Computational Statistics Using R and R Studio
An Introduction for Scientists

Randall Pruim

SC 11 Education Program (November, 2011)

Contents

About These Notes

These materials were prepared for the SC 11 Education Program held in Seattle in November 2011. Much of the material is recycled from two workshops:

- *Teaching Statistics Using R*, a workshop conducted prior to the May 2011 United States Conference on Teaching Statistics.

  You can find out more about this workshop at `http://mosaic-web.org/uscots2011/`.

- *Computational Science for Biology Educators*, an SC11 workshop held at Calvin College in June 2011.

  You can find out more about this workshop at `http://www.calvin.edu/isri/sc11/`

The activities and examples in these notes are intended to highlight a modern approach to statistics and statistics education that focuses on modeling, resampling based inference, and multivariate graphical techniques.

These notes contain far more than will be covered in the workshop, so they can serve as a reference for those who want to learn more. For still more reference material, see the above mentioned notes at `http://mosaic-web.org/uscots2011/`.

R and R Packages

R can be obtained from `http://cran.r-project.org/`. Download and installation are pretty straightforward for Mac, PC, or linux machines.

In addition to R, we will make use of several packages that need to be installed and loaded separately. The `mosaic` package (and its dependencies) will be assumed throughout. Other packages may appear from time to time, including

- `fastR`: companion to *Foundations and Applications of Statistics* by R. Pruim
- `abd`: companion to *Analysis of Biological Data* by Whitlock and Schluter
- `vcd`: visualizing categorical data

We also make use of the `lattice` graphics package which is installed with R but must be loaded before use.

RStudio

RStudio is an alternative interface to R. RStudio can be installed as a desktop (laptop) application or as a server application that is accessible to others via the Internet. RStudio is available from

`http://www.rstudio.org/`

Calvin has provided accounts for Education Program participants on an RStudio server at

`http://dahl.calvin.edu:8787`

There are some things in these notes (in particular those using `manipulate()`) that require the RStudio interface to R. Most things should work in any flavor of R.

Marginal Notes

Marginal notes appear here and there. Sometimes these are side comments that we wanted to say, but didn't want to interrupt the flow to mention. These may describe more advanced features of the language or make suggestions about how to implement things in the classroom. Some are warnings to help you avoid common pitfalls. Still others contain requests for feedback.

DIGGING DEEPER
Many marginal notes will look like this one.

CAUTION!
But warnings are set differently to make sure they catch your attention.

SUGGESTION BOX
So, do you like having marginal notes in these notes?

DIGGING DEEPER
If you know LaTeX as well as R, then `Sweave` provides a nice solution for mixing the two.

Document Creation

This document was created November 13, 2011, using `Sweave` and R version 2.13.1 (2011-07-08). Sweave is R's system for reproducible research and allows text, graphics, and R code to be intermixed and produced by a single document.

Project MOSAIC

The USCOTS11 workshop was developed by Project MOSAIC. Project MOSAIC is a community of educators working to develop new ways to introduce mathematics, statistics, computation, and modeling to students in colleges and universities.

The purpose of the MOSAIC project is to help us share ideas and resources to improve teaching, and to develop a curricular and assessment infrastructure to support the dissemination and evaluation of these ideas. Our goal is to provide a broader approach to quantitative studies that provides better support for work in science and technology. The focus of the project is to tie together better diverse aspects of quantitative work that students in science, technology, and engineering will need in their professional lives, but which are today usually taught in isolation, if at all.

In particular, we focus on:

Modeling The ability to create, manipulate and investigate useful and informative mathematical representations of a real-world situations.

Statistics The analysis of variability that draws on our ability to quantify uncertainty and to draw logical inferences from observations and experiment.

Computation The capacity to think algorithmically, to manage data on large scales, to visualize and interact with models, and to automate tasks for efficiency, accuracy, and reproducibility.

Calculus The traditional mathematical entry point for college and university students and a subject that still has the potential to provide important insights to today's students.

Drawing on support from the US National Science Foundation (NSF DUE-0920350), Project MOSAIC supports a number of initiatives to help achieve these goals, including:

Faculty development and training opportunities, such as the USCOTS 2011 workshop and our 2010 gathering at the Institute for Mathematics and its Applications.

M-casts, a series of regularly scheduled seminars, delivered via the Internet, that provide a forum for instructors to share their insights and innovations and to develop collaborations to refine and develop them. A schedule of future M-casts and recordings of past M-casts are available at the Project MOSAIC web site, `http://mosaic-web.org`.

The development of a "concept inventory" to support teaching modeling. It is somewhat rare in today's curriculum for modeling to be taught. College and university catalogs are filled with descriptions of courses in statistics, computation, and calculus. There are many textbooks in these areas and the most new faculty teaching statistics, computation, and calculus have a solid idea of what should be included. But modeling is different. It's generally recognized as important, but few if instructors have a clear view of the essential concepts.

The construction of syllabi and materials for courses that teach the MOSAIC topics in a better integrated way. Such courses and materials might be wholly new constructions, or they might be incremental modifications of existing resources that draw on the connections between the MOSAIC topics.

We welcome and encourage your participation in all of these initiatives.

1 An Introduction to R

This is a lightly modified version of a handout RJP used with his Intro Stats students Spring 2011. Aside from the occasional comment to instructors, this chapter could be used essentially as is with students.

1.1 Welcome to R and RStudio

R is a system for statistical computation and graphics. We use R for several reasons:

1. R is open-source and freely available for Mac, PC, and Linux machines. This means that there is no restriction on having to license a particular software program, or have students work in a specific lab that has been outfitted with the technology of choice.
2. R is user-extensible and user extensions can easily be made available to others.
3. R is commercial quality. It is the package of choice for many statisticians and those who use statistics frequently.
4. R is becoming very popular with statisticians and scientists, especially in certain sub-disciplines, like genetics. Articles in research journals such as *Science* often include links to the R code used for the analysis and graphics presented.
5. R is very powerful. Furthermore, it is gaining new features every day. New statistical methods are often available first in R.

RStudio provides access to R in a web browser. This has some additional advantages: no installation is required, the interface has some additional user-friendly components, and work begun on one machine can be picked up seamlessly later on somewhere else.

The URL for the RStudio server on Calvin's supercomputer is

`http://dahl.calvin.edu:8787`

It is also possible to download RStudio server and set up your own server or RStudio desktop for stand-alone processing.

Once you have logged in to an RStudio server, you will see something like Figure 1.1.

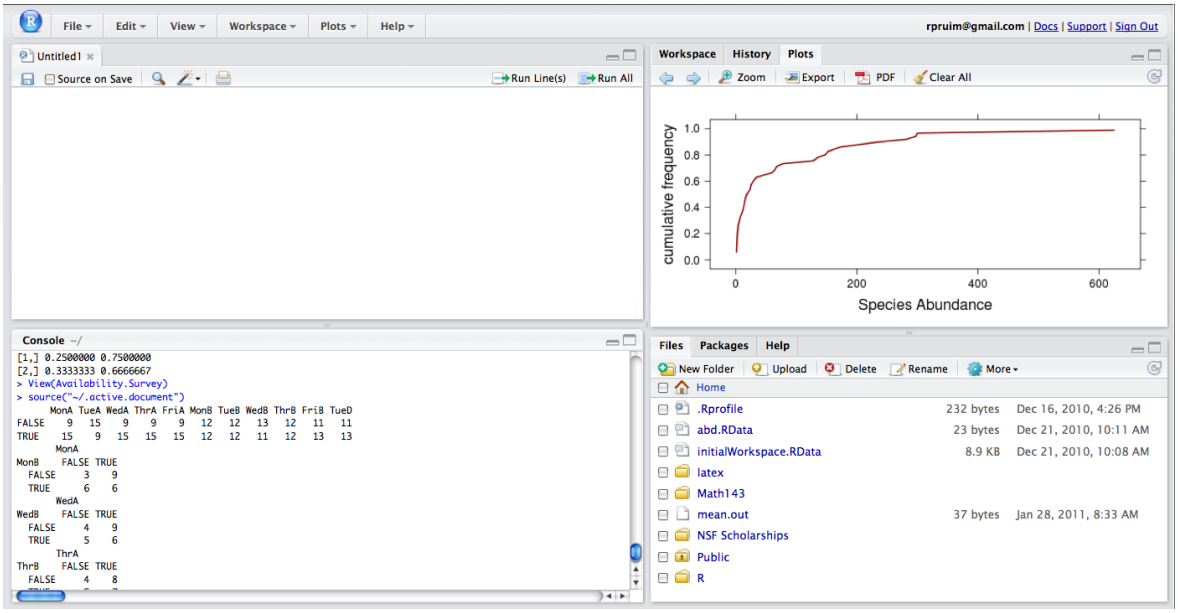


Figure 1.1: Welcome to RStudio.

Notice that RStudio divides its world into four panels. Several of the panels are further subdivided into multiple tabs. RStudio offers the user some control over which panels are located where and which tabs are in which panels, so you initial configuration might not be exactly like the one illustrated here. The console panel is where we type commands that R will execute.

1.2 Using R as a Calculator

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

```
> 5 + 3
[1] 8
> 15.3 * 23.4
[1] 358
> sqrt(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
> product <- 15.3 * 23.4       # <- is assignment operator, same as =
> product
[1] 358
> 15.3 * 23.4 -> newproduct    # -> assigns to the right
> newproduct
[1] 358
```

Once variables are defined, they can be referenced with other operators and functions.

Teaching Tip
It's probably best to settle on using one or the other of the right-to-left assignment operators rather than to switch back and forth. In this document, we will primarily use the arrow operator.

```
> .5 * product                # half of the product
[1] 179
> log(product)                # (natural) log of the product
[1] 5.881
> log10(product)              # base 10 log of the product
[1] 2.554
> log(product, base=2)         # base 2 log of the product
[1] 8.484
```

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

```
> 15.3 * 23.4 -> product; product    # save result and show it
[1] 358
```

Four Things to Know About R

Teaching Tip
To help students get the hang of function arguments, ask them *What information does the computer need to compute this?*

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 (this probably isn't what you want at this point).

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

   If you begin a command and hit the TAB key, RStudio 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.

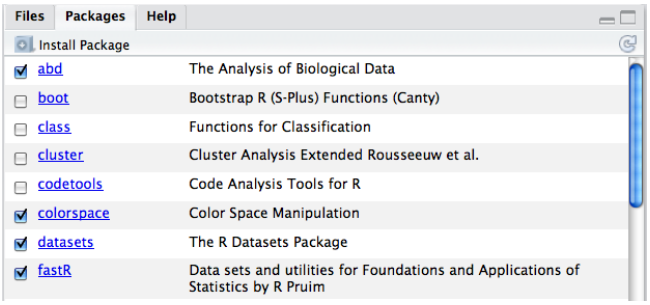
Caution!
Your students will sometimes find themselves in a syntactic hole from which they cannot dig out. Teach them about the ESC key early.

4. If you see a + prompt, it means R is waiting for more input.

   Often this means that you have forgotten a closing parenthesis or made some other syntax error. If you have messed up and just want to get back to the normal plot, hit the escape key and start the command fresh.

1.3 R Packages

In addition to its core features, R provides many more features through a (large) number of packages. To use a package, it must be installed (one time), and loaded (each session). A number of packages are already available in RStudio. The Packages tab in RStudio will show you the list of installed packages and indicate which of these are loaded.



Here are some packages we will use frequently:

- `fastR` (a package with some nice utilities; available on CRAN)
- `lattice` (for graphics; this will always be installed in R)
- `mosaic` (for teaching statistics; this will need to be installed in R from CRAN, see section 1.8.3)
- `Hmisc` (a package with some nice utilities; available on CRAN)

CAUTION!
You should load these packages when working through these notes or you may get different behavior. If you are using the RStudio server via the web, your session will persist even if you logout or switch browsers, so you won't have to do this each time you come back. If you run a stand alone version of R you may.

Since `fastR` requires the others in this list, it suffices to load that one package and others will load automatically. You can do this by checking the appropriate box in the packages tab or at the command line using

```
> library(fastR)
```

or

```
> require(fastR)
```

There are other packages that we use from time to time as well.

You should install `Hmisc` and `mosaic` the first time you use R. Once a package is installed, it is available to be loaded in the current or any future session. You can install these packages by clicking on the "Install Package" button in RStudio and following the directions or by using the following command:

```
> install.packages('Hmisc')               # note the quotation marks
> install.packages('mosaic')              # note the quotation marks
```

Once these are installed, you can load them by checking the box in the Packages tab or by using `require()` (or `library()`)

```
> require(lattice)          # could also use library(lattice)
> require(Hmisc)            # do this one before mosaic
> require(mosaic)           # do this one after Hmisc
```

To speed things up,

- All the packages used here have been pre-installed on dahl and pre-loaded on the loaner laptops. (But we recommend that you give the web-based version of RStudio a try rather than using the local version on the laptops.)
- If you want to install these packages on your own machine, you can get them all in one go using the command

  ```
  > source("http://www.calvin.edu/~rpruim/talks/SC11/SC11-packages.R")
  ```

  This may take a few minutes to complete.

1.4 Getting Help

If something doesn't go quite right, or if you can't remember something, it's good to know where to turn for help. In addition to asking your friends and neighbors, you can use the R help system.

1.4.1 ?

To get help on a specific function or data set, simply precede its name with a `?`:

```
> ?col.whitebg()
```

1.4.2 `apropos()`

If you don't know the exact name of a function, you can give part of the name and R will find all functions that match. Quotation marks are mandatory here.

```
> apropos('hist')            # must include quotes.  single or double.
 [1] "event.history"              "hist"
 [3] "hist.data.frame"            "hist.default"
 [5] "hist.FD"                    "hist.scott"
 [7] "histbackback"               "histochart"
 [9] "histogram"                  "histogram"
[11] "history"                    "histSpike"
[13] "ldahist"                    "loadhistory"
[15] "panel.histogram"            "panel.xhistogram"
[17] "pmfhistogram"               "prepanel.default.histogram"
[19] "savehistory"                "truehist"
[21] "xhistogram"                 "xhistogram"
```

1.4.3 `??` and `help.search()`

If that fails, you can do a broader search using `??` or `help.search()`, which will find matches not only in the names of functions and data sets, but also in the documentation for them.

```
> ??histogram                  # any of these will work
> ??"histogram"
> ??'histogram'
> help.search('histogram')
```

1.4.4 Examples and Demos

Many functions and data sets in R include example code demonstrating typical uses. For example,

```
> example(histogram)
```

Not all package authors are equally skilled at creating examples. Some of the examples are next to useless, others are excellent.

will generate a number of example plots (and provide you with the commands used to create them). Examples such as this are intended to help you learn how specific R functions work. These examples also appear at the end of the documentation for functions and data sets.

The `mosaic` package (and some other packages as well) also includes demos. Demos are bits of R code that can be executed using the `demo()` command with the name of the demo.

To see how demos work, give this a try:

```
> demo(histogram)
```

Demos are intended to illustrate a concept, a method, or some such thing, and are independent of any particular function or data set.

You can get a list of available demos using

```
> demo()                    # all demos
> demo(package='mosaic')    # just demos from mosaic package
```

1.5 Data

1.5.1 Data in Packages

Many packages contain data sets. You can see a list of all data sets in all loaded packages using

```
> data()
```

Typically (provided the author of the package allowed for lazy loading of data) you can use data sets by simply typing their names. But if you have already used that name for something or need to refresh the data after making some changes you no longer want, you can explicitly load the data using the `data()` function with the name of the data set you want.

```
> data(iris)
```

1.5.2 Data Frames

Data sets are usually stored in a special structure called a **data frame**.

Data frames have a 2-dimensional structure.

- Rows correspond to **observational units** (people, animals, plants, or other objects we are collecting data about).
- Columns correspond to **variables** (measurements collected on each observational unit).

