line type, line width, plot symbol, and color, respectively. To specify multiples (one for each group), use the `c()` function (to remember this function, remember that “c” is for “concatenate”). An example is below.

```
densityplot(~age, data = HELPrct, groups = sex, lty = 1, lwd = c(2, 4))
histogram(~age, data = HELPrct, col = "green")
```



```
# There are 25 numbered plot symbols
xyplot( Sepal.Length ~ Sepal.Width, data=iris, groups=Species,
        pch=c(1,2,3), col=c('brown', 'darkgreen', 'purple'), cex=.75 )
```



Note: If you change the colors and symbols this way, they will *not* match what is generated in the legend using `auto.key=TRUE`. So it can be better to set these things in a different way if you are using `groups`. See below.

You can see a list of the hundreds of available color names using

```
colors()
```

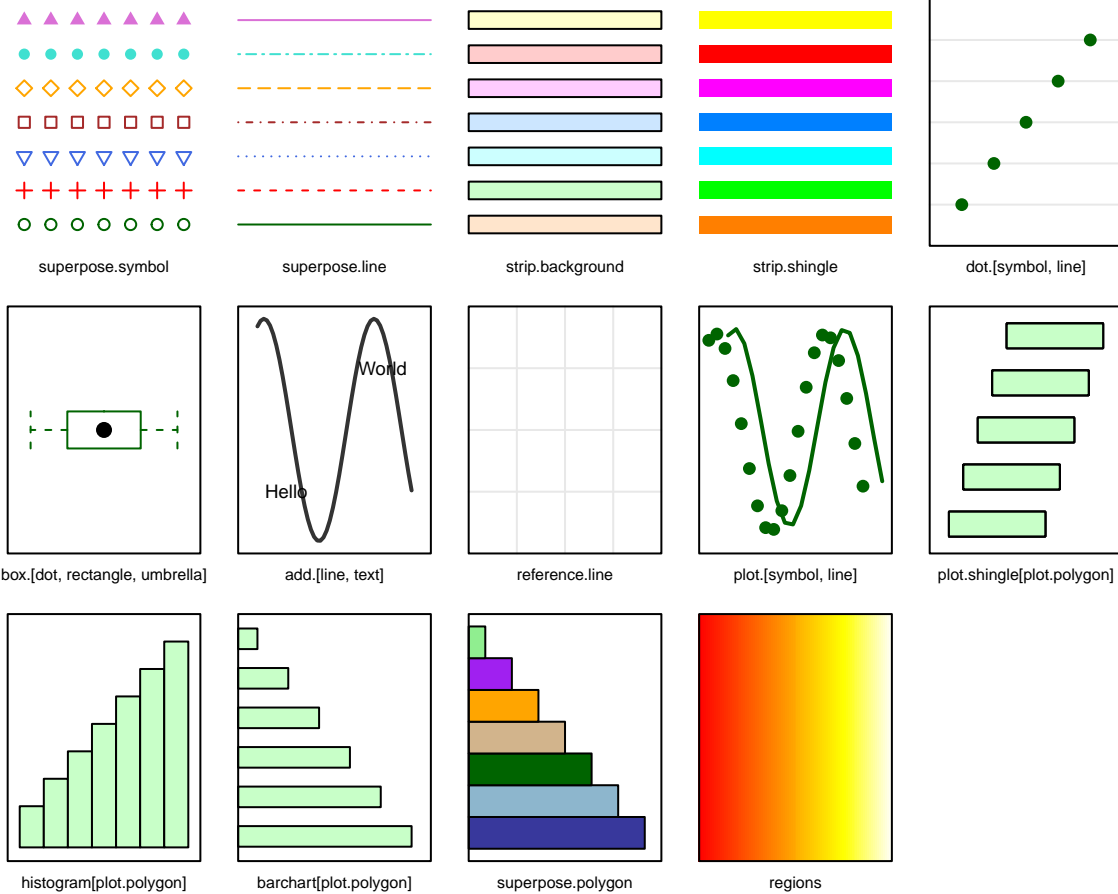
1.7.3 trellis.par.set()

Default settings for lattice graphics are set using `trellis.par.set()`. Don't like the default font sizes? You can change them! For example, change to a 7 point (base) font using

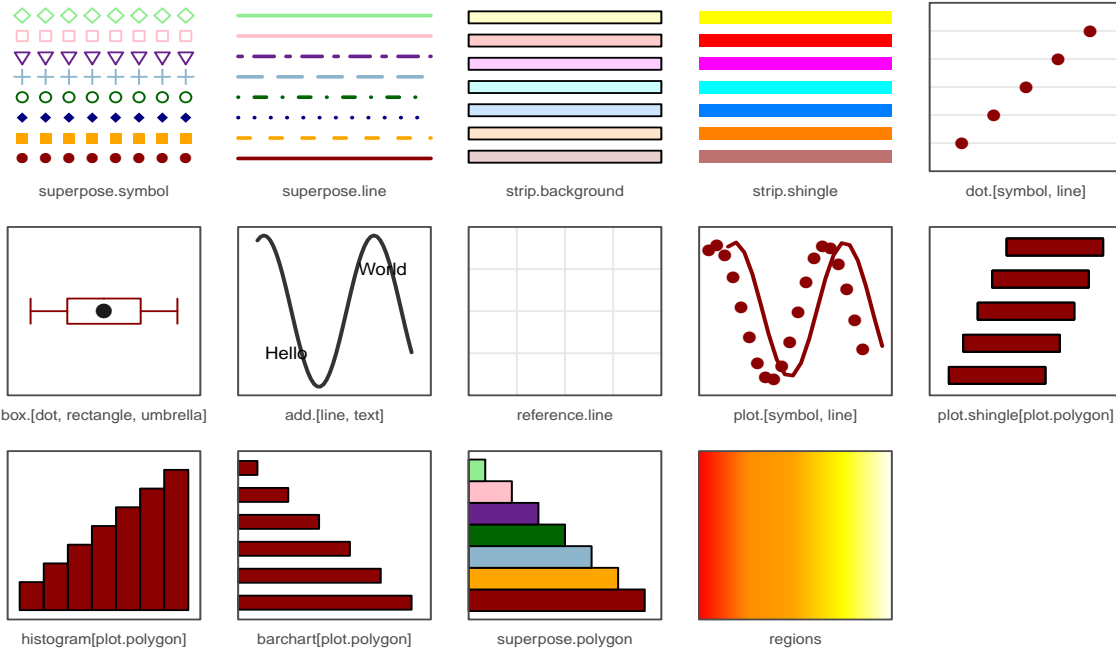
```
trellis.par.set(fontsize = list(text = 7)) # base size for text is 7 point
```

Nearly every feature of a lattice plot can be controlled: fonts, colors, symbols, line thicknesses, colors, etc. Rather than describe them all here, we'll mention only that groups of these settings can be collected into a theme. `show.settings()` will show you what the current theme looks like. Below are a few examples of changing the theme settings, then viewing the results.

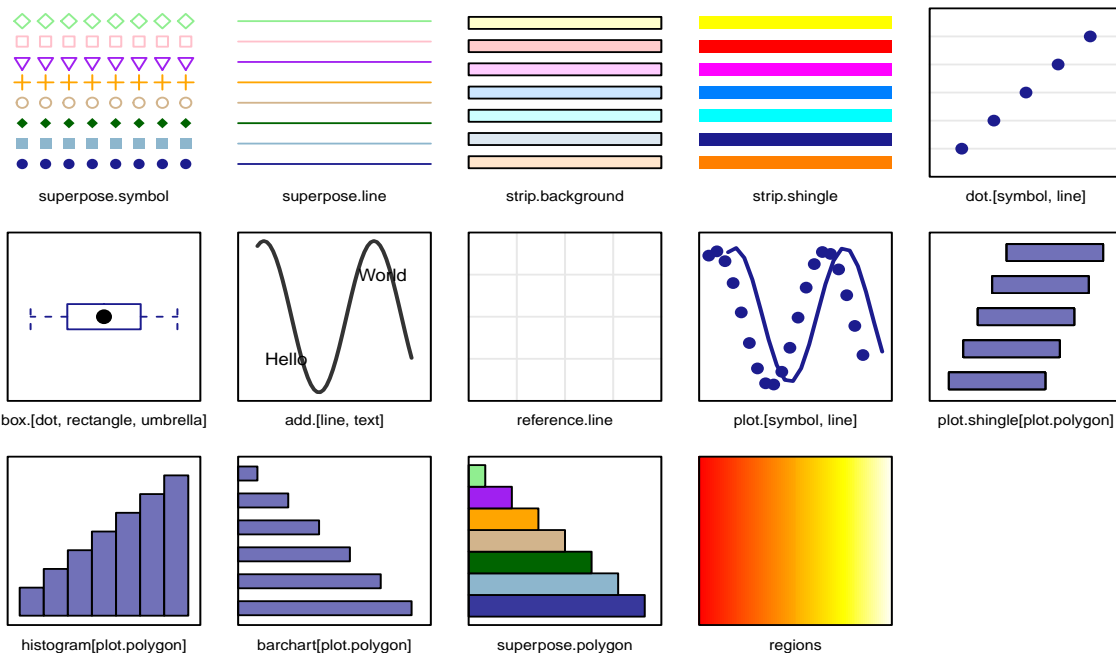
```
trellis.par.set(theme = col.whitebg()) # a theme in the lattice package
show.settings()
```

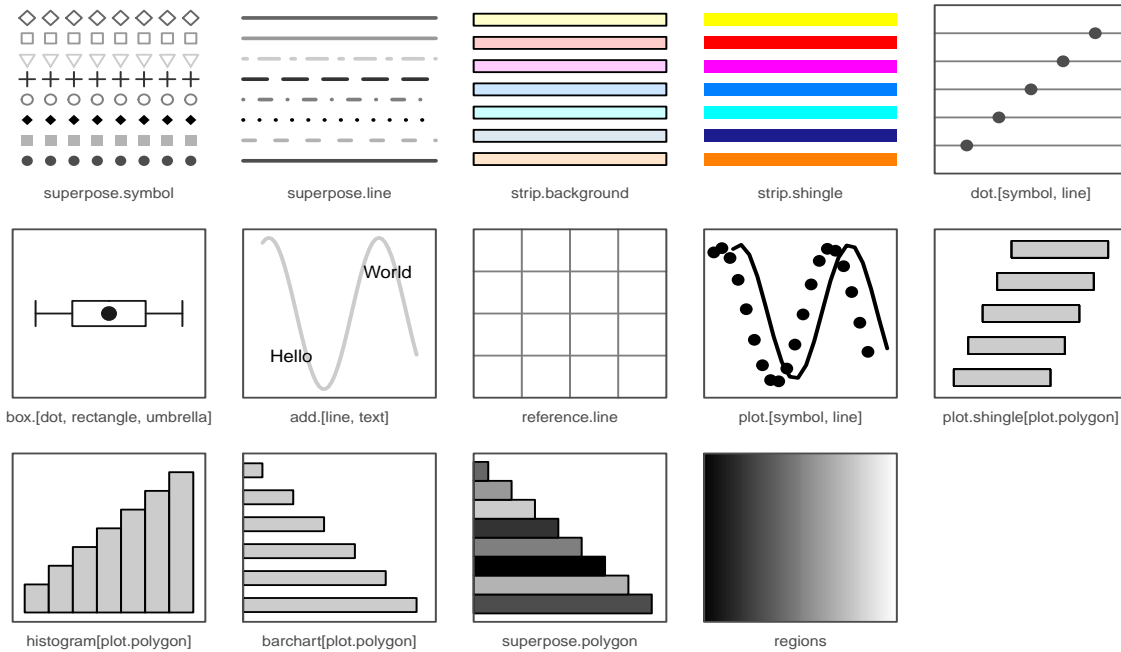
```
require(abd)
trellis.par.set(theme = col.abd()) # a theme in the abd package
show.settings()
```



```
trellis.par.set(theme = col.mosaic) # a theme in the mosaic package
show.settings()
```



```
trellis.par.set(theme = col.mosaic(bw = TRUE)) # a b/w theme in the mosaic package
show.settings()
```



```
trellis.par.set(theme = col.mosaic()) # back to the mosaic theme
trellis.par.set(fontsize = list(text = 9)) # and back to a 10 point font
```

Want to save your settings?

```
# save current settings
mySettings <- trellis.par.get()
# switch to abd defaults
trellis.par.set(theme = col.abd())
# switch back to my saved settings
trellis.par.set(mySettings)
```

1.8 Getting Help in RStudio

1.8.1 The RStudio help system

There are several ways to get RStudio to help you when you forget something. Most objects in packages have help files that you can access by typing something like:

```
?bargraph
?histogram
?HELPrct
```

You can search the help system using

```
help.search("Grand Rapids") # Does R know anything about Grand Rapids?
```

This can be useful if you don't know the name of the function or data set you are looking for.