We'll talk later about how to get your own data into R. For now we'll use some data that comes with R and is all ready for you to use. The `iris` data frame contains 5 **variables** measured for each of 150 iris plants (the observational units). The `iris` data set is included with the default R installation. (Technically, it is located in a package called `datasets` which is always available.)

There are several ways we can get some idea about what is in the `iris` data frame.

```
> str(iris)
```

```
'data.frame':        150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa    :50
 1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
 Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
 Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
 3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
 Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
```

```
> head(iris)
  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
```

In interactive mode, you can also try

```
> View(iris)
```

to see the data or

```
> ?iris
```

to get the documentation about for the data set.

Access to an individual variable in a data frame uses the `$` operator in the following syntax:

```
> dataframe$variable
```

or

```
> with(dataframe, variable)
```

For example, either of

```
> iris$Sepal.Length
```

or

```
> with(iris, Sepal.Length)
```

shows the contents of the `Sepal.Length` variable in the following format.

```
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1
 [21] 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4 5.1
 [41] 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2
 [61] 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7
 [81] 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
[101] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0
[121] 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9
[141] 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

But this isn't very useful for a large data set. We would prefer to compute numerical or graphical summaries. We'll do that shortly.

CAUTION! Avoid the use of `attach()`.

The `attach()` function in R can be used to make objects within dataframes accessible in R with fewer keystrokes, but we strongly discourage its use, as it often leads to name conflicts. The Google R Style Guide (`http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`) echoes this advice, stating that *The possibilities for creating errors when using attach are numerous. Avoid it.* It is far better to directly access variables using the `\$` syntax or to use the `with()` function.

1.5.3 Using Your Own Data

RStudio will help you import your own data. To do so use the "Import Dataset" button in the Workspace tab. You can load data from text files, from the web, or from google spreadsheets.

Using `read.csv()` and `read.table()`

If you are not using RStudio, or if you want to automate the loading of data from files, instead of using the RStudio menus, you can read files using `read.csv()` or `read.table()` (for white space delimited files). The `mosaic` package includes a function called `read.file()` that uses slightly different default settings and infers whether it should use `read.csv()`, `read.table()`, or `load()` based on the file name.

LOAD()
`load()` is used for opening files that store R objects in 'native' format.

Each of these functions also accepts a URL in place of a file name, which provides an easy way to distribute data via the Internet:

```
> births <- read.table('http://www.calvin.edu/~rpruim/data/births.txt', header=TRUE)
> head(births)        # number of live births in the US each day of 1978.
```

```
    date births datenum dayofyear
1 1/1/78   7701    6575         1
2 1/2/78   7527    6576         2
3 1/3/78   8825    6577         3
4 1/4/78   8859    6578         4
5 1/5/78   9043    6579         5
6 1/6/78   9208    6580         6
```

The `mosaic` package provides `read.file()` which attempts to infer the file type from its extension and then applies `read.csv()`, `read.table()`, or `load()`.

```
> births <- read.file('http://www.calvin.edu/~rpruim/data/births.txt')
```

It also sets a number of defaults, including `header=TRUE`.

Using RStudio Server menus

This is a two step process.

1. Get the file onto the server.

   Upload (in the Files tab) your csv file to the server, where you can create folders and store files in your personal account.

2. Load the data from the server into your R session.

   Now import "from a text file" in the Workspace tab.

In either case, be sure to do the following:

- Choose good variables names.
- Put your variables names in the first row.
- Use each subsequent row for one observational unit.
- Give the resulting data frame a good name.

From Google

In the beta version of RStudio, this was super easy: Just click, select your spreadsheet, choose a name, and you're done. Eventually, this functionality will come to the publicly available server version.

For the time being, in the publicly released version, you need to get Google to tell you the URL for the csv feed of your data (see
http://blog.revolutionanalytics.com/2009/09/how-to-use-a-google-spreadsheet-as-data-in-r.html

for instructions if you don't know how), and then you can read the data using `read.csv()` using that URL in place of the filename.

1.6 Summarizing Data

1.6.1 A Few Numerical Summaries

R includes functions that compute a wide range of numerical and graphical summaries. Most of the numerical summaries already familiar to you have obvious names. Here are a few examples.

```
> mean(iris$Sepal.Length)
```

```
 mean
5.843
```

```
> with(iris, mean(Sepal.Length))
```

```
 mean
5.843
```

```
> # this syntax is provided by the mosaic package for a few frequently used summaries
> mean(Sepal.Length, data=iris)
```

```
[1] 5.843
```

```
> with(iris, median(Sepal.Length))                # or median(iris$Sepal.Length)
```

```
median
   5.8
```

```
> with(iris, quantile(Sepal.Length))              # or quantile(iris$Sepal.Length)
```

```
  0%  25%  50%  75% 100%
 4.3  5.1  5.8  6.4  7.9
```

```
> with(iris, IQR(Sepal.Length))                   # or IQR(iris$Sepal.Length)
```

```
[1] 1.3
```

The `favstats()` function in the `mosaic` package computes several numerical summaries all at once.

```
> require(mosaic)               # if you haven't already loaded the mosaic package
> favstats(iris$Sepal.Length)
```

```
 min  Q1 median  Q3 max  mean     sd   n missing
 4.3 5.1    5.8 6.4 7.9 5.843 0.8281 150       0
```

Here's something a little fancier.

```
> require(Hmisc)              # this should already be loaded from earlier, right?
> summary(Sepal.Length ~ Species, data=iris, fun=favstats)
```

```
Sepal.Length    N=150

+-------+----------+---+----+-----+-------+---+----+-----+------+---+--------+
|       |          |N  |min |Q1   |median |Q3 |max |mean |sd    |n  |missing |
```

```
+-------+----------+---+----+-----+-------+---+----+-----+------+---+--------+
|Species|setosa    | 50|4.3 |4.800|5.0    |5.2|5.8 |5.006|0.3525| 50|0       |
|       |versicolor| 50|4.9 |5.600|5.9    |6.3|7.0 |5.936|0.5162| 50|0       |
|       |virginica | 50|4.9 |6.225|6.5    |6.9|7.9 |6.588|0.6359| 50|0       |
+-------+----------+---+----+-----+-------+---+----+-----+------+---+--------+
|Overall|          |150|4.3 |5.100|5.8    |6.4|7.9 |5.843|0.8281|150|0       |
+-------+----------+---+----+-----+-------+---+----+-----+------+---+--------+
```

The mosaic package also allows the following type of summary for mean, median, standard deviation, variance, and a few other key numerical summaries.

```
> mean( Sepal.Length ~ Species, data=iris )
     Species     S  N Missing
1     setosa 5.006 50       0
2 versicolor 5.936 50       0
3  virginica 6.588 50       0
> median( Sepal.Length ~ Species, data=iris )
     Species   S  N Missing
1     setosa 5.0 50       0
2 versicolor 5.9 50       0
3  virginica 6.5 50       0
> sd( Sepal.Length ~ Species, data=iris )
     Species      S  N Missing
1     setosa 0.3525 50       0
2 versicolor 0.5162 50       0
3  virginica 0.6359 50       0
```

1.6.2 Lattice Graphics

There are several ways to make graphs in R. One approach is a system called `lattice` graphics. The first step for using `lattice` is to load the `lattice` package using the check box in the Packages tab or using the following command:

```
> require(lattice)
```

`lattice` plots make use of a **formula interface**:

```
> plotname( y ~ x | z, data=dataname, groups=grouping_variable, ...)
```

- Here are the names of several `lattice` plots:
  - `histogram` (for histograms)
  - `bwplot` (for boxplots)
  - `xyplot` (for scatter plots)
  - `qqmath` (for quantile-quantile plots)
- `x` is the name of the variable that is plotted along the horizontal (x) axis.
- `y` is the name of the variable that is plotted along the vertical (y) axis. (For some plots, this slot is empty because R computes these values from the values of `x`.)
- `z` is a conditioning variable used to split the plot into multiple subplots called **panels**.
- `grouping_variable` is used to display different groups differently (different colors or symbols, for example) within the same panel.
- `...` There are many additional arguments to these functions that let you control just how the plots look. (But we'll focus on the basics for now.)

1.6.3 Histograms: `histogram()`

Let's switch to a more interesting data set from the Health Evaluation and Linkage to Primary Care study. The HELP study was a clinical trial for adult inpatients recruited from a detoxification unit. Patients with no primary care physician were randomized to receive a multidisciplinary assessment and a brief motivational intervention or usual care, with the goal of linking them to primary medical care. You can find out more about this data using R's help

```
> ?HELP
```

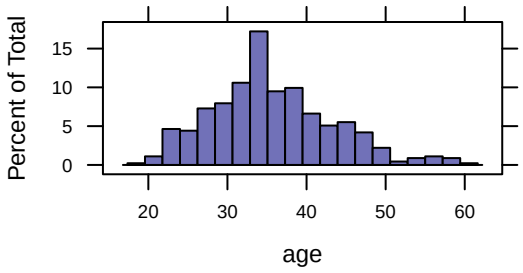
Histograms display a distribution using the important idea the

$$\text{AREA} = \text{relative frequency}$$

So where there is more area, there is more data. For a historam, rectangles are used to indicate how much data are in each of several "bins". The result is a picture that shows a rough "shape" of the distribution.

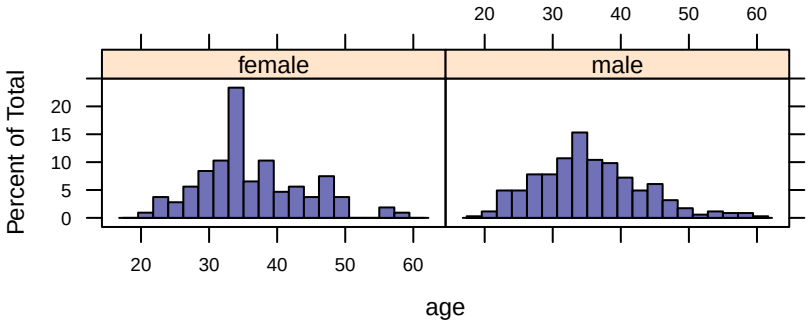
The y component of the formula is empty since we let R compute the heights of the bars for us.

```
> histogram(~ age, data=HELP, n=20)       # n= 20 gives approx. 20 bars
```



We can use a conditional variable to give us separate histograms for each sex.

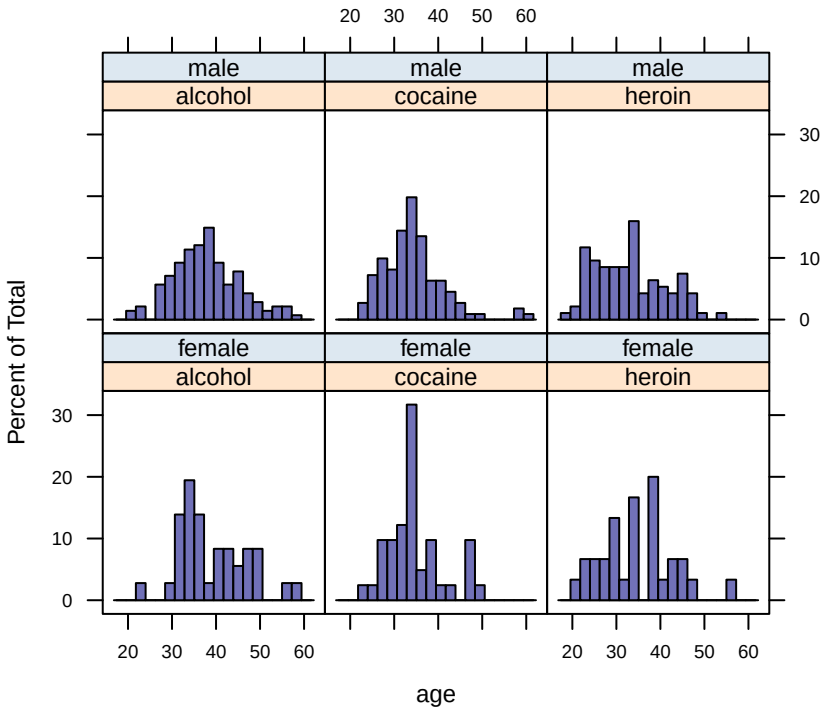
```
> histogram(~ age | sex, data=HELP, n=20)
```



In lattice lingo, the two subplots are called panels and the labels at the top are called strips. (Strips can be placed on the left side if you prefer.)

We can even condition on two things at once:

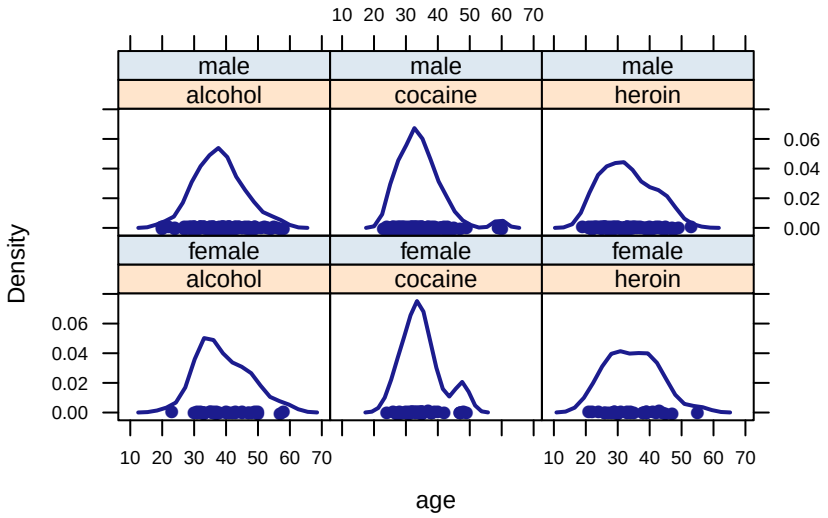
```
> histogram(~ age | substance + sex, data=HELP, n=20)
```



1.6.4 Density plots `densityplot()`

Density plots are smoother versions of histograms.

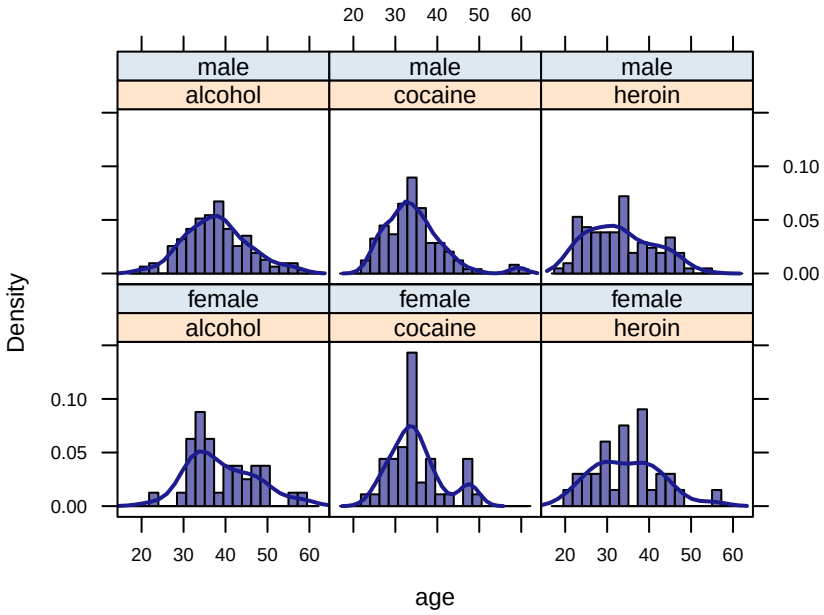
```
> densityplot(~ age | substance + sex, data=HELP, n=20)
```



Digging Deeper
There are several optional arguments to the `densityplot()` function that can be used to control the type and degree of smoothing used in these plots. The general method is called kernel density estimation.

If we want to get really fancy, we can do both at once:

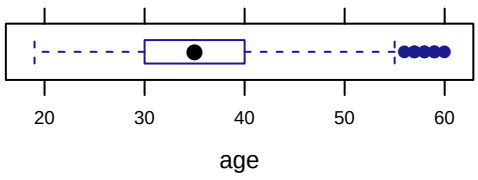
```
> histogram(~age|substance+sex,data=HELP,n=20, type='density',
                          panel=function(...){ panel.histogram(...); panel.densityplot(...)}
        )
```



1.6.5 Boxplots: `bwplot()`

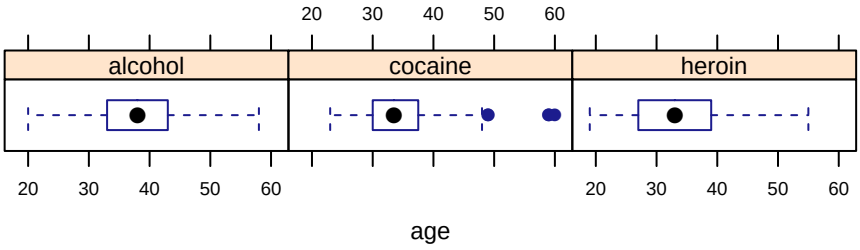
Boxplots are made pretty much the same way as histograms:

```
> bwplot(~ age, data=HELP)
```



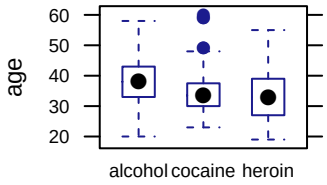
We can use conditioning as we did for histograms:

```
> bwplot(~ age | substance, data=HELP)
```



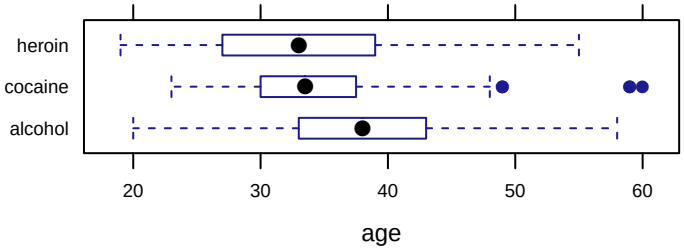
But there are better ways to do this.