1.8.2 Tab completion

As you type the name of a function in RStudio, you can hit the tab key and it will show you a list of all the ways you could complete that name. After you type the opening parenthesis, if you hit the tab key, you will get a list of all the possible input arguments and (sometimes) some helpful hints about what they are.)

1.8.3 History

If you know you have done something before, but can't remember how, you can search your history. The history tab shows a list of recently executed commands. There is also a search bar to help you find things from longer ago.

1.8.4 Error messages

When things go wrong, R tries to help you out by providing an error message. Typos are probably the most common cause of errors: for example, you might misspell a function or argument name, forget to close a set of parentheses or brackets, or misplace a comma. One common error message is illustrated below.

```
fred <- 23
frd

## Error in eval(expr, envir, enclos): object 'frd' not found
```

The object `frd` is not found because it was mistyped. It should have been `fred`. Another common mistake is forgetting to load required packages. If you see an “object not found” message, check your typing and check to make sure that the necessary packages have been loaded. If you get an error and can't make sense of the message, you can try copying and pasting your command and the error message and sending to me in an email.

1.9 Graphical Summaries – Important Ideas

1.9.1 The Most Important Template

The plots we have created have all following a single template

```
goal ( formula , data = mydata )
```

We will see this same template used again for numerical summaries and linear and non-linear modeling as well, so it is important to master it.

- **goal:** The name of the function generally describes your goal, the thing you want the computer to produce for you. In the case of plotting, it is the name of the plot. When we do numerical summaries it will be the name of the numerical summary (mean, median, etc.).

- **formula:** For plotting, the formula describes which variables are used on the x-axis, the y-axis and for conditioning. The general scheme is

```
y ~ x | z
```

where **z** is the conditioning variable. Sometimes **y** or **z** are missing (but the right-hand side **x** must always be included in a formula).

- **data:** A data frame must be given in which the variables mentioned in the formula can be found. Variables not found there will be looked for in the enclosing environment. Sometimes we will take advantage of this to avoid creating a temporary data frame just to make a quick plot, but generally it is best to have all the information inside a data frame.

1.9.2 Patterns and Deviations from Patterns

The goal of a statistical plot is to help us *see*

- potential patterns in the data, and
- deviations from those patterns.

1.9.3 Different Plots for Different Kinds of Variables

Graphical summaries can help us see the *distribution* of a variable or the *relationships* between two (or more) variables. The type of plot used will depend on the kinds of variables involved. Later, when we do more quantitative statistical analysis, we will see that the analysis we use will also depend on the kinds of variables involved, so this is an important idea.

1.9.4 Side-by-side Plots and Overlays Can Reveal Importance of Additional Factors

The **lattice** graphics plots make it particularly easy to generate plots that divide the data into groups and either produce a panel for each group (using **l**) or display each group in a different way (different colors or symbols, using the **groups** argument). These plots can reveal the possible influence of additional variables – sometimes called covariates.

1.9.5 Area = (relative) frequency

Many plots are based on the key idea that our eyes are good at comparing areas. Plots that use area (e.g., histograms, mosaic plots, bar charts, pie charts) should always obey this principle

$$\text{Area} = (\text{relative}) \text{ frequency}$$

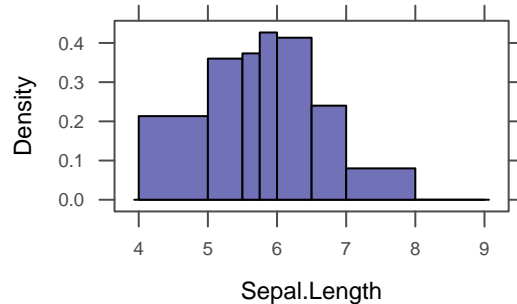
Plots that violate this principle can be deceptive and distort the true nature of the data.

An Example: Histogram with unequal bin widths

It is possible to make histograms with bins that have different widths. But in this case it is important that the height of the bars is chosen so that area (*NOT height*) is proportional to frequency. Using height instead of area would distort the picture.

When unequal bin sizes are specified, `histogram()` by default chooses the density scale:

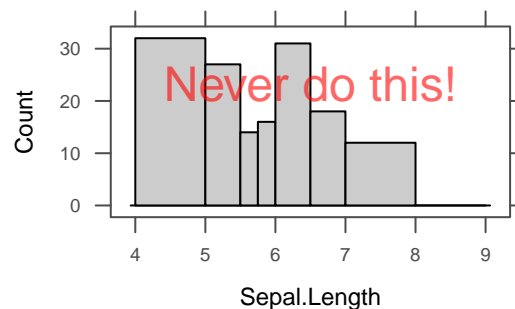
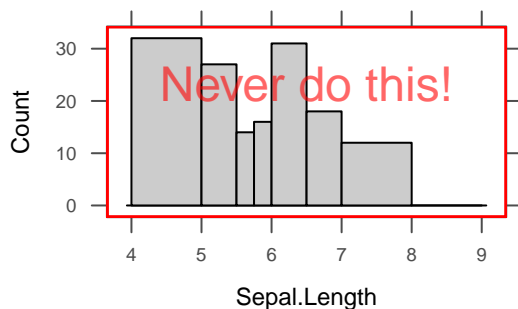
```
histogram(~Sepal.Length, data = iris, breaks = c(4, 5, 5.5, 5.75, 6, 6.5, 7, 8, 9))
```



The density scale is important. It tells R to use a scale such that the area (height \times width) of the rectangles is equal to the relative frequency. For example, the bar from 5.0 to 5.5 has width $\frac{1}{2}$ and height about 0.36, so the area is 0.18, which means approximately 18% of the sepal lengths are between 5.0 and 5.5.