```
> bwplot(age ~ substance, data=HELP)
```



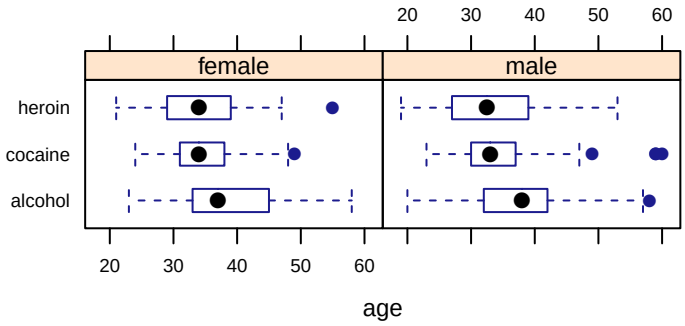
This is improved, but the species names run into each other. We could fix that run-together text by using abbreviated names or rotating the labels 45 or 90 degrees. Instead of those solutions, we can also just reverse the roles of the horizontal and vertical axes.

```
> bwplot(substance ~ age, data=HELP)
```



We can combine this with conditioning if we like:

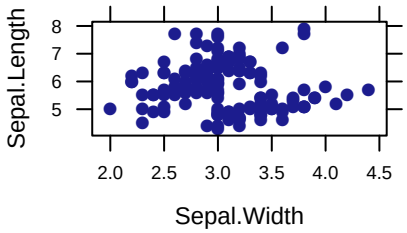
```
> bwplot(substance ~ age | sex, data=HELP)
```



1.6.6 Scatterplots: `xyplot()`

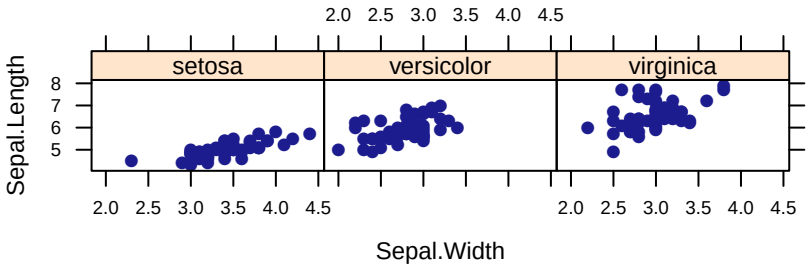
Scatterplots are made with `xyplot()`. The formula interface is very natural for this. Just remember that the "y variable" comes first. (Its label is also farther left on the plot, if that helps you remember.)

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



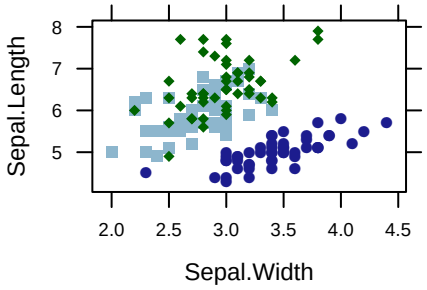
Again, we can use conditioning to make a panel for each species.

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



Even better (for this example), we can use the `groups` argument to indicate the different species using different symbols on the same panel.

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



1.6.7 Saving Your Plots

There are several ways to save plots, but the easiest is probably the following:

1. In the Plots tab, click the "Export" button.
2. Copy the image to the clipboard using right click.
3. Go to your Word document and paste in the image.
4. Resize or reposition your image in Word as needed.

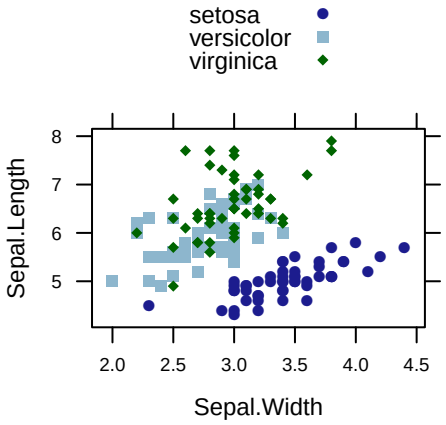
1.6.8 A Few Bells and Whistles

There are lots of arguments that control how these plots look. Here are just a few examples.

auto.key

It would be useful to have a legend for the previous plot. `auto.key=TRUE` turns on a simple legend. (There are ways to have more control, if you need it.)

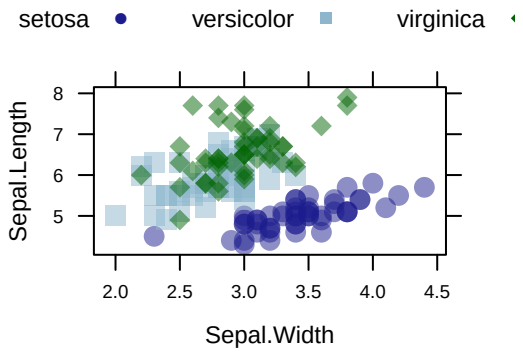
```
> xyplot(Sepal.Length ~ Sepal.Width, groups=Species, data=iris,
         auto.key=TRUE)
```



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.

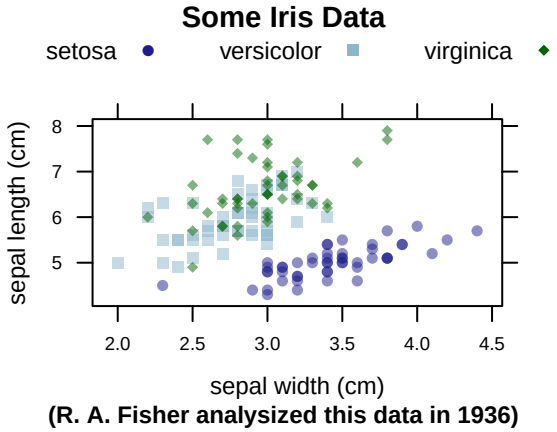
```
> xyplot(Sepal.Length ~ Sepal.Width, groups=Species, data=iris,
         auto.key=list(columns=3),
         alpha=.5,
         cex=1.3)
```



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 analysized this data in 1936)",
         xlab="sepal width (cm)",
         ylab="sepal length (cm)",
         alpha=.5,
         auto.key=list(columns=3))
```



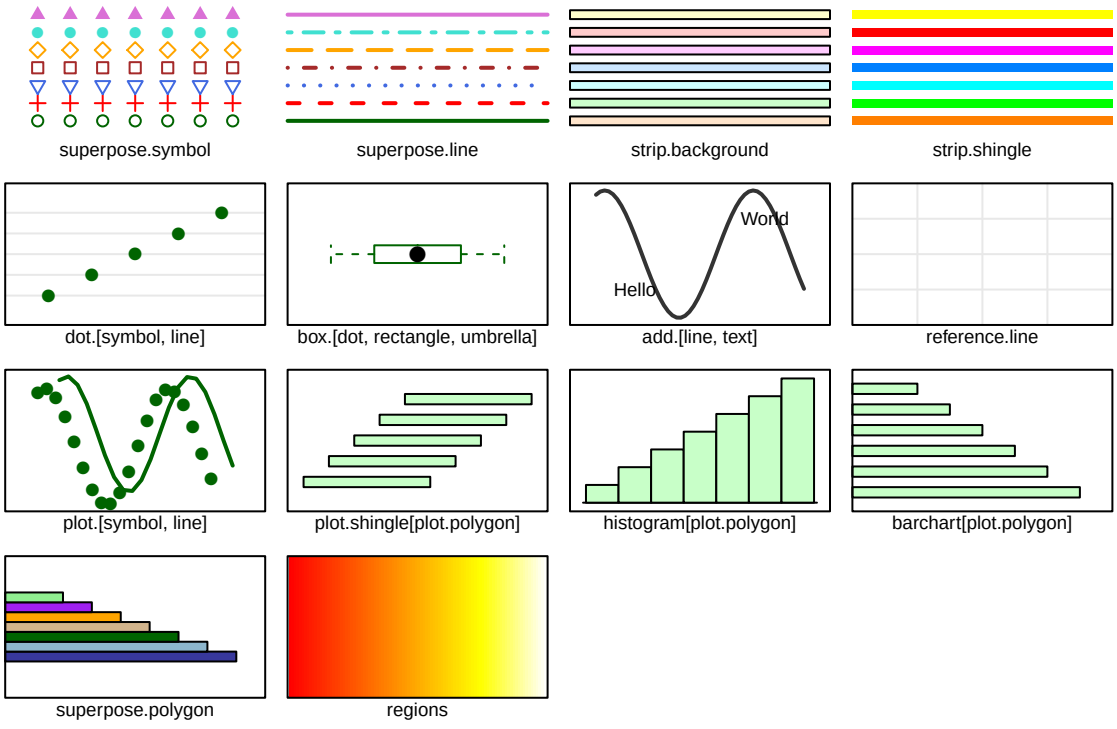
trellis.par.set()

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

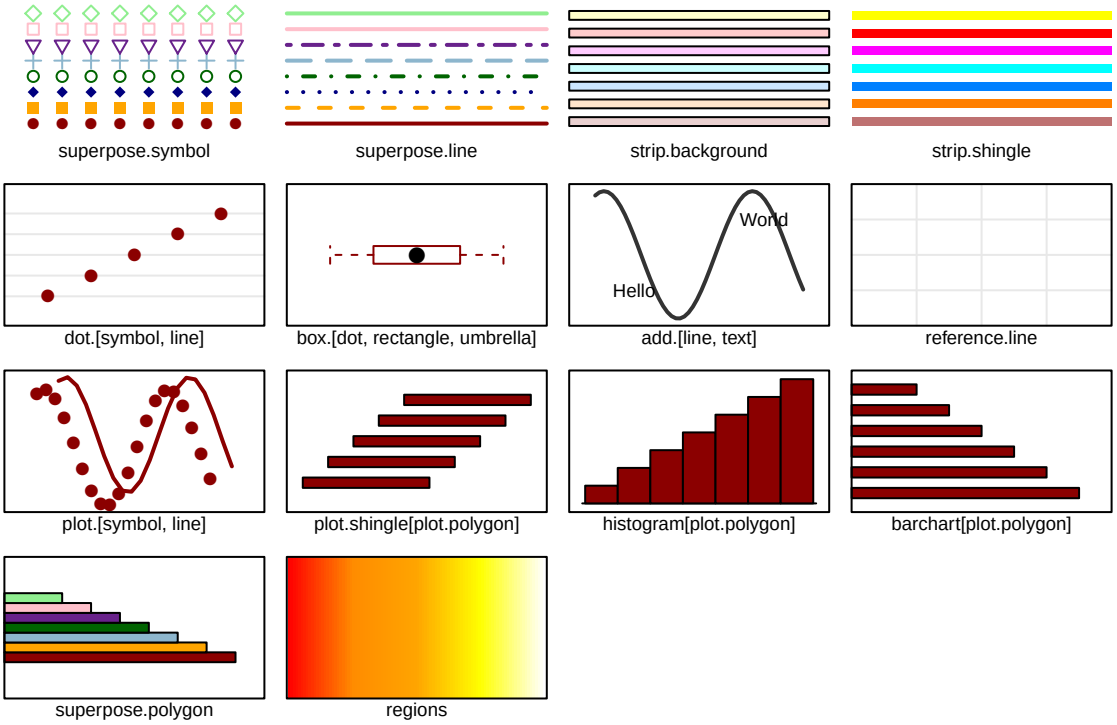
```
> trellis.par.set(fontsize=list(text=7))    # base size for text is 7 point
```

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 theme looks like.

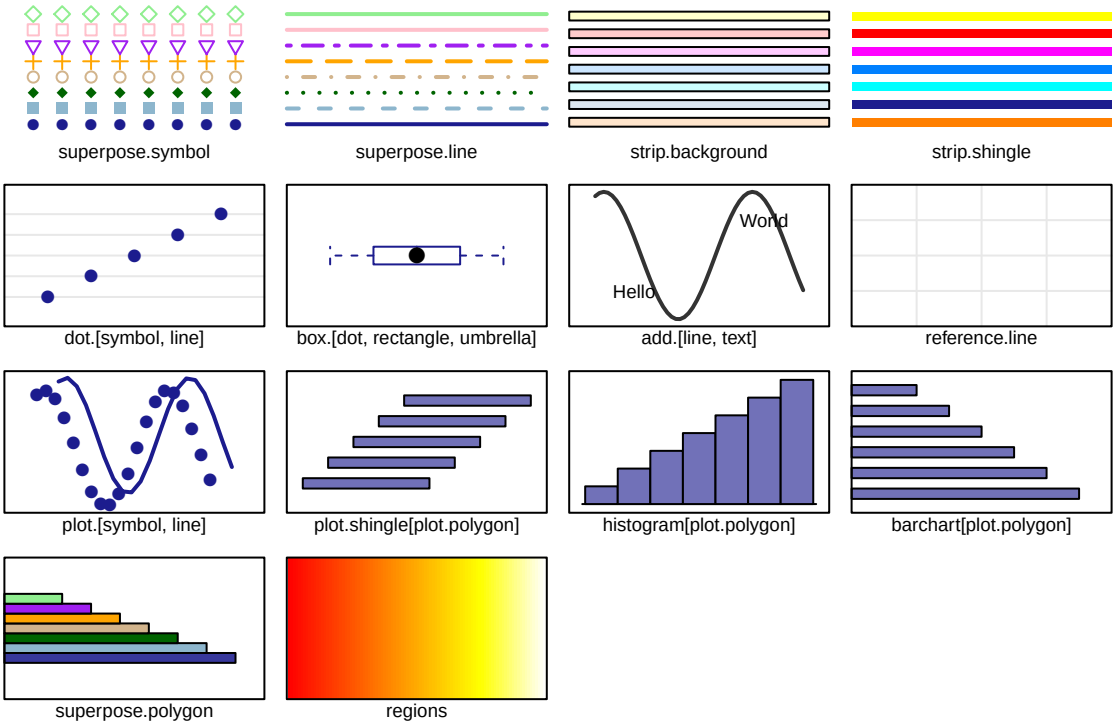
```
> trellis.par.set(theme=col.whitebg())      # a theme in the lattice package
> show.settings()
```




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



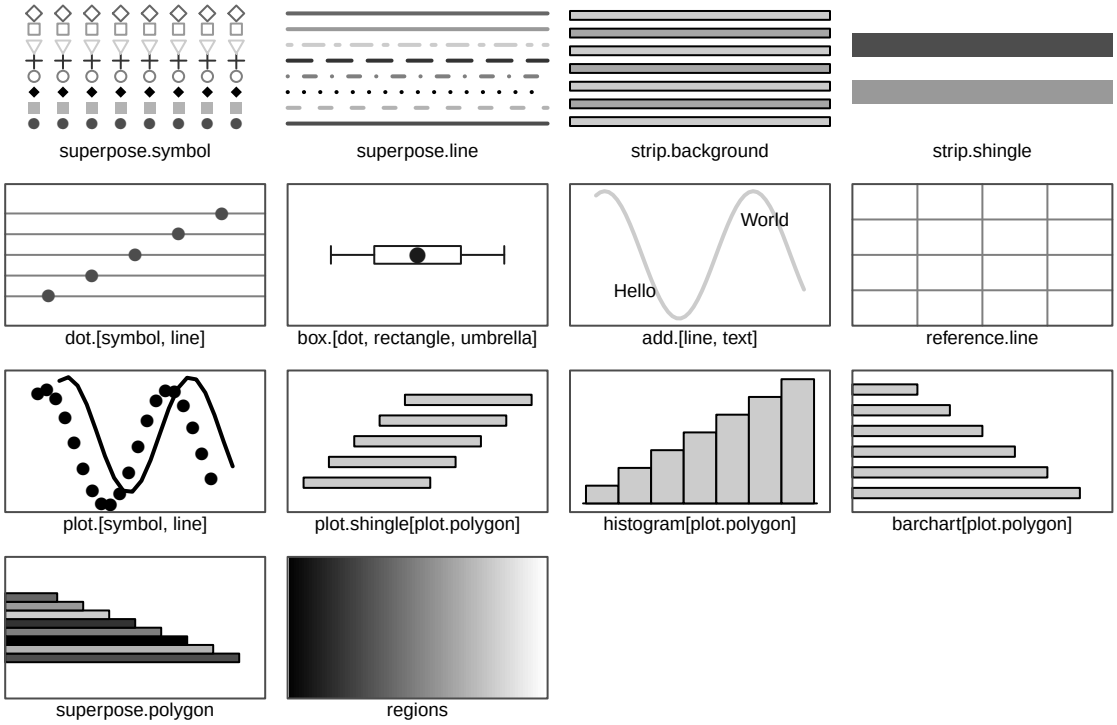

```
> trellis.par.set(theme=col.mosaic())        # a theme in the mosaic package
> show.settings()
```




```
> trellis.par.set(theme=col.mosaic(bw=TRUE)) # black and white version of previous theme
> show.settings()
```

SUGGESTION BOX
Do you have a great eye for colors? Help us design other lattice themes.

DIGGING DEEPER
The RColorBrewer package provides several palettes of colors that are highly distinguishable




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

1.6.9 Tabulating Categorical Data

The Current Population Survey (CPS) is used to supplement census information between census years. These CPS data frame consist of a random sample of persons from the CPS, with information on wages and other characteristics of the workers, including sex, number of years of education, years of work experience, occupational status, region of residence and union membership.

```
> head(CPS,3)
  wage educ race sex hispanic south married exper union age sector
1  9.0   10    W   M       NH    NS Married    27   Not  43  const
2  5.5   12    W   M       NH    NS Married    20   Not  38  sales
3  3.8   12    W   F       NH    NS  Single     4   Not  22  sales
```

Making Frequency and Contingency Tables with `xtabs()`

Categorical variables are often summarized in a table. R can make a table for a categorical variable using `xtabs()`.

```
> xtabs(~ race, CPS)
race
 NW   W
 67 467
> xtabs(~ sector, CPS)
sector
clerical    const    manag    manuf    other     prof    sales  service
      97       20       55       68       68      105       38       83
```

Alternatively, we can use `table()`, `proptable()`, and `perctable()` to make tables of counts, proportions, or percentages.

```
> with(CPS, table(race))
```

```
race
 NW   W
 67 467
```

```
> with(CPS, proptable(sector))
```

```
sector
clerical    const    manag    manuf    other     prof    sales  service
 0.18165  0.03745  0.10300  0.12734  0.12734  0.19663  0.07116  0.15543
```

```
> with(CPS, perctable(sector))
```

```
sector
clerical    const    manag    manuf    other     prof    sales  service
  18.165    3.745   10.300   12.734   12.734   19.663    7.116   15.543
```

We can make a **cross-table** (also called a **contingency table** or a **two-way table**) summarizing this data with `xtabs()`. This is often a more useful view of data with two categorical variables.

```
> xtabs(~ race + sector, CPS)
```

```
    sector
race clerical const manag manuf other prof sales service
  NW       15     3     6    11     5    7     3      17
  W        82    17    49    57    63   98    35      66
```

Entering Tables by Hand

Because categorical data is so easy to summarize in a table, often the frequency or contingency tables are given instead. You can enter these tables manually as follows:

```
> myrace <- c( NW=67, W=467 )       # c for combine or concatenate
> myrace
```

```
 NW   W
 67 467
```

```
> mycrosstable <- rbind(               # bind row-wise
   NW = c(clerical=15, const=3, manag=6,  manuf=11, other=5,  prof=7, sales=3, service=17),
   W  = c(82,17,49,57,63,98,35,66)    # no need to repeat the column names
                         )
> mycrosstable
```

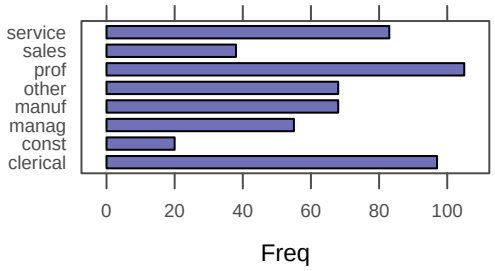
```
   clerical const manag manuf other prof sales service
NW       15     3     6    11     5    7     3      17
W        82    17    49    57    63   98    35      66
```

Replacing `rbind()` with `cbind()` will allow you to give the data column-wise instead.

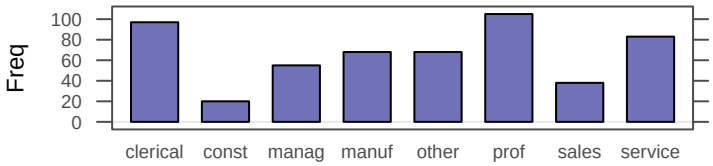
1.6.10 Graphing Categorical Data

The `lattice` function `barchart()` can display these tables as barcharts.

```
> barchart(xtabs(~ sector, CPS))
```

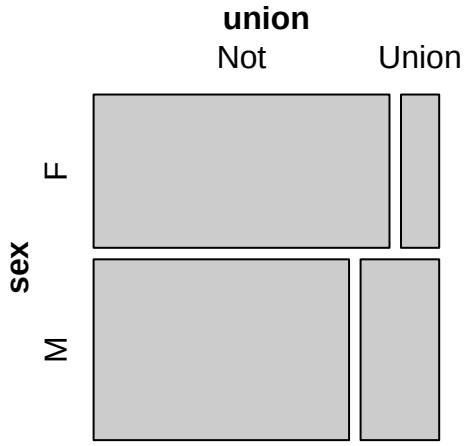



```
> barchart(xtabs(~ sector, CPS), horizontal=FALSE)  # vertical bars
```



Just as bar charts are used to display the distribution of one categorical variable, mosaic plots can do the same for cross tables. `mosaic()` (from the `vcd` package) is not a `lattice` plot, but it does use a similar formula interface.

```
> require(vcd)                          # load the visualizing categorical data package
> mosaic(~ sex + union, CPS)
```



> Caution!
> The `mosaic()` function has nothing to do with the `mosaic` package, they just happen to share the same name.

Alternatively, we can send `mosaic()` the output of `xtabs()`:

```
> mosaic(xtabs(~ sex + union, CPS))  # non-whites are more likely to be unionized
```

Neither `mosaic()` nor the similar `mosaicplot()` are as clever as one could hope. In particular, without some extra customization, both tend to look bad if the levels of the variables have long names. `mosaic()` plots also always stay square.

Or we can send our own hand-made table (although the output isn't quite as nice without some extra effort we won't discuss just now):

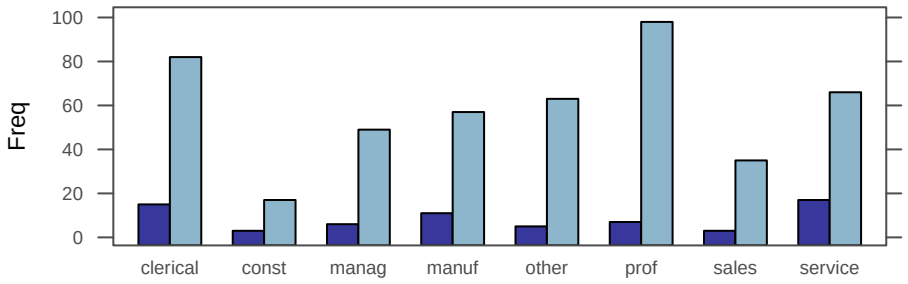
```
> mosaic(mycrosstable)
```

Barcharts can also be used to display two-way tables. First we convert the cross-table to a data frame. Then we can use this data frame for plotting.

```
> cps <- as.data.frame(xtabs(~ sector + race, data=CPS)); cps
```

```
     sector race Freq
1  clerical   NW   15
2     const   NW    3
3     manag   NW    6
4     manuf   NW   11
5     other   NW    5
6      prof   NW    7
7     sales   NW    3
8   service   NW   17
9  clerical    W   82
10    const    W   17
11    manag    W   49
12    manuf    W   57
13    other    W   63
14     prof    W   98
15    sales    W   35
16  service    W   66
```

```
> barchart(Freq ~ sector, groups=race, data=cps)
```



1.7 Additional Notes on R Syntax

1.7.1 Text and Quotation Marks

For the most part, text in R must be enclosed in either single or double quotations. It usually doesn't matter which you use, unless you want one or the other type of quotation mark *inside* your text. Then you should use the other type of quotation mark to mark the beginning and the end.

```
> text1 <- "Mary didn't come"           # apostrophe inside requires double quotes around text
> text2 <- 'Do you use "scare quotes"?'  # this time we flip things around
```

If you omit quotes, you will often see error messages telling you that R can't find an object because R will look for a function, data set or other object with that name instead of treating your text as text.

```
> text3 <- blah
Error: object 'blah' not found
```

1.7.2 Functions

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 `col.whitebg()`) have no arguments, but you still need the parentheses.
- Most arguments have names, but you don't need to use the names *if you give the arguments in the correct order.*

  If you use names, you can give the arguments out of order. The following do the same thing,

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

  But these do not work

  ```
  > xyplot(Sepal.Length ~ Sepal.Width, Species, iris)
  > xyplot(Sepal.Length ~ Sepal.Width, iris, Species)
  ```

  The first fails because the second argument is `data`, so `iris` needs to be in the second position if it is not named. The second fails because `groups` is not the third argument. (There are many other arguments between `data` and `groups` .) The documentation for functions shows the correct order of arguments.
- Typically, we will not use names for the first argument or two (these tend to be very important arguments that have to be there) but will use names for the rest (these are often optional arguments that can be present or not, depending on whether we want the default behavior or something special).

1.8 Installing R

1.8.1 RStudio in the cloud

Our primary version of R will be the online version of RStudio. You should have an RStudio account at `http://beta.rstudio.org/` using your Gmail address. RStudio is a brand new (and quite nice) interface to R that runs in a web browser. This has the advantage that you don't have to install or configure anything. Just login and you are good to go. Futhermore, RStudio will "remember" what you were doing so that each time you login (even on a different machine) you can pick up right where you left off. This is "R in the cloud" and works a bit like GoogleDocs for R.

If you find bugs or have suggestions for RStudio, let us know. It is in rapid development at the moment, and we can pass along your feedback to the developers.

This should be all you need for this course. But if you prefer to have a stand-alone version (because you study somewhere without an internet connection, or have data that can't be loaded into the cloud, for example), read on.

1.8.2 RStudio on Your Desktop/Laptop

There is also a stand-alone version of the RStudio environment that you can install on your desktop or laptop machine. This can be downloaded from `http://www.rstudio.org/`. This assumes that you have a version of R installed on your computer (see below for instructions to download this from CRAN).

1.8.3 Getting R from CRAN

CRAN is the Comprehensive R Archive Network (`http://cran.r-project.org/`). You can download free versions of R for PC, Mac, and Linux from CRAN. (If you use the RStudio stand-alone version, you also need to install R this way first.) All the instructions for downloading and installing are on CRAN. Just follow the appropriate instructions for your platform.

1.8.4 RStudio in the cloud on your own server

At present, we are using a beta version of RStudio running on their servers. It is also possible to install this on servers at your institution. This will almost certainly require a discussion with your administrators, but may be worthwhile to facilitate student use in this manner.

1.9 R Examples

The commands below are illustrated with the data sets `iris` and `CPS`. To apply these in other situations, you will need to substitute the name of your data frame and the variables in it.

`answer <- 42`	Store the value 42 in a variable named `answer`.
`log(123); log10(123); sqrt(123)`	Take natural logarithm, base 10 logarithm, or square root of 123.
`x <- c(1,2,3)`	Make a variable containing values 1, 2, and 3 (in that order).
`data(iris)`	(Re)load the data set `iris`.
`summary(iris$Sepal.Length)`	Summarize the distribution of the `Sepal.Length` variable in the `iris` data frame.
`summary(iris)`	Summarize each variable in the `iris` data frame.
`str(iris)`	A different way to summarize the `iris` data frame.
`head(iris)`	First few rows of the data frame `iris`.
`require(Hmisc)` `require(abd)`	Load packages. (This can also be done by checking boxes in the Packages tab.)
`summary(Sepal.Length~Species,data=iris,fun=favstats)`	Compute favorite statistics of `Sepal.Length` for each `Species`. [requires `Hmisc`]
`histogram(~Sepal.Length|Species, iris)`	Histogram of `Sepal.Length` conditioned on `Species`.
`bwplot(Sepal.Length~Species, iris)`	Boxplot of `Sepal.Length` conditioned on `Species`.
`xyplot(Sepal.Length~Sepal.Width|Species, iris)`	Scatterplot of `Sepal.Length` by `Sepal.Width` with separate panels for each `Species`.
`xtabs(~sector, CPS)`	Frequency table of the variable `sector`.
`barchart(xtabs(~sector, CPS))`	Make a barchart from the table.
`xtabs(~sector + race, CPS)`	Cross tabulation of `sector` and `race`.
`mosaic(~sector + race, CPS)`	Make a mosaic plot.
`xtData <- as.data.frame( xtabs(~sector + race, Trematodes) )`	Save cross table information as `xtData`.
`barchart(Freq~sector, data=xtData, groups=race)`	Use `xtData` to make a segmented bar chart.
`sum(x)`; `mean(x)`; `median(x)`; `var(x)`; `sd(x)`; `quantile(x)`	Sum, mean, median, variance, standard deviation, quantiles of `x`.

1.10 Exercises

1.1 Calculate the natural logarithm (log base e) and base 10 logarithm of 12,345.

What happens if you leave the comma in this number?