It would be incorrect to choose `type="count"` or `type="proportion"` since this distorts the picture of the data. Fortunately, R will warn you if you try:

```
## Warning in histogram.formula(~Sepal.Length, data = iris, breaks = c(4, 5, : type='count' can
be misleading in this context
```



Notice how different this looks. Now the heights are equal to the relative frequency, but this makes the wider bars have too much area.

Exercises

In your answers to these questions, include both the plots and the code you used to make them as well as any required discussion. Once you have obtained a basic plot that satisfies the requirements of the question, feel free to use some of the “bells and whistles” to make the plots even better.

1.1 Create a scatterplot using the two variables in the `oldfaith` data frame. What do we learn about Old Faithful eruptions from this plot?

1.2 Where do the data in the `CPS85` data frame (in the `mosaic` package) come from? What are the observational units? How many are there?

1.3 Choose a quantitative variable that interests you in the `CPS85` data set. Make an appropriate plot and comment on what you see.

1.4 Choose a categorical variable that interests you in the `CPS85` data set. Make an appropriate plot and comment on what you see.

1.5 Create a plot that displays two or more variables from the `CPS85` data. At least one should be quantitative and at least one should be categorical. Comment on what you can learn from your plot.

1.6 Where do the data in the `mpg` data frame (in the `ggplot2` package) come from? What are the observational units? How many are there?

1.7 Choose a quantitative variable that interests you in the `mpg` data set. Make an appropriate plot and comment on what you see.

1.8 Choose a categorical variable that interests you in the `mpg` data set. Make an appropriate plot and comment on what you see.

1.9 Create a plot that displays two or more variables from the `mpg` data. At least one should be quantitative and at least one should be categorical. Comment on what you can learn from your plot.

1.10 The file at <http://www.calvin.edu/~rpruim/data/Fires.csv> is a csv file containing data on wild lands fires in the US over a number of years. You can load this data one of two ways.

- Go to the workspace tab, select Import Data Set, choose From Web URL... and follow the instructions.
- Use the following command in R:

```
Fires <- read.csv("http://www.calvin.edu/~rpruim/data/Fires.csv")
```

You can also use either of these methods to read from a file rather than from a web URL, so this is a good way to get your own data into R.

- a) The source for these data claim that data before a certain year should not be compared to data from after that year because the older data were computed a different way and are not considered as reliable. What year is the break point? Use graphs of the data over time to estimate when something changed.
- b) You can trim the data to just the subset you want using `subset()`. For example, to get just the subset of years since 1966, you would use

```
Fires2 <- subset(Fires, Year > 1966)
```

Be sure to use a new name for the subset data frame if you want to keep the original data available.

Use `subset()` to create a data set that contains only the data from the new data regime (based on your answer in the previous problem).

- c) Using only the data from this smaller set, how would you describe what is happening with fires over time?

1.11 Use R's help system to find out what the `i1` and `i2` variables are in the `HELPrct` data frame. Make histograms for each variable and comment on what you find out. How would you describe the shape of these distributions? Do you see any outliers (observations that don't seem to fit the pattern of the rest of the data)?

1.12 Compare the distributions of `i1` and `i2` among men and women.

1.13 Compare the distributions of `i1` and `i2` among the three `substance` groups.

1.14 The `SnowGR` contains historical data on snowfall in Grand Rapids, MI. The snowfall totals for November and December 2014 were 31 inches and 1 inch, respectively.

- a) Create histograms of November and December snowfall totals. How unusual were the snowfall totals we had in 2014?
- b) If there is very little snow in December, should we expect to have unusually much or little snow in February? Make a scatter plot comparing December and February historic snowfall totals and comment on what you see there.

2

Numerical Summaries

2.1 Tabulating Data

A table is one kind of numerical summary of a data set. In fact, you can think of histograms and bar graphs as graphical representations of summary tables. But sometimes it is nice to have the table itself. R provides several ways of obtaining such tables.

2.1.1 Tabulating a categorical variable

The formula interface

There are several functions for tabulating categorical variables. `tally()` uses a syntax that is very similar to `bargraph()`. We'll call this method the **formula interface**. (R calls anything with a wiggle (`~`) a formula.)

```
tally(~substance, data = HELPrct)

##
## alcohol cocaine heroin
##      177      152      124

tally(~substance, data = HELPrct, format = "prop")

##
## alcohol cocaine heroin
## 0.3907285 0.3355408 0.2737307

tally(~substance, data = HELPrct, format = "perc")

##
## alcohol cocaine heroin
## 39.07285 33.55408 27.37307
```

The `$`-interface

`table()` and its cousins use the `$` operator which selects one variable out of a data frame.

```
KidsFeet$sex      # general syntax: dataframe$variable

##  [1] B B B B B B B G G B B B B B G G G G G B B G G G B G B B B G G G B B G G G G
## Levels: B G
```

We'll call this interface the `$`-interface.

```
table(HELPrct$substance)

##
## alcohol cocaine  heroin
##      177      152      124

perctable(HELPrct$substance) # display percents instead of counts

##
## alcohol cocaine  heroin
## 39.07285 33.55408 27.37307

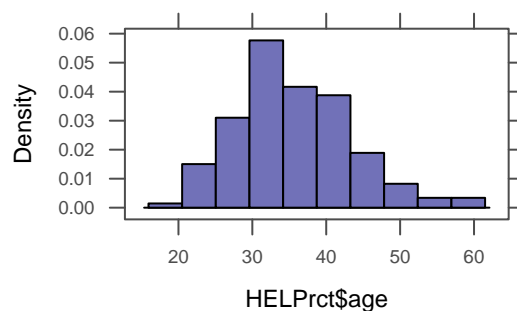
proptable(HELPrct$substance) # display proportions instead of counts

##
## alcohol cocaine  heroin
## 0.3907285 0.3355408 0.2737307
```

Two interfaces

Some functions in R require the formula interface, some require the `$`-interface, and some allow you to use either one.¹ For example, `histogram` will also work like this.

```
histogram(HELPrct$age)
```



¹One of the things that the `mosaic` package does is provide a formula interface for many functions that only had a `$`-interface before.

But notice that the output is not quite as nice, since the default label for the horizontal axis now shows both the data frame name and the variable name with a `$` between. *My advice is to use formula interfaces whenever they are available.*

2.1.2 Tabulating a quantitative variable

Although `tally()` and `table()` work with quantitative variables as well as categorical variables, this is only useful when there are not too many different values for the variable.

```
tally(~age, data = HELPrct)
```

```
##
## 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
##  1  2  3  8  5  8  7 13 18 15 18 18 20 28 35 18 25 23 20 18 27 10 20 10 13  7 13  5 14  5
## 49 50 51 52 53 54 55 56 57 58 59 60
##  8  2  1  1  3  1  2  1  2  2  2  1
```

Tabulating in bins (optional)

It is often more convenient to group quantitative data into bins. We just have to tell R what the bins are. For example, suppose we wanted to group the 20s, 30s, 40s, etc. together.

```
# let's add a new variable to HELPrct
```

```
HELPrct <- transform(HELPrct, binnedAge = cut(age, breaks = c(10, 20, 30, 40, 50, 60, 70)))
head(HELPrct)
```

```
##   age anysubstatus  anysub cesd d1 daysanysub dayslink drugrisk e2b female  sex g1b
## 1  37             1   yes  49  3         177       225      0  NA      0  male yes
## 2  37             1   yes  30 22           2        NA      0  NA      0  male yes
## 3  26             1   yes  39  0           3       365     20  NA      0  male  no
## 4  39             1   yes  15  2        189       343      0  1      1 female no
## 5  32             1   yes  39 12           2        57      0  1      0  male  no
## 6  47             1   yes   6  1          31       365      0  NA      1 female no
##  homeless i1 i2 id indtot linkstatus link      mcs      pcs pss_fr racegrp satreat
## 1  housed 13 26  1    39          1 yes 25.111990 58.41369      0  black      no
## 2 homeless 56 62  2    43          NA <NA> 26.670307 36.03694      1  white      no
## 3  housed  0  0  3    41           0 no  6.762923 74.80633     13  black      no
## 4  housed  5  5  4    28           0 no 43.967880 61.93168     11  white     yes
## 5 homeless 10 13  5    38           1 yes 21.675755 37.34558     10  black      no
## 6  housed  4  4  6    29           0 no 55.508991 46.47521      5  black      no
##  sexrisk substance treat binnedAge
## 1      4  cocaine   yes  (30,40]
## 2      7  alcohol   yes  (30,40]
## 3      2  heroin    no  (20,30]
## 4      4  heroin    no  (30,40]
## 5      6  cocaine   no  (30,40]
## 6      5  cocaine   yes  (40,50]
```

```
tally(~binnedAge, data = HELPrct)
```

```
##
## (10,20] (20,30] (30,40] (40,50] (50,60] (60,70]
##      3      113      224      97      16      0

table(HELPrct$binnedAge)

##
## (10,20] (20,30] (30,40] (40,50] (50,60] (60,70]
##      3      113      224      97      16      0
```

That's not quite what we wanted: 30 is in with the 20s, for example. Here's how we fix that.

```
HELPrct <- transform( HELPrct,
  binnedAge = cut(age, breaks=c(10,20,30,40,50,60,70), right=FALSE) )
tally( ~binnedAge, data=HELPrct )

##
## [10,20) [20,30) [30,40) [40,50) [50,60) [60,70)
##      1      97      232      105      17      1

table( HELPrct$binnedAge)

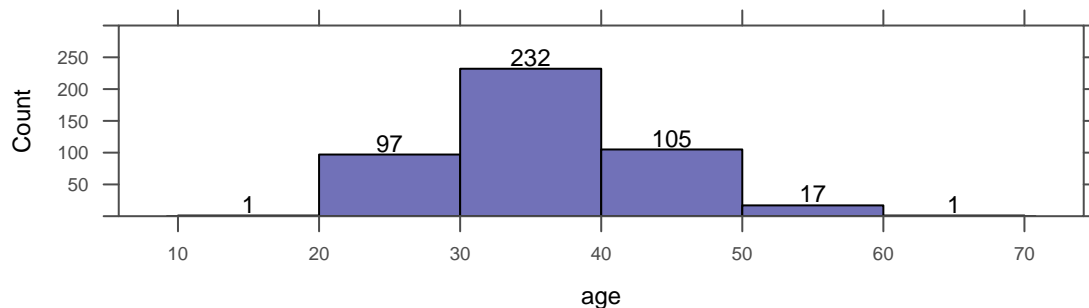
##
## [10,20) [20,30) [30,40) [40,50) [50,60) [60,70)
##      1      97      232      105      17      1
```

We won't use this very often, since typically seeing this information in a histogram is more useful.

Labeling a histogram

The `histogram()` function offers you the option of adding the counts to the graph, via the logical (TRUE/FALSE) `label` argument.

```
histogram(~age, data = HELPrct, label = TRUE, type = "count", width = 10, center = 5, ylim = c(0,
  300), right = FALSE)
```



2.1.3 Cross-tables: Tabulating two or more variables

`xtabs()` can also compute cross tables for two (or more) variables.

```
tally(~sex + substance, data = HELPrct)
```

```
##           substance
## sex      alcohol cocaine heroin
## female      36      41      30
## male       141     111      94
```

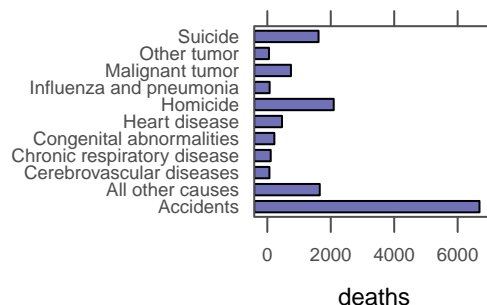
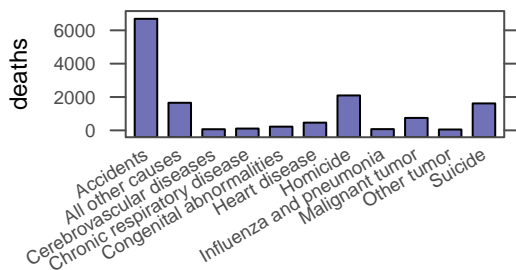
2.2 Working with Pre-Tabulated Data