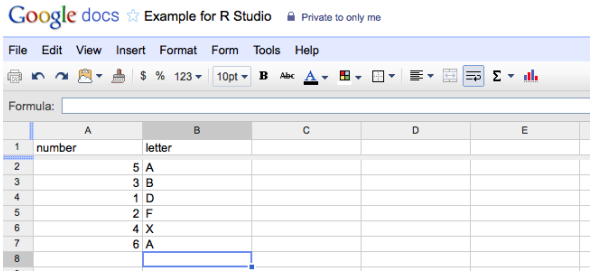
```
> log(12,345)
[1] 0.4252
```

1.2 Install and load the `mosaic` package. Make sure `lattice` is also loaded (no need to install it, it is already installed).

Here are some other packages you may like to install as well.

- `Hmisc` (Frank Harrell's miscellaneous utilities),
- `vcd` (visualizing categorical data),
- `fastR` (*Foundations and Applications of Statistics*), and
- `abd` (*Analysis of Biological Data*).

1.3 Enter the following small data set in an Excel or Google spreadsheet and import the data into RStudio.



You can import directly from Google. From Excel, save the file as a csv and import that (as a text file) into RStudio. Name the data frame `JunkData`.

1.4 What is the average (mean) *width* of the sepals in the `iris` data set?

1.5 Determine the average (mean) sepal width for each of the three species in the `iris` data set.

1.6 The `Jordan8687` data set (in the `fastR` package) contains the number of points Michael Jordan scored in each game of the 1986–87 season.

a) Make a histogram of this data. Add an appropriate title.

b) How would you describe the shape of the distribution?

c) In approximately what percentage of his games, did Michael Jordan score less than 20 points? More than 50? (You may want to add `breaks=seq(0,70,by=5)` to your command to neaten up the bins.)

1.7 Cuckoos lay their eggs in the nests of other birds. Is the size of cuckoo eggs different in different host species nests? The `cuckoo` data set (in `fastR`) contains data from a study attempting to answer this question.

a) When were these data collected? (Use `?cuckoo` to get information about the data set.)

b) What are the units on the length measurements?

c) Make side-by-side boxplots of the length of the eggs by species.

d) Calculate the mean length of the eggs for each host species.

e) What do you think? Does it look like the size is differs among the different host species? Refer to your R output as you answer this question. (We'll learn formal methods to investigate this later in the semester.)

1.8 The `Utilities2` data set in the `mosaic` package contains a number of variables about the utilities bills at a residence in Minnesota over a number of years. Since the number of days in a billing cycle varies from month to month, variables like `gasbillpday` (`elecbillpday`, etc.) contain the gas bill (electric bill, etc.) divided by the number of days in the billing cycle.

a) Make a scatter plot of `gasbillpday` vs. `monthsSinceY2K` using the command

```
> xyplot(gasbillpday ~ monthsSinceY2K, data=Utilities2, type='l')   # the letter l
```

What pattern(s) do you see?

b) What does `type='l'` do? Make your plot with and without it. Which is easier to read in this situation?

c) What happens if we replace `type='l'` with `type='b'`?

d) Make a scatter plot of `gasbillpday` by `month`. What do you notice?

e) Make side-by-side boxplots of `gasbillpday` by `month` using the `Utilities2` data frame. What do you notice?

Your first try probably won't give you what you expect. The reason is that month is coded using numbers, so R treats it as numerical data. We want to treat it as categorical data. To do this in R use `factor(month)` in place of `month`. R calls categorical data a **factor**.

f) Make any other plot you like using this data. Include both a copy of your plot and a discussion of what you can learn from it.

1.9 The table below is from a study of nighttime lighting in infancy and eyesight (later in life).

	no myopia	myopia	high myopia
darkness	155	15	2
nightlight	153	72	7
full light	34	36	3

a) Recreate the table in RStudio.

b) What percent of the subjects slept with a nightlight as infants?

There are several ways to do this. You could use R as a calculator to do the arithmetic. You can save some typing if you use the function `prop.table()`. See `?prop.table` for documentation. If you just want row and column totals added to the table, see `mar_table()` in the `vcd` package.

c) Make a mosaic plot for this data. What does this plot reveal?

2 Getting Interactive with manipulate

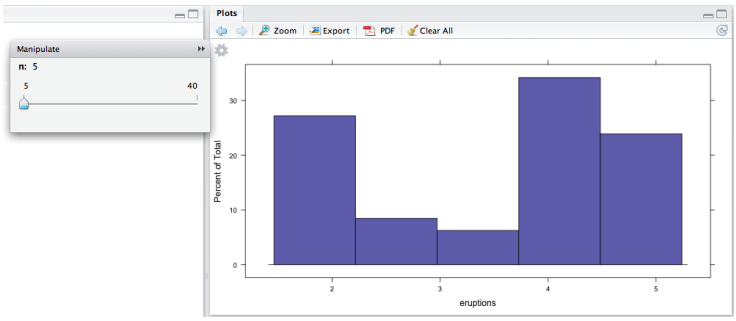
One very attractive feature of RStudio is the `manipulate()` function, which can allow the creation of a set of controls (such as `slider()`, `picker()` or `checkbox()`) that can be used to dynamically change values within the expression. When a value is changed using these controls, the expression is automatically re-executed and redrawn. This can be used to quickly prototype a number of activities and demos as part of a statistics lecture.

2.1 Simple Things

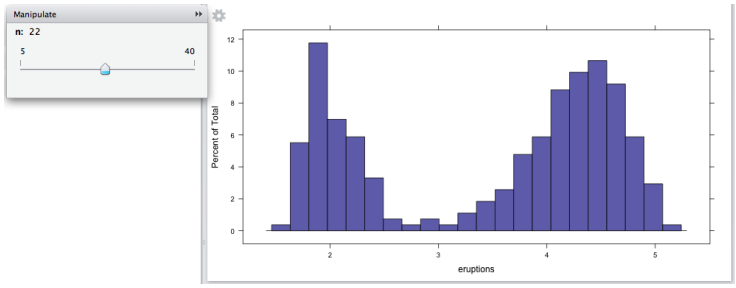
2.1.1 Sliders

```
> if(require(manipulate)) {
   manipulate(
         histogram( ~ eruptions, data=faithful, n=n),
         n = slider(5,40)
   )
 }
```

This generates a plot along with a slider ranging from 5 bins to 40.

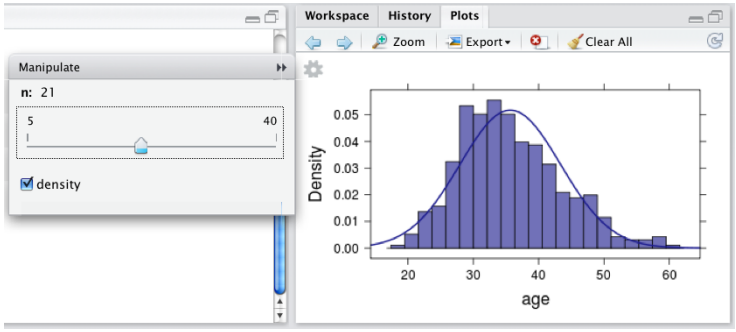


When the slider is changed, we see a clearer view of the eruptions of Old Faithful.



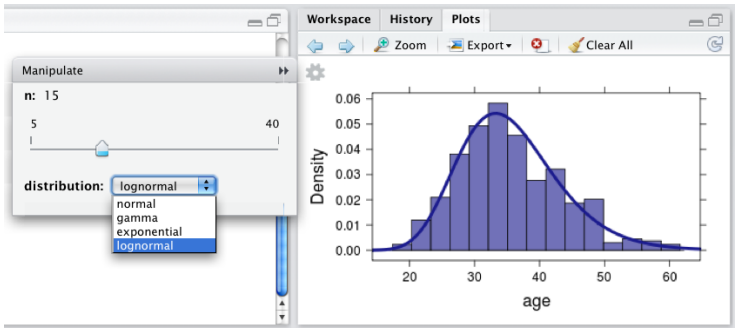
2.1.2 Check Boxes

```
> if(require(manipulate)) {
   manipulate(
          xhistogram( ~ age, data=HELP, n=n, density=density),
          n = slider(5,40),
          density = checkbox()
   )
 }
```



2.1.3 Drop-down Menus

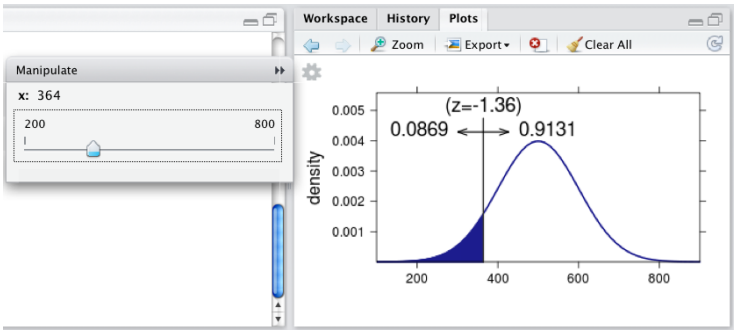
```
> if(require(manipulate)) {
   manipulate(
          xhistogram( ~ age, data=HELP, n=n, fit=distribution, dlwd=4),
          n = slider(5,40),
          distribution =
                 picker('normal', 'gamma', 'exponential', 'lognormal',
                        label="distribution")
   )
 }
```



A slightly fancier version of this is provided as `mhistogram()` in the `mosaic` package.

2.1.4 Visualizing Normal Distributions

```
> if(require(manipulate)) {
   manipulate( xpnorm( x, 500, 100 ), x = slider(200,800) )
 }
```



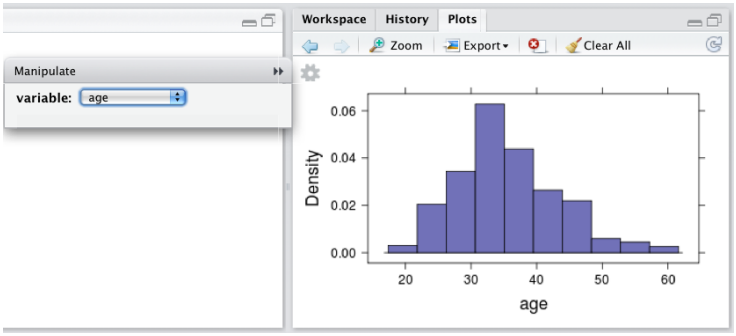
Exercises

2.1 The following code makes a scatterplot with separate symbols for each sex.

```
> xyplot(cesd ~ age, data=HELP, groups=sex)
```

Build a `manipulate` example that allows you to turn the grouping on and off with a checkbox.

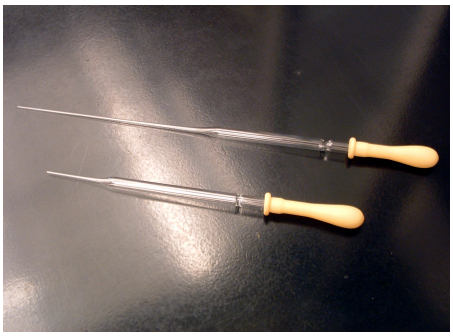
2.2 Build a `manipulate` example that uses a picker to select from a number of variables to make a plot for. Here's an example with a histogram:

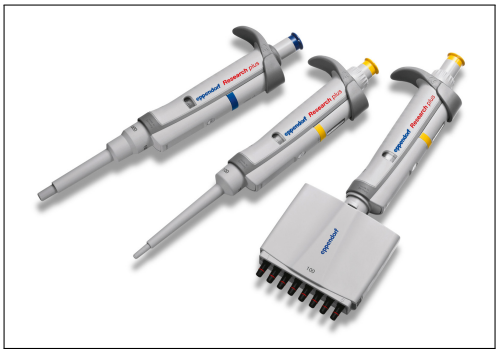


2.3 Design your own interactive demonstration idea and implement it using RStudio `manipulate` tools.

3 A Crash Course in Statistics for Biologists (and Their Friends)

3.1 Why Use R?





Modern statistics is done with statistical computing tools. There are many possibilities.[1] R offers the following set of advantages:

1. R is free and available on any platform (Mac, PC, Linux, etc.) and also, via RStudio, in a web browser.

   This means students have access to R whenever and wherever they need it.

2. R is powerful – you won't outgrow it.

   If one goal is to prepare students for future research, R will grow with them.

3. R produces excellent quality graphics.

   R produces publication quality graphics. Via the `lattice` package, a wide range of useful plots are easy to produce. For those willing to learn a bit more, plots can be customized to your heart's contents.

[1] Excel is *not* among them. Consider this description of Excel 2007 from [MH08]: "Excel 2007, like its predecessors, fails a standard set of intermediate-level accuracy tests in three areas: statistical distributions, random number generation, and estimation. Additional errors in specific Excel procedures are discussed. Microsoft's continuing inability to correctly fix errors is discussed. No statistical procedure in Excel should be used until Microsoft documents that the procedure is correct; it is not safe to assume that Microsoft Excel's statistical procedures give the correct answer. Persons who wish to conduct statistical analyses should use some other package."
Most statisticians will find it challenging to take your science seriously if you did your analysis in Excel.

4. R helps you think about data in ways that are useful for statistical analysis.

   As with most true statistical packages, R prefers data arranged with variables in columns and observational units in rows. Being able to identify each of these is a crucial pre-requisite to understanding statistical procedures.

5. R promotes reproducible research.

   R commands provide an exact record of how an analysis was done. Commands can be edited, rerun, commented, shared, etc.

6. R is up-to-date.

   Many new analysis methods appear first in R.

7. There are many packages available that automate particular tasks.

   The CRAN (Comprehensive R Archive Network) repository contains more than 3000 packages that provide a wide range of additional capabilities (including many with biological applications.)

   The Bioconductor repository (`http://www.bioconductor.org/`) contains nearly 500 additional packages that focus on biological and bioinformatics applications.

8. R can be combined with other tools.

   R can be used within programming languages (like Python) or in scripting environments to automate data analysis pipelines.

9. R is popular – including among biologists.

   R is becoming increasingly popular among biologists, especially among those doing work in genetics and genomics. R is very widely used in graduate programs and academic research, and is gaining market share in industry as well.

10. R provides a gentle introduction to general computation.

    Although R can be used "one command at a time" (that's the way we will be using it here, for the most part), R is a full featured programming language.

11. The best is yet to come.

    A number of projects are underway that promise to further improve on R, including

    - RStudio and its `manipulate` features
    - Chris Wild's new graphical interface (only videos available at this point)
    - `mosaic` (beta) and `mosaicManip` (coming soon) packages from Project Mosaic

3.2 Computational Statistics: Where Are All The Formulas?

You perhaps remember an introduction to statistics that focussed on memorizing a list of formulas and figuring out some way to remember which one to use in which situation. Here we present a different approach based on some key ideas.

1. Randomization

   Randomization lies at the heart of statistics no matter how it is done: random samples, random assignment to treatment groups, etc.

2. Simulation

   With the availability of modern computational tools, we can study randomization by simulation. This has the advantage of emphasizing the core logic behind statistical inference and avoids (or at least reduces) the need to learn many formulas. Furthermore, for many modern statistical procedures, this is the only way they can be done.

   The familiar, traditional statistical methods are usually approximations to randomization methods.[2]

3. Tabulation and Visualization

   Tabulation and Visualization are in some sense two sides of the same coin. Randomness means things do not turn out the same way every time. Instead there is a **distribution** of outcomes. Inferences are drawn by considering these distributions, which can be tabulated and visualized.

In one sense this approach is not new. It is the approach that motivated people like Fisher in the early part of the last century. But because modern computational tools were not available at that time, explicit formulas (usually approximations) were needed in order to perform statistical calculations efficiently.

3.3 Three Illustrative Examples

3.3.1 The Lady Tasting Tea

This section is a slightly modified version of a handout R. Pruim has given Intro Biostats students on Day 1 after going through the activity as a class discussion.