Sometimes data arrive pre-tabulated. We can use `barchart()` instead of `bargraph()` to graph pre-tabulated data.²

```
require(abd) # data sets from Analysis of Biological Data
TeenDeaths
```

```
##           cause deaths
## 1           Accidents 6688
## 2           Homicide 2093
## 3           Suicide 1615
## 4 Malignant tumor  745
## 5           Heart disease 463
## 6 Congenital abnormalities 222
## 7 Chronic respiratory disease 107
## 8 Influenza and pneumonia  73
## 9 Cerebrovascular diseases  67
## 10          Other tumor   52
## 11        All other causes 1653
```

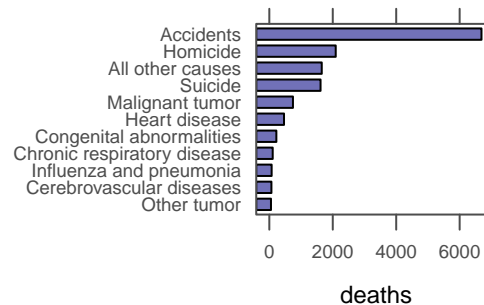
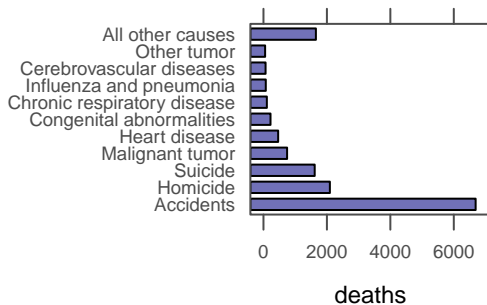
```
barchart(deaths ~ cause, data = TeenDeaths, scales = list(x = list(rot = 30)))
# ('scales' argument above is to rotate the x-labels so they don't overlap) you may find it
# useful, but learning it is optional.
barchart(cause ~ deaths, data = TeenDeaths)
```



²`bargraph()` converts raw data into a summary table and then calls `barchart()` to do the plotting.

Notice that by default the causes are displayed in alphabetical order. R assumes that categorical data is nominal (that is, there is no particular natural or logical ordering to the categories) unless you say otherwise. Here are two ways to have things appear in a different order. The first uses the order in which categories (to R “levels”) appear in the `TeenDeaths` data frame. The second reorders based on the value of the `deaths` variable.

```
barchart(ordered(cause, levels = cause) ~ deaths, TeenDeaths)
barchart(reorder(cause, deaths) ~ deaths, TeenDeaths)
```



2.3 Summarizing Distributions of Quantitative Variables

Important Note

Numerical summaries are a convenient way to describe a distribution, but remember that numerical summaries do not necessarily tell you everything there is to know about a distribution. When working with a new dataset, it is *always* important to explore the data as fully as possible (commonly including graphical as well as numerical summaries, and sometimes even examining the data table directly) before accepting any simplified summary as a good representation of the data. You might discover certain patterns in the data, interesting features, or even outliers or mistakes in the data, that make certain summaries misrepresentations of the whole.

Notation

In statistics n (or sometimes N) almost always means the number of observations (i.e., the number of rows in a data frame).

If y is a variable in a data set with n observational units, we can denote the n values of y as

- $y_1, y_2, y_3, \dots, y_n$ (in the original order of the data).
- $y_{(1)}, y_{(2)}, y_{(3)}, \dots, y_{(n)}$ (in sorted order from smallest to largest).

The symbol \sum represents summation (adding up a bunch of values).

2.4 Measures of Center

Measures of center attempt to give us a sense of what is a typical value for the distribution.

$$\text{mean of } y = \bar{y} = \frac{\sum_{i=1}^n y_i}{n} = \frac{\text{sum of values}}{\text{number of values}}$$

median of y = the “middle” number (after putting the numbers in increasing order)

- The mean is the “balancing point” of the distribution.
- The median³ is the 50th percentile: half of the distribution is below the median, half is above.
- If the distribution is symmetric, then the mean and median are the same.
- In a skewed distribution, the mean is pulled farther toward the tail than the median is.
- *A few very large or very small values can change the mean a lot*, so the mean is **sensitive to outliers** and is a better measure of center when the distribution is symmetric than when it is skewed.
- The median is a **resistant measure** (resistant to the presence of outlier) – it is not affected much by a few very large or very small values.

2.5 Measures of Spread

$$\text{variance of } y = s_y^2 = \frac{\sum_{i=1}^n (y_i - \bar{y})^2}{n - 1}$$

$$\text{standard deviation of } y = s_y = \sqrt{s_y^2}$$

= square root of variance

$$\text{interquartile range} = \text{IQR} = Q_3 - Q_1$$

= difference between first and third quartiles (defined shortly)

- Roughly, the standard deviation is the “average deviation from the mean”. (That’s not exactly right because of the squaring involved and because we are dividing by $n - 1$ instead of by n . More on that denominator later.)
- The mean and standard deviation are especially useful for describing **normal distributions** and other unimodal, symmetric distributions that are roughly “bell-shaped”. (We’ll learn more about normal distributions later.)
- Like the mean, the variance and standard deviation are sensitive to outliers and less suited for summarizing skewed distributions.
- It is perhaps of some value to compute the variance and standard deviation by hand once or twice to make sure you understand how these measures are defined, but we will typically let R do the calculations for us.

³A note about calculating medians: If the number of datapoints is odd, the median is the middle value (after putting the observations in increasing order). In cases where there is an even number of observations, the median is the average of the middle two observations.

To get a numerical summary of a variable (a statistic), we need to tell R which statistic we want and the variable and data frame involved. There several ways we can do this in R. Here are several ways to get the mean, for example:

```
mean(HELPrct$age)           # this is the most traditional way; should always work

## [1] 35.65342

mean(~age, data=HELPrct)    # similar to our plotting methods; only works for some functions

## [1] 35.65342

with(HELPrct, mean(age))    # one more way this can be done

## [1] 35.65342
```

Using the formula style, we can now compute several different statistics.

```
mean(~age, data = HELPrct)

## [1] 35.65342

sd(~age, data = HELPrct)

## [1] 7.710266

var(~age, data = HELPrct)

## [1] 59.4482
```

```
median(~age, data = HELPrct)

## [1] 35

IQR(~age, data = HELPrct)

## [1] 10

favstats(~age, data = HELPrct) # this computes several statistics at once

##   min Q1 median Q3 max   mean    sd  n missing
##   19 30    35  40  60 35.65342 7.710266 453      0
```

It is also possible to compute these statistics separately for each of several groups. The syntax is much like the the syntax we used when plotting. In fact, we have two choices for the formula: $y \sim x$ or $\sim x | z$.


```
mean(age ~ sex, data = HELPrct)

##   female      male
## 36.25234 35.46821

sd(age ~ sex, data = HELPrct)

##   female      male
## 7.584858 7.750110

favstats(~age | sex, data = HELPrct)

##   .group min Q1 median   Q3 max   mean      sd  n missing
## 1 female  21 31   35 40.5  58 36.25234 7.584858 107      0
## 2 male   19 30   35 40.0  60 35.46821 7.750110 346      0
```

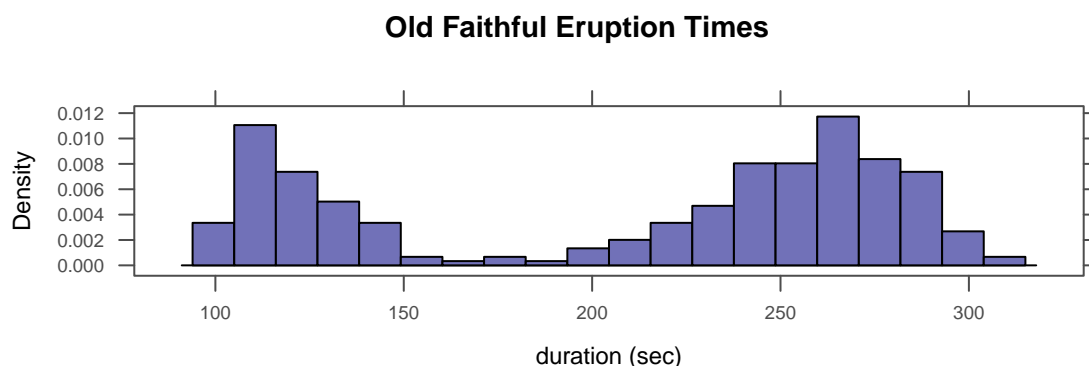
2.5.1 A word of caution

None of these measures (especially the mean and median) is a particularly good summary of a set of data if the distribution of the data is not unimodal. The histogram below shows the lengths of eruptions of the Old Faithful geyser at Yellowstone National Park.

```
favstats(~ Duration, data=oldfaith)

##   min  Q1 median   Q3 max   mean      sd  n missing
##   96 130   240 267.75 306 209.8778 68.39213 270      0

histogram( ~ Duration, data=oldfaith, n=20,
           main="Old Faithful Eruption Times", xlab="duration (sec)")
```



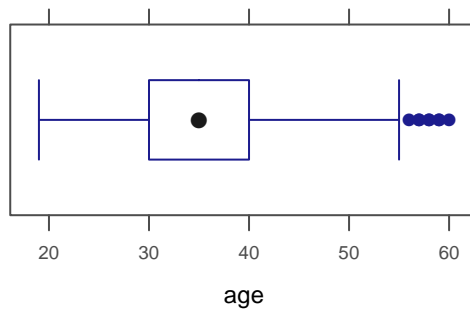
Notice that the mean and median do not represent typical eruption times very well. Nearly all eruptions are either quite a bit shorter or quite a bit longer. (This is especially true of the mean.)

2.5.2 Box plots

Boxplots (also called box-and-whisker plots) are a graphical representation of a **5-number summary** of a quantitative variable. The five numbers are the five **quantiles**:

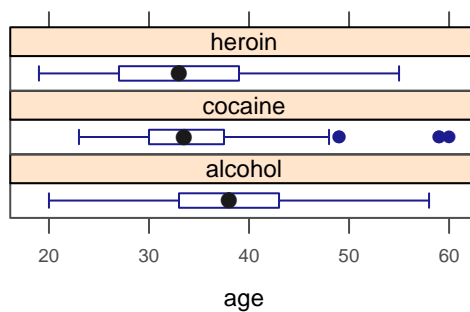
- Q_0 , the minimum
- Q_1 , the first quartile (25th percentile)
- Q_2 , the median (50th percentile)
- Q_3 , the third quartile (75th percentile)
- Q_4 , the maximum

```
bwplot(~age, data = HELPrct)
```



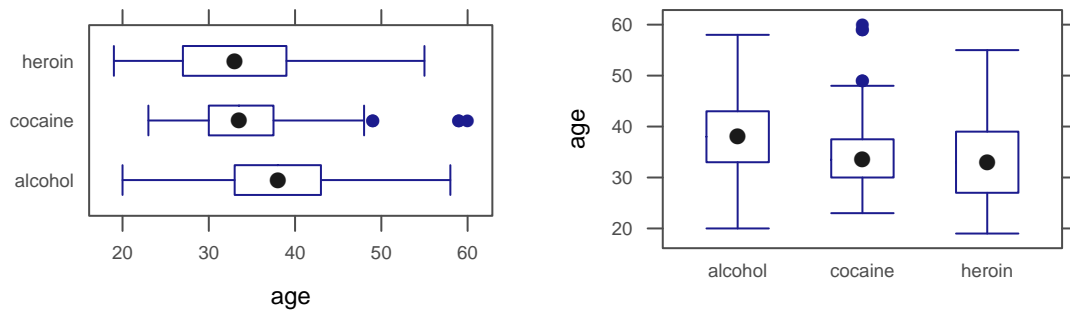
Boxplots provide a way of comparing multiple groups that is especially informative and visually effective. Here is one way to make boxplots of multiple groups (it should look familiar from what we know about histogram):

```
bwplot(~age | substance, data = HELPrct, layout = c(1, 3))
```



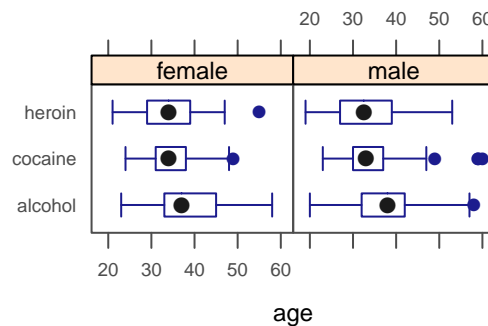
But `bwplot()` has a better way. Put the quantitative variable on one side of the wiggle and the categorical on the other. The placement determines which goes along the vertical axis and which along the horizontal axis – just like it did for `xyplot()`.

```
bwplot(substance ~ age, data = HELPrct)
bwplot(age ~ substance, data = HELPrct)
```