There is a famous story about a lady who claimed that tea with milk tasted different depending on whether the milk was added to the tea or the tea added to the milk. The story is famous because of the setting in which she made this claim. She was attending a party in Cambridge, England, in the 1920s. Also in attendance were a number of university dons and their wives. The scientists in attendance scoffed at the woman and her claim. What, after all, could be the difference?

All the scientists but one, that is. Rather than simply dismiss the woman's claim, he proposed that they decide how one should *test* the claim. The tenor of the conversation changed at this suggestion, and the scientists began to discuss how the claim should be tested. Within a few minutes cups of tea with milk had been prepared and presented to the woman for tasting.

At this point, you may be wondering who the innovative scientist was and what the results of the experiment were. The scientist was R. A. Fisher, who first described this situation as a pedagogical example in his 1925 book on statistical methodology [Fis25]. Fisher developed statistical methods that are among the most important and widely used methods to this day, and most of his applications were biological.

You might also be curious about how the experiment came out. How many cups of tea were prepared? How many did the woman correctly identify? What was the conclusion?

Fisher never says. In his book he is interested in the method, not the particular results. But let's suppose we decide to test the lady with ten cups of tea. We'll flip a coin to decide which way to prepare the cups. If we flip a head, we will pour the milk in first; if tails, we put the tea in first. Then we present the ten cups to the lady and have her state which ones she thinks were prepared each way.

[2] There are some limits to the simulation approach (some simulations still require too much computational power to be used in practice), so there are still advantages to direct analytical methods.

It is easy to give her a score (9 out of 10, or 7 out of 10, or whatever it happens to be). It is trickier to figure out what to do with her score. Even if she is just guessing and has no idea, she could get lucky and get quite a few correct – maybe even all 10. But how likely is that?

Let's try an experiment. I'll flip 10 coins. You guess which are heads and which are tails, and we'll see how you do.

⋮

Comparing with your classmates, we will undoubtedly see that some of you did better and others worse.

Now let's suppose the lady gets 9 out of 10 correct. That's not perfect, but it is better than we would expect for someone who was just guessing. On the other hand, it is not impossible to get 9 out of 10 just by guessing. So here is Fisher's great idea: Let's figure out how hard it is to get 9 out of 10 by guessing. If it's not so hard to do, then perhaps that's just what happened, so we won't be too impressed with the lady's tea tasting ability. On the other hand, if it is really unusual to get 9 out of 10 correct by guessing, then we will have some evidence that she must be able to tell something.

But how do we figure out how unusual it is to get 9 out of 10 just by guessing? There are other methods (and you may already know of one), but for now, let's just flip a bunch of coins and keep track. If the lady is just guessing, she might as well be flipping a coin.

So here's the plan. We'll flip 10 coins. We'll call the heads correct guesses and the tails incorrect guesses. Then we'll flip 10 more coins, and 10 more, and 10 more, and .... That would get pretty tedious. Fortunately, computers are good at tedious things, so we'll let the computer do the flipping for us.

The `mosaic` package must be installed before it can be used. It is available from CRAN.

The `rflip()` function can flip one coin

```
> require(mosaic)
> rflip()
Flipping 1 coins [ Prob(Heads) = 0.5 ] ...

T

Result: 0 heads.
```

or a number of coins

```
> rflip(10)
Flipping 10 coins [ Prob(Heads) = 0.5 ] ...

H T H H T H H H T H

Result: 7 heads.
```

Typing `rflip(10)` a bunch of times is almost as tedious as flipping all those coins. But it is not too hard to tell R to `do()` this a bunch of times.

```
> do(3) * rflip(10)
  n heads tails
1 10     8     2
2 10     4     6
3 10     1     9
```

Let's get R to `do()` it for us 10,000 times and make a table of the results.

```
> random.ladies <- do(10000) * rflip(10)
```

```
> table(random.ladies$heads)

   0    1    2    3    4    5    6    7    8    9   10
   5  102  467 1203 2048 2470 2035 1140  415  108    7

> perctable(random.ladies$heads)     # display table using percentages

    0     1     2     3     4     5     6     7     8     9    10
 0.05  1.02  4.67 12.03 20.48 24.70 20.35 11.40  4.15  1.08  0.07
```

We can display this table graphically using a plot called a **histogram**.

```
> histogram(~ heads, random.ladies,
         breaks=-.5 + (0:11)
 )
```

We have control of the histogram display by specifying the breaks from -.5 to 10.5:

```
> -.5 + (0:11)

 [1] -0.5  0.5  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

You might be surprised to see that the number of correct guesses is exactly 5 (half of the 10 tries) only 25% of the time. But most of the results are quite close to 5 correct. 67% of the results are 4, 5, or 6, for example. And 90% of the results are between 3 and 7 (inclusive). But getting 8 correct is a bit unusual, and getting 9 or 10 correct is even more unusual.

So what do we conclude? It is possible that the lady could get 9 or 10 correct just by guessing, but it is not very likely (it only happened in about 1.2% of our simulations). So *one of two things must be true*:

- The lady got unusually "lucky", or
- The lady is not just guessing.

Although Fisher did not say how the experiment came out, others have reported that the lady correctly identified all 10 cups! [Sal01]

A different design

Suppose instead that we prepare five cups each way (and that the woman tasting knows this). We give her five cards labeled "milk first", and she must place them next to the cups that had the milked poured first. How does this design change things?

```
> results <- do(10000) * table(sample( c('M','M','M','M','M','T','T','T','T','T'), 5))
> perctable(results$M)

      1       2       3       4       5
 9.9739 39.8252 39.4536 10.3254  0.4219
```

3.3.2 Golfballs in the Yard

> *This example can be used as a first example of hypothesis testing or as an introduction to chi-squared tests. As an introduction to hypothesis testing it is very useful in helping students understand what a test statistic is and its role in hypothesis testing.*

The Story

Allan Rossman once lived along a golf course. One summer he collected the golf balls that landed in his yard and tallied how many were labeled with 1's, 2's, 3's, and 4's because he was curious to know whether these numbers were equally likely.[3]

Of the first 500 golf balls, 14 had either no number or a number other than 1, 2, 3, or 4. The remaining 486 golf balls form our sample:

1	2	3	4	other
137	138	107	104	14

We can enter this data into R using the `c()` function.

```
> golfballs <- c(137, 138, 107, 104)
```

Coming up with a test statistic

At this point, ask students what they think the data indicates about the hypothesis that the four numbers are equally likely. Students usually notice right away that there are a lot of 2's. But perhaps that's just the result of random sampling. We can generate random samples and see how the random samples compare with the actual data:

```
> table(rdata(486, 1:4))   # 486 draws from the numbers 1 thru 4 (equally likely)

  1   2   3   4
111 132 117 126
```

`rgolfballs` can be generated in advance if you don't want to distract your students with thinking about how do create it. There is a pre-built `rgolfballs` in the `fastR` package.

It is useful to generate some more of these samples to get a better feel for the sampling distribution under the null:

```
> do(25) * table(rdata(486, 1:4))

    1   2   3   4
1  104 144 112 126
2  122 128 123 113
3  128 106 124 128
4  117 122 140 107
5  107 138 117 124
6  124 124 124 114
7  119 128 111 128
8  110 148 110 118
9  124 146 113 103
10 134 120 125 107
11 124  99 132 131
12 114 113 141 118
13 122 116 118 130
14 126 116 127 117
15 115 118 116 137
16 111 125 129 121
17 134  99 117 136
18 133 124 111 118
19 133 119 114 120
20 122 125 119 120
21 119 133 109 125
22 135 118 124 109
23 118 124 102 142
24 137 101 131 117
25 103 110 141 132
```

Tip
Have your students calculate test statistics mentally from a small portion of the sampling distribution. Assign each student a row or two, then ask for a show of hands to see how many exceed the test statistic calculated from the data.

From this we see that it is not incredibly unlikely to see a count of 138 or more. (See samples 1, 4, 5, 8, 9, 12, 23, 25.) Students are often surprised just how often this occurs.

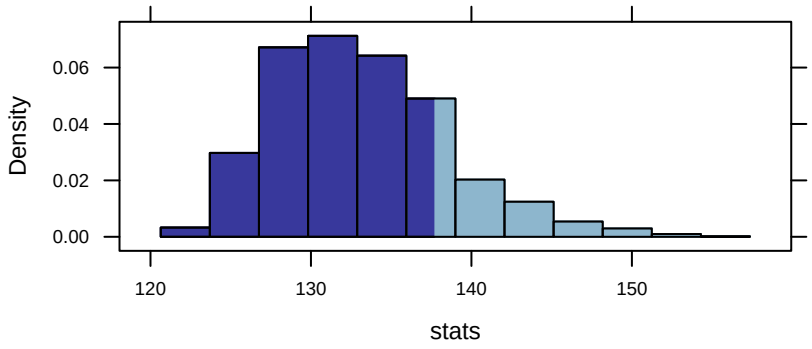
[3] You can have some discussion with your students about what population is of interest here. Given the location of the house, the golf balls were primarily struck by golfers of modest ability and significant slice, all playing on one particular golf course. These results may or may not extend to other larger populations of golfers.

Once students understand the idea of a test statistic and how it is computed from data, it's time to let the computer automate things. First, we generate a better approximation to the sampling distribution assuming each number is equally likely.

```
> rgolfballs <- do(2000) * table(rdata(486, 1:4))
```

The `statTally()` function can tabulate and display the sampling distribution and compared to the test statistic.

```
> print(statTally(golfballs, rgolfballs, max))
Test Stat function: max


Test Stat applied to sample data = 138

Test Stat applied to random data:

50% 90% 95% 99%
132 141 144 149

Of the random samples
         1620 ( 81 % ) had test stats < 138
         66 ( 3.3 % ) had test stats = 138
         314 ( 15.7 % ) had test stats > 138
```



More test statistics

One of the goals of this activity is to have the students understand the role of a test statistic. Students are encouraged to dream up test statistics of their own. The minimum count is often suggested as an alternative, so we try that one next.

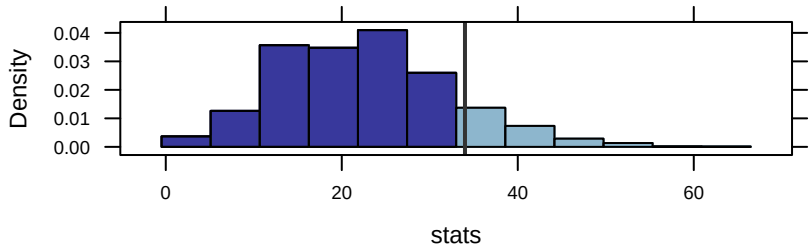
```
> print(statTally(golfballs, rgolfballs, min))   # output surpressed.
```

These two test statistics (maximum count and minimum count) feel like they aren't making full use of our data. Perhaps we would do better if we looked at the difference between the maximum and minimum counts. This requires writing a simple function.

See Section A.7 for a tutorial on writing your own functions.

```
> mystat1 <- function(x) { diff(range(x)) }
> print(statTally(golfballs, rgolfballs, mystat1, v=mystat1(golfballs))) # add a vertical line
```

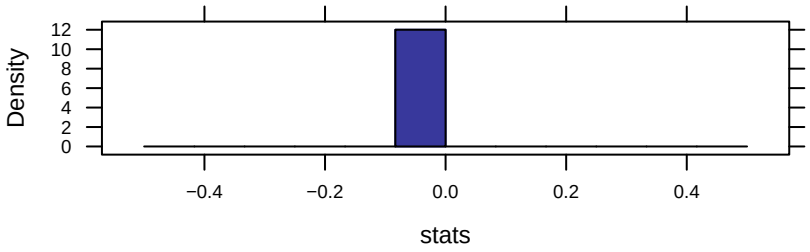
```
Test Stat function: mystat1


Test Stat applied to sample data = 34

Test Stat applied to random data:

50% 90% 95% 99%
 22  36  40  49

Of the random samples
         1715 ( 85.75 % ) had test stats < 34
         27 ( 1.35 % ) had test stats = 34
         258 ( 12.9 % ) had test stats > 34
```



The World's Worst Test Statistic

Usually I get lucky and someone will suggest the world's worst test statistic: The sum of the differences between the counts and $486/4 = 121.5$.

```
> mystat2 <- function(x) { sum( x - 121.5 ) }
> print(statTally(golfballs, rgolfballs, mystat2))


Test Stat function: mystat2


Test Stat applied to sample data = 0

Test Stat applied to random data:

50% 90% 95% 99%
  0   0   0   0

Of the random samples
         0 ( 0 % ) had test stats < 0
         2000 ( 100 % ) had test stats = 0
         0 ( 0 % ) had test stats > 0
```

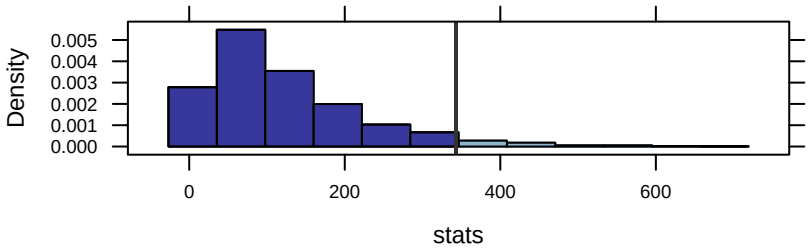


Tip
As students come up with test statistics, let them name them, or name them after them (S for the Smith statistic, etc.) It adds to the fun of the activity and mirrors how the statistics they will learn about got their names.

This test statistic is bad because it doesn't depend on the data, so the distribution of the test statistic is the same whether the null hypothesis is true or false.

But it is close to a good idea. Let's add in an absolute value...

```
> sad <- function(x) { sum(abs(x-121.5)) }
> print(statTally(golfballs, rgolfballs, sad, v=sad(golfballs)))
Test Stat function: sad


Test Stat applied to sample data = 64

Test Stat applied to random data:

  50%   90%   95%   99%
29.00 49.00 54.00 63.01

Of the random samples
         1980 ( 99 % ) had test stats < 64
         5 ( 0.25 % ) had test stats = 64
         15 ( 0.75 % ) had test stats > 64
```

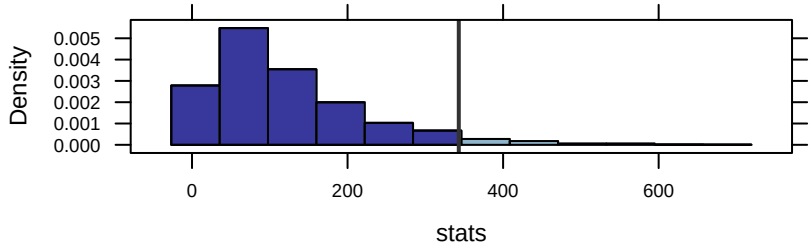


It is a matter of teaching style whether you write this function with the magic number 121.5 hard-coded in or use `mean(x)` instead. The latter is preferable for generalizable method, of course. But the former may have pedagogical advantages, especially in the Intro Stats course.

Squaring those differences (or equivalently using the standard deviation or variance) is also often suggested.

```
> print(statTally(golfballs, rgolfballs, var, v=var(golfballs)))
Test Stat function: var


Test Stat applied to sample data = 343
```

```
Test Stat applied to random data:

  50%   90%   95%   99%
 95.0 253.7 317.9 449.7

Of the random samples
         1923 ( 96.15 % ) had test stats < 343
         3 ( 0.15 % ) had test stats = 343
         74 ( 3.7 % ) had test stats > 343
```



This example illustrates some important ideas about hypothesis testing, namely.

1. The test statistic must summarize the evidence we will use to judge the null hypothesis *in a single number* computed from our sample.
2. We judge the test statistic computed from our sample by comparing it to test statistics computed on random samples *assuming the Null Hypothesis is true.*
3. Some test statistics work better than others.

   A good test statistic should look quite different when the null hypothesis is true from how it looks when the null hypothesis is false. (This is a first hint at the idea of power.)

3.3.3 Not just a trivial example

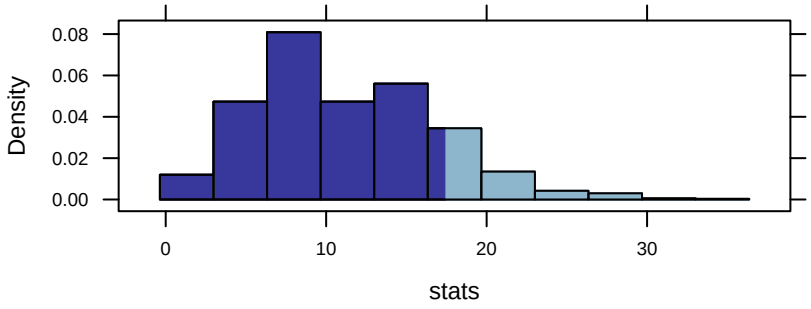
The test we are developing here is called a goodness of fit test. This is the same sort of test we might use, for example, to test whether the phenotype ratios in a plant hybridization experiment are consistent with a certain genetic explanation. Suppose we observed a 20:50:35 ratio in a situation predicted a 1:2:1 ratio (Mendelian codominance). Are our data consistent with the theory?

Here are a few reasonable test statistics for this situation.

```
> theory <- c('A', 'B', 'B', 'C')
> table( rdata(105, theory) )

 A  B  C
23 56 26

> sad <- function(observed) {               # sum of absolute deviation
         n <- sum(observed)
         expected <- c(.25, .5, .25) * n
         sum ( abs( observed - expected) )
 }
```

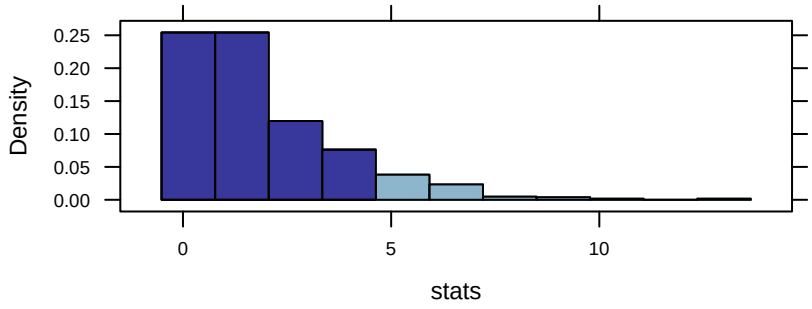
```
> mad <- function(observed) {              # mean of absolute deviation
         n <- sum(observed)
         expected <- c(.25, .5, .25) * n
         mean ( abs( observed - expected) )
 }

> chs <- function(observed) {              # Pearson's chi-squared test statistic
         n <- sum(observed)
         expected <- c(.25, .5, .25) * n
         sum ( ( observed - expected )^2 / expected )
 }

> mls <- function(observed) {              # max likelihood chi-squared test statistic
         n <- sum(observed)
         expected <- c(.25, .5, .25) * n
         2 * sum ( log( observed ) - log( observed / expected ) )
 }
```

```
> random.data <- do(1000) * table( rdata(105, theory) )
> print( statTally( c(20, 50, 35), random.data, sad) )
Test Stat function: sad


Test Stat applied to sample data = 17.5

Test Stat applied to random data:

  50%   90%   95%   99%
10.50 19.05 21.50 26.50

Of the random samples
         849 ( 84.9 % ) had test stats < 17.5
         13 ( 1.3 % ) had test stats = 17.5
         138 ( 13.8 % ) had test stats > 17.5
```




```
> random.data <- do(1000) * table( rdata(105, theory) )
> print( statTally( c(20, 50, 35), random.data, chs) )
Test Stat function: chs


Test Stat applied to sample data = 4.524

Test Stat applied to random data:
```

```
  50%   90%   95%   99%
1.400 4.657 5.651 8.943

Of the random samples
         897 ( 89.7 % ) had test stats < 4.524
         2 ( 0.2 % ) had test stats = 4.524
         101 ( 10.1 % ) had test stats > 4.524
```



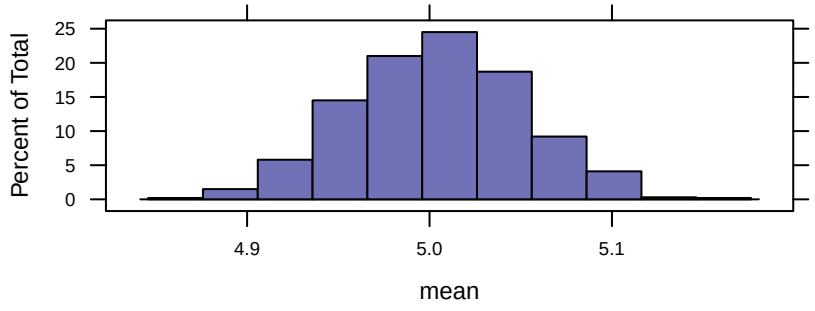
The traditional Chi-squared test is an approximation to the test above (which saves us the time associated with the simulations, and gives a tidy formula). R can, of course, compute that too:

```
> chisq.test( c(20,50,35), p=c(.25,.5,.25) )
        Chi-squared test for given probabilities

data:  c(20, 50, 35)
X-squared = 4.524, df = 2, p-value = 0.1042

> xchisq.test( c(20,50,35), p=c(.25,.5,.25) )     # mosaic version is a bit more verbose
        Chi-squared test for given probabilities

data:  c(20, 50, 35)
X-squared = 4.524, df = 2, p-value = 0.1042

 20.00    50.00    35.00
(26.25)  (52.50)  (26.25)
 [1.49]   [0.12]   [2.92]
<-1.22>  <-0.35>  < 1.71>

key:
        observed
        (expected)
        [contribution to X-squared]
        <residual>
```

3.3.4 Estimating the Mean Sepal Lengths of Irises via Bootstrap

```
> data(iris)                              # loads iris data from datasets package
> summary(iris)                           # summarize each variable
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
 Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa    :50
 1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
 Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
 Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
```

```
 3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
 Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
> summary( Sepal.Length ~ Species, data=iris, fun=mean )     # mean for each species
Sepal.Length    N=150

+-------+----------+---+-----+
|       |          |N  |mean |
+-------+----------+---+-----+
|Species|setosa    | 50|5.006|
|       |versicolor| 50|5.936|
|       |virginica | 50|6.588|
+-------+----------+---+-----+
|Overall|          |150|5.843|
+-------+----------+---+-----+
> setosa <- subset(iris, Species=='setosa')
> mean(setosa$Sepal.Length)
 mean
5.006
> with( resample(setosa),  mean( Sepal.Length) )
 mean
4.938
> with( resample(setosa),  mean( Sepal.Length) )
 mean
5.024
> with( resample(setosa),  mean( Sepal.Length) )
 mean
4.938
> res <- do(1000) * with( resample(setosa),  mean( Sepal.Length) )


> histogram( ~ mean, res )
```




```
> quantile( res$mean, c(0.025, 0.975) )    # middle 95% of resampled means
 2.5% 97.5%
4.916 5.096
```

Let's see how that compares with the interval produced by a t-test:

```
> interval(t.test( iris$Sepal.Length ))
```

```
Method: One Sample t-test

mean of x
    5.843

95% confidence interval:
5.71 5.977
```

3.4 The Multi-World Metaphor for Hypothesis Testing

In this section we move toward a more systematic approach to statistical inference with the goal of providing a flexible tool that works in a wide range of situations.

Statistical inference is hard to teach.[4] Often, instead of teaching the logic of inference, we teach methods and techniques for calculating the quantities used in inference: standard errors, t-statistics, p-values, etc.

Perhaps because students don't understand the logic, they have strong misconceptions about confidence intervals and, especially, about p-values. For example, even among professional scientists, the mainstream (mis)-understanding of p-values is that they reflect the probability that the null hypothesis is correct.

Part of the reason why statistical inference is hard to grasp is that the logic is genuinely hard. It involves contrapositives, it involves conditional probabilities, it involves "hypotheses." And what is a hypothesis? To a scientist, it is a kind of theory, an idea of how things work, an idea to be proven through lab or field research or the collection of data. But statistics hews not to the scientist's but to the philosopher's or logician's rather abstract notion of a hypothesis: a "proposition made as a basis for reasoning, without any assumption of its truth." (Oxford American Dictionaries) Or more simply:

A hypothesis is a statement that may be true or false.

What kind of scientist would frame a hypothesis without some disposition to think it might be true? Only a philosopher or a logician.

To help connect the philosophy and logic of statistical hypotheses with the sensibilities of scientists, it might be helpful to draw from the theory of kinesthetic learning — an active theory, not a philosophical proposition — that learning is enhanced by carrying out a physical activity. Many practitioners of the reform style of teaching statistics engage in the kinesthetic style; they have students draw M&Ms from a bag and count the brown ones, they have students walk randomly to see how far they get after n steps [Kap09], or they toss a globe around the classroom to see how often a students' index finger lands in the ocean [GN02].

To teach hypothesis testing in a kinesthetic way, you need a physical representation of the various hypotheses found in statistical inference. One way to do this involves not the actual physical activity of truly kinesthetic learning, but concrete stories of making different sorts of trips to different sorts of worlds. A hypothesis may be true on some worlds and false on others. On some planets we will know whether a hypothesis is true or false, on others, the truth of a hypothesis is an open question.

Of course, the planet that we care about, the place about which we want to be able to draw conclusions. It's Planet Earth:

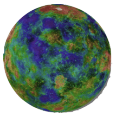
The simpler "definition" is a useful way to describe this to students, but it is not equivalent to the dictionary definition which includes an important notion of the reason for hypotheses. A hypothesis is a statement that is posed for the purposes of determining what would follow if the statement were true. This is the sense of 'hypothesis of a theorem'. It is also the sense of the null hypothesis. We assume the null hypothesis is true and 'see what follows' from that assumption. Unfortunately, it is not the way this word is often used in high school science courses.

Thought of another way, the scientists have conflated the words 'hypothesis' and 'conjecture'.

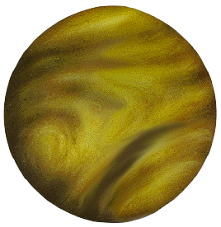
[4] Here we are considering only the frequentist version of inference. The Bayesian approach is different and has different features that make it hard to teach and understand. In these notes, we will be agnostic about frequentist vs Bayesian, except to acknowledge that, for good or bad, the frequentist approach is vastly dominant in introductory statistics courses.

We want to know which hypotheses are true on Earth and which are false. But Earth is a big place and complicated, and it's hard to know exactly what's going on. So, to deal with the limits of our abilities to collect data and to understand complexity, statistical inference involves a different set of planets. These planets resemble Planet Earth in some ways and not in others. But they are simple enough that we know exactly what's happening on them, so we know which hypotheses are true and which are false on these planets.

These planets are:



Planet Sample



Planet Null

Planet Alt

Planet Sample is populated entirely with the cases we have collected on Planet Earth. As such, it somewhat resembles Earth, but many of the details are missing, perhaps even whole countries or continents or seas. And of course, it is much smaller than Planet Earth.

Planet Null is a boring planet. Nothing is happening there. Express it how you will: All groups all have the same mean values. The model coefficients are zero. Different variables are unrelated to one another. Dullsville. But even if it's dull, it's not a polished billiard ball that looks the same from every perspective. It's the clay from which Adam was created, the ashes and dust to which man returns. But it varies from place to place, and that variation is not informative. It's just random.

Finally, there is Planet Alt. This is a cartoon planet, a caricature, a simplified representation of our idea of what is (or might be) going on, our theory of how the world might be working. It's not going to be exactly the same as Planet Earth. For one thing, our theory might be wrong. But also, no theory is going to capture all the detail and complexity of the real world.

In teaching statistical inference in a pseudo-kinesthetic way, we use the computer to let students construct each of the planets. Then, working on that planet, the student can carry out simulations of familiar statistical operations. Those simulations tell the student what they are likely to see *if they were on that planet*. Our goal is to figure out whether Earth looks more like Planet Null or more like Planet Alt.

For the professional statisticians, the planet metaphor is unnecessary. The professional has learned to keep straight the various roles of the null and alternative hypothesis, and why one uses a sample standard error to estimate the standard deviation of the sampling distribution from the population. But most students find the array of concepts confusing and make basic categorical errors. The concrete nature of the planets simplifies the task of keeping the concepts in order. For instance, Type I errors only occur on Planet Null. You have to be on Planet Alt to make a Type II error. And, of course,

no matter how many (re)samples we make on Planet Sample, it's never going to look more like Earth than the sample itself.

3.4.1 The Sampling Distribution

Section 3.5 shows some examples of drawing random samples repeatedly from a set of data. To discuss the sampling distribution, it's helpful to make clear that the sampling distribution refers to results from the **population**. In the planet metaphor, this means that the samples are drawn from Planet Earth:

It can be a challenge to find a compelling example where you have the population, rather than a sample, in hand. Sports statistics provide one plausible setting, where you have, for example, the names and attributes of every professional player. The ability to sample from Earth mapping software provides another compelling setting. (See Section 5.2.1.)

For our example, we will take the complete list of results from a running race held in Washington, D.C. in April 2005 — the Cherry Blossom Ten-Mile Run. The data set gives the `sex`, `age`, and `net` running time (start line to finish line) for each of the 8636 runners who participated: the population of runners in that race.

```
> population <- TenMileRace
> nrow(population)
[1] 8636
```

If you have the population on the computer, there's little point in taking a random sample. But the point here is to illustrate the consequences of taking a sample. So, imagine that in fact it was difficult or expensive or dangerous or destructive to collect the data on a runner, so you want to examine just a small subset of the population.

Let's collect a sample of 100 runners:

```
> planet.sample <- sample(population, 100)
```

With that sample, we can calculate whatever statistics are of interest to us, for instance:

```
> with(planet.sample, mean(age))
 mean
36.72
> with(planet.sample, sd(age))
   sd
10.58
> with(planet.sample, mean(sex=="F"))
mean
0.46
> lm(net ~ age + sex, data=planet.sample)
...
(Intercept)          age         sexM
     5372.0         18.9       -865.2
```

These various numbers are informative in their own way about the sample, but it's important to know how they might relate to the population. For example, how precise are these numbers? That is, if someone had collected a different random sample, how different would their results likely be?

Given that we have the population in hand, that we're on Planet Earth, we can go back and collect such samples many more times and study how much they vary one to the other. We'll do this by replacing `planet.sample` with a command to sample anew from the population.

```
> with(sample(population,100), mean(age))

 mean
36.72

> with(sample(population,100), sd(age))

   sd
9.455

> with(sample(population,100), mean(sex=="F"))

mean
 0.5

> lm(net ~ age + sex, data=sample(population,100))

...
(Intercept)          age         sexM
    5995.05        -2.15      -767.17
```

Slightly different results! To quantify this, repeat the sampling process many times and look at the distribution: the **sampling distribution**.

```
> sample.means <- do(500) * with(sample(population,100), mean(age))
> sample.sds   <- do(500) * with(sample(population,100), sd(age))
> sample.props <- do(500) * with(sample(population,100), mean(sex=="F"))
> sample.regressions  <- do(500) * lm(net ~ age + sex, data=sample(population,100))
```

You can display the sampling distribution in several ways: histograms, box-and-whisker plots, density plots. This is worth doing with the class. Make sure to point out that the distribution depends on the statistic being calculated: in the above examples, the mean age, the standard deviation of ages, the fraction of runners who are female, the relationship between running time and `sex` and `age`. (We'll get to the dependence on sample size in a little bit.)

Insofar as we want to be able to characterize the repeatability of the results of the sampling process, it's worth describing it in terms of the spread of the distribution: for instance, the standard deviation. Of course, when talking about a sampling distribution, we give another name to the standard deviation: the *standard error*. Actually calculating the standard error may perhaps solidify in the students mind that it is a standard deviation.