And we can combine this idea with conditioning:

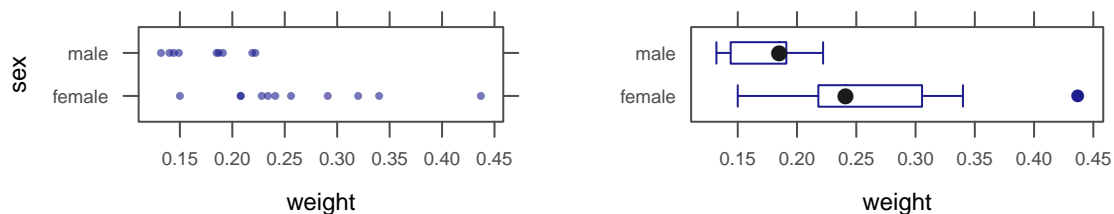
```
bwplot(substance ~ age | sex, data = HELPrct)
```



2.5.3 Small data sets

When we have relatively small data sets, it may not make sense to use a boxplot. With very few observations, boxplots can be misleading, in that they suggest the presence of more observations than are really contained in the dataset. In these cases, it is better to display all the data. `xyplot()` allows you to put a categorical variable along one axis and a quantitative variable along the other. For some data sets, either option can produce a plot that gives a good picture of the data.

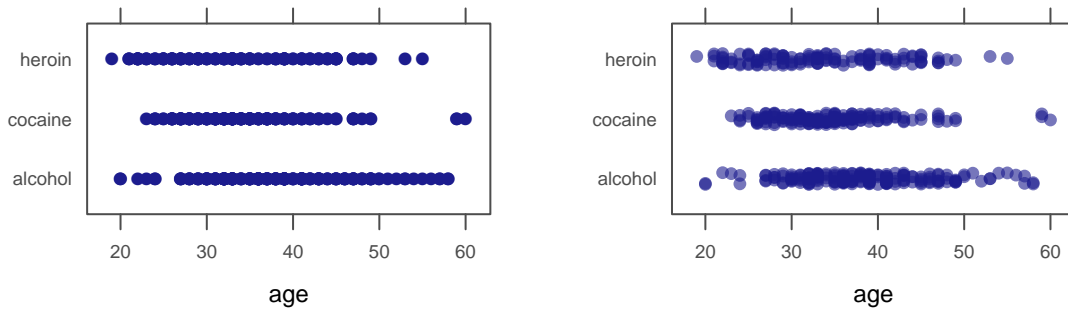
```
xyplot(sex ~ weight, data = Mosquitoes, cex = 0.5, alpha = 0.6)
bwplot(sex ~ weight, data = Mosquitoes)
```



R actually provides a separate function, `stripplot()`, for this type of scatter plot (where one variable is categorical). Note the effect of the `jitter` argument – it moves each data point slightly up or down, to reduce

overplotting (data points being plotted exactly on top of one another) and make it clearer how many data-points were observed for each possible combination of x- and y-values.

```
stripplot(substance ~ age, data = HELPrct)
stripplot(substance ~ age, data = HELPrct, jitter = TRUE, alpha = 0.6)
```



2.6 Summarizing Categorical Variables

The most common summary of a categorical variable is the **proportion** of observations in each category. For a single category:

$$\hat{p} = \frac{\text{number in one category}}{n}$$

Proportions can be expressed as fractions, decimals or percents. For example, if there are 10 observations in one category and $n = 50$ observations in all, then

$$\hat{p} = \frac{10}{50} = \frac{2}{5} = 0.40 = 40\%$$

If we code our categorical variable using 1 for observations in a single category of interest – “the one category” – and 0 for observations in any other category, then *a proportion is a sample mean*.

$$\frac{1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0}{25} = \frac{10}{25}$$

2.7 Relationships Between Two Variables

It is also possible to give numerical summaries of the relationship between two variables. The most common one is the **correlation coefficient**, which we will learn about later.

Practice Exercises

2.1 Create a data set with $n = 6$ values, each an integer between 0 and 10 (inclusive) that has the smallest possible variance. Compute the mean and variance of this data set “by hand” (that is, without using `mean()` or `sd()` or `var()` in R or similar features on a calculator).

2.2 Create a data set with $n = 6$ values, each an integer between 0 and 10 (inclusive) that has the largest possible variance. Compute the variance of this data set “by hand” (that is, without using `mean()` or `sd()` or `var()` in R or similar features on a calculator).

2.3 Create side-by-side boxplots of the variable `i1` (average number of drinks per day) comparing the different `substance` groups in the `HELPrct` data frame.

For each `substance` group, explain how you can tell from the boxplots whether the mean will be larger than the median or the median larger than the mean.

2.4 Compute the mean and median values of `i1` (average number of drinks per day) for each of the `substance` groups in the `HELPrct` data frame.

Bibliography

CPS85, 31
 DAAG, 7
 HELPrct, 9, 32, 45
 IQR(), 40
 SnowGR, 32
 TeenDeaths, 38
 alr3, 10
 barchart(), 37
 bargraph(), 11, 33, 37
 bwplot(), 11, 13, 42
 c(), 23
 colors(), 24
 cut(), 35
 data(), 8
 densityplot(), 11, 12
 dim, 10
 favstats(), 40
 freqpolygon(), 12
 ggplot2, 31
 histogram(), 11, 12, 15, 30, 36
 knitr, 21
 lattice, 12, 14, 16, 29
 log(), 5
 log10(), 5
 max, 11
 mean(), 40, 45
 mean, 11
 median(), 40
 median, 11
 min, 11
 mosaic, 7, 9–12, 15, 31, 34
 mpg, 31
 mtable(), 35
 ncol, 10
 nrow, 10
 oldfaith, 31
 perctable, 34
 proptable, 34
 sd(), 40, 45
 sd, 11
 show.settings(), 24
 sqrt(), 5
 stripplot(), 43, 44
 subset(), 32
 table(), 34, 35
 tally(), 33, 35
 trellis.par.set(), 24
 var(), 40, 45
 var, 11
 xtabs(), 35, 37
 xyplot(), 16, 42, 43