```
> sd(sample.means)  # standard error for mean age

 mean
1.113

> sd(sample.sds)    # standard error for sd of age

    sd
0.7214

> sd(sample.props)  # standard error for fraction female

   mean
0.05243

> sd(sample.regressions)  # standard errors for regression statistics

Intercept       age      sexM     sigma r-squared
321.06953   9.09629 191.47578  76.14492   0.06383
```


Example 3.1. This is an activity to carry out in class, with each student or pair of students at a computer.

1. Make sure you can replicate the calculations for the standard error of one of the statistics in the `TenMileRace` example with a sample of size $n = 100$. Your reproduction won't be exact, but it should be reasonably close.

2. Now repeat the calculations, but use sample sizes that are larger. From $n = 100$, increase to $n = 400$, then $n = 1600$, then $n = 6400$. How does the standard error depend on n? Does larger n lead to a bigger or smaller standard error? Which of these formulas most closely matches the pattern you see:

   (a) The standard error increases with n.

   (b) The standard error increases with $\sqrt{n}$.

   (c) The standard error gets smaller with increasing n with the pattern $1/\sqrt{n}$.

   (d) The standard error gets smaller with increasing n with the pattern $1/n$.

3. Use the pattern you observed to predict what will be the standard error for a sample of size $n = 1000$. Then carry out the actual simulation of repeated sampling using that sample size and compare your prediction to the result you actually got.

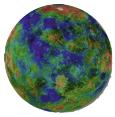
4. In the above, you used `do(500)` replications of random sampling. Suppose you use `do(100)` or `do(2000)` replications instead? Do your results depend systematically on the number of replications?

◇

3.4.2 The Re-Sampling Distribution

The sampling distribution is a lovely theoretical thing. But if it were easy to replicate taking a sample over and over again, wouldn't you just take a larger sample in the first place? The practical problem you face is that the sample you took was collected with difficulty and expense. There's no way you are going to go back and repeat the process that was so difficult in the first place. Remember, real samples from the real population (on planet Earth) can't be obtained by simply asking a computer to sample for us.

This is where Planet Sample comes in.



In our example we have access to the entire population, but this is not typically the case.

Although Planet Earth is very large and we have only incomplete information about it, all the data for Planet Sample is in our computer. So

Although sampling from Planet Earth is difficult, sampling from Planet Sample is easy.

To illustrate, here's a sample from the running population:

```
> planet.sample = sample(population, 100) # one sample from the population
```

In reality, of course, you wouldn't run a command to create the sample. You would go out and do the hard work of randomly selecting cases from your population, measuring their relevant attributes, and recording that data. So let's pretend that's what we did. Pretend that `planet.sample` is the result of laborious data collection.

Now you can go through the exact calculations you did to construct the sampling distribution (on Planet Earth), but instead sample from Planet Sample:

```
> with(sample(planet.sample,100), mean(age))

 mean
35.58

> with(sample(planet.sample,100), mean(age))

 mean
35.58

> with(sample(planet.sample,100), mean(age))

 mean
35.58
```

Wait, something went wrong. We are getting the same mean every time. The reason is that Planet Sample is small. It only has 100 inhabitants and we are sampling all 100 each time. We could take smaller samples, but we want to learn about samples of the same size as our actual sample, so that's not a good option.

CAUTION! Remember to use `resample()` rather than `sample()` to compute a resampling distribution.

Our solution to this problem is to sample *with replacement*. That is, we will select an inhabitant of Planet Sample, record the appropriate data, and then put them back on the planet – possibly selecting that same inhabitant again later in our sample. We'll call sampling with replacement from Planet Sample **resampling** for short. The `resample()` function in the `mosaic` package can do this as easily as sampling without replacement.

```
> with(resample(planet.sample,100), mean(age))

 mean
36.44

> with(resample(planet.sample,100), mean(age))

 mean
35.24

> with(resample(planet.sample,100), mean(age))

 mean
35.78
```

Ah, that looks better. Now let's resample a bunch of times.[5]

```
> resample.means <- do(500) * with(resample(planet.sample,100), mean(age))
> resample.sds   <- do(500) * with(resample(planet.sample,100), sd(age))
> resample.props <- do(500) * with(resample(planet.sample,100), mean(sex=="F"))
> resample.regressions <- do(500) * lm(net ~ age + sex, data=sample(planet.sample,100))
```

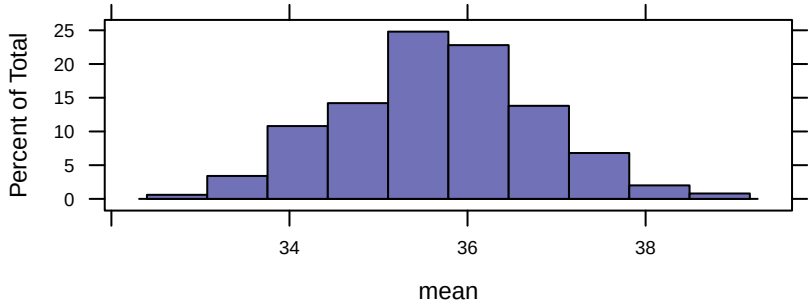
And, then, summarize the resulting distributions:

```
> sd(resample.means)        # standard deviation of mean ages

 mean
1.091

> sd(resample.sds)          # standard deviation of sd of age

   sd
1.079
```

[5] By default, `resample()` will draw a sample as large as the population of Planet Sample, so we can drop the sample size if we like.

```
> sd(resample.props)        # standard deviation of fraction female
   mean
0.05169
> sd(resample.regressions)  # standard deviation of resampled regression statistics
Intercept       age      sexM     sigma r-squared
1.574e-12 1.648e-14 4.724e-13 1.571e-13 6.073e-17
```

```
> histogram( ~ mean, resample.means )
```



Sampling like this on Planet Sample, isn't quite the same as constructing the sampling distribution. This is for the simple reason that it is being done on Planet Sample rather than Planet Earth. To emphasize the distinction, it's helpful to refer to the resulting distribution as the *resampling distribution* in contrast to the sampling distribution.

Example 3.2. Have your students compare the results they get from resampling of a fixed sample of size n, to repeated draws from the population with the same sample size n.

Emphasize that the resampling process is good for estimating the width of the sampling distribution, but not so good for estimating the center. That is, the results on Planet Sample will generally compare quite well to Planet Earth for the standard error, but the means of the sampling distributions and the resampling distributions can be quite different.

In a more advanced class, you might ask how big a sample is needed to get a reasonable estimate of the standard error. ◇

The question naturally arises, is the standard error estimated from the resampling distribution good enough to use in place of the standard deviation of the actual sampling distribution. Answering this question requires some solid sense of *what you are using the standard error for*. In general, we use standard errors to get an idea of whether the point estimate is precise enough for the purpose at hand. Such questions can be productively addressed on Planet Alt, where we will journey after a short detour to that most boring of all places, Planet Null.

3.4.3 The Sampling Distribution Under the Null Hypothesis

The Null Hypothesis is often introduced in terms of the values of population parameters, e.g., "The population mean is 98.6," or "The difference between the two group means is zero," or "The population proportion is 50%."

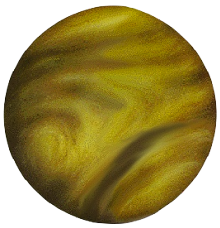
Perhaps this is fine if all you want to talk about is means or proportions or differences between means, as is so often the focus of an introductory statistics course. But instructors would like to think that

their students are going to go farther, and that the introductory course is meant to set them up for doing so.

In the multi-planet metaphor, the Null Hypothesis is about a place where variables are unrelated to one another. Any measured relationship, as indicated by a difference in sample means, a sample correlation coefficient different from zero, non-zero model coefficients, etc., is, on Planet Null, just the result of random sampling fluctuations.

Planet Null is, however, easy to describe in the case of hypotheses about a single proportion. In fact, that is what our Lady Tasting Tea example did (Section 3.3.1).

This formulation is very general and is not hard for students to understand. Ironically, it doesn't work so well for the very simplest null hypotheses that are about single-group means or proportions, so it turns out to be easier to introduce the null hypothesis with the supposedly more complicated cases, e.g., differences in means or proportions, regression coefficients, etc.



Like Planet Sample, Planet Null is a place that you construct. You construct it in a way that makes the Null Hypothesis true, destroying relationships between variables. You've already seen the process in Section **??**: randomization with `shuffle`. The basic idea is to treat relationships as a mapping from the values of explanatory variables in each case to a response variable. By randomizing the explanatory variables relative to the response, you generate the world in which the Null Hypothesis holds true: Planet Null.

Examples

Let's look as some examples.

Example 3.3. The `Whickham` data set contains data from a 1970's sample of 1314 residents of Whickham, a mixed urban and rural district near Newcastle upon Tyne, in the UK. Did more than half of the adult residents of Whickham smoke at that time?

We could use the same approach that worked for the Lady Tasting Tea, but we'll do things a little differently this time. First, let's take a quick look at some of the data.

```
> n <- nrow(Whickham); n
```

```
[1] 1314
```

```
> head(Whickham,3)
```

```
  outcome smoker age
1   Alive    Yes  23
2   Alive    Yes  18
3    Dead    Yes  71
```

We'll answer our question by comparing the smoking rate on Planet Sample

```
> with(Whickham, proptable(smoker))                    # less than 50% of sample smoke
```

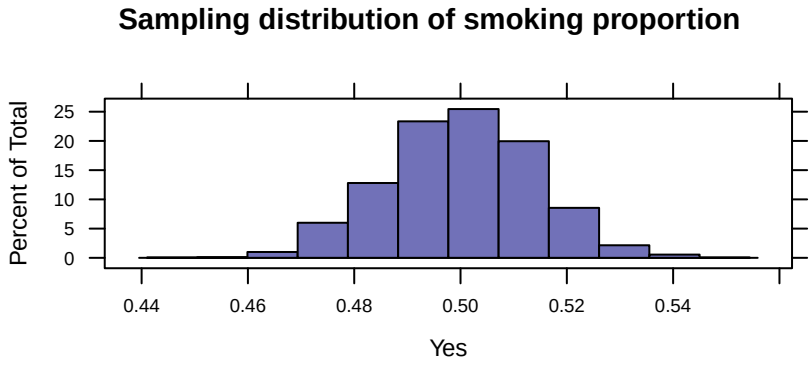
```
smoker
    No    Yes
0.5571 0.4429
```

to the sampling distribution on Planet Null.

```
> planet.null <- data.frame( smoker=c('Yes','No') )  # equal mix of Yes and No on Planet Null
> null.dist <- do(2000) * with( resample(planet.null, n), proptable(smoker) )
> head(null.dist,3)
      No    Yes
1 0.4985 0.5015
2 0.4909 0.5091
3 0.4939 0.5061
> table(with(null.dist, Yes < .4429))
FALSE
 2000
```

2000 of our 2000 samples from Planet Null had a proportion of smokers smaller than the proportion on Planet Sample. It does not appear that our sample came from Planet Null.

```
> histogram( ~ Yes, null.dist, main='Sampling distribution of smoking proportion')
```



◇

Example 3.4. Do the foot widths differ between boys and girls, judging from the `KidsFeet` data?

```
> data(KidsFeet)
> lm( width ~ sex, data=KidsFeet ) # our sample
...
(Intercept)         sexG
      9.190       -0.406
```

Conceptually, this example can be done without the use of `lm()`, but the syntax is trickier. `lm()` takes care of computing the means of the two groups and the difference in those means. See Section **??**.

Looks like girls' feet are a little bit narrower. But is this the sort of thing we might equally well see on Planet Null, where there is no systematic difference between boys' and girls' feet?

```
> do(1) * lm( width ~ shuffle(sex), data=KidsFeet ) # planet null
  Intercept   sexG  sigma r-squared
1      8.92 0.1484 0.5108   0.02175
```

For this particular sample from Planet Null, the girls' feet are a little wider than the boys'. By generating a sampling distribution on Planet Null, we can see the size of the relationship to be expected just due to sampling fluctuations in a world where there is no relationship.

```
> # (approximate) distribution on planet null
> planet.null <- do(500) * lm( width ~ shuffle(sex), data=KidsFeet )
> head(planet.null,2)
  Intercept   sexG  sigma r-squared
1     8.815 0.3639 0.4815   0.13079
2     8.940 0.1074 0.5135   0.01138
```

```
> with( planet.null, mean(abs(sexG) > abs(-0.4058)) )  # a p-value

 mean
0.012
```

The value of −0.4058 observed in our sample is not very likely on Planet Null. This suggests that our sample was not collected from Planet Null: we can reject the Null Hypothesis.

◇

Example 3.5. Is the survival rate for smokers different from that for non-smokers?

```
> do(1) * lm( outcome=="Alive" ~ smoker, data=Whickham ) # from our sample

  Intercept smokerYes  sigma r-squared
1    0.6858   0.07538 0.4482  0.006941

> do(1) * lm( outcome=="Alive" ~ shuffle(smoker), data=Whickham ) # planet null

  Intercept smokerYes  sigma r-squared
1    0.7145   0.01061 0.4497 0.0001374

> # distribution on planet null
> null.distribution  = do(500) * lm( outcome=="Alive" ~ shuffle(smoker), data=Whickham )
> with(null.distribution, mean( abs(smokerYes)  > abs(0.07538) ) ) # a p-value

mean
   0
```

If you're shocked to see that smoking is associated with greater likelihood of being alive (7.5 percentage points greater!) and that the data indicate that this is statistically significant, you should be. But the problem isn't with the calculation, which is the same one you will get from the textbook formulas. The problem is with the failure to take into account covariates. What's shocking is that we teach students about p-values without teaching them about covariates and how to adjust for them.

The big covariate here is `age`. It happens that in the Whickham data, younger people are more likely to smoke. To see this you :

```
> do(1) * lm( smoker=="Yes" ~ age, data=Whickham )          # our sample

  Intercept       age  sigma r-squared
1    0.5961 -0.003264 0.4938   0.01311

> do(1) * lm( smoker=="Yes" ~ shuffle(age), data=Whickham ) # under the null

  Intercept        age sigma r-squared
1    0.4694 -0.0005652 0.497 0.0003931

> # distribution on planet null
> null.distribution <- do(500) * lm(smoker=="Yes" ~ shuffle(age), data=Whickham )
> with(null.distribution, mean(abs(age) > abs(-0.00326)) ) # approx. p-value

mean
   0
```

So, greater `age` is associated with lower `smoker` status. And, of course, older people are more likely to die. Taking both factors together, it turns out that smokers are less likely to die, but that's because they are young. ◇

Example 3.6. Let's make up for the deficiency in the above smoking example. One way to do this is to adjust for `age` when considering the effect of `smoker` status. We'll consider this more in Chapter **??**, but for now, we'll just build the model.

```
> # our sample on planet Earth
> do(1) * glm( outcome=="Alive" ~ smoker + age, data=Whickham, family="binomial" )

  Intercept smokerYes     age
1     7.599   -0.2047 -0.1237
```

```
> # o sample on planet null
> do(1) * glm( outcome=="Alive" ~ shuffle(smoker) + age, data=Whickham, family="binomial" )
  Intercept smokerYes     age
1     7.393   0.02131 -0.1219
> # distribution on planet null
> null.distribution <- do(500) * glm( outcome=="Alive" ~ shuffle(smoker) + age,
                                      data=Whickham, family="binomial" )
> with(null.distribution, mean(abs(smokerYes) > abs(-0.205)) ) # approx. p-value
 mean
0.222
```

You can see that the coefficient on `smokerYes` is negative, and that the p-value indicates significance. So, smoking in these data are associated with a lower probability of survival, but only when adjusting for age.

You might have noticed that the model built here was a logistic model. There's good reason to do that, since we are modeling probabilities the value of which must always be between 0 and 1. But notice also that the logic of hypothesis testing remains the same: construct a Planet Null by randomizing an explanatory variable with respect to a response variable. ◇

As this is being written, the US Space Shuttle is carrying out it's last couple of missions before being retired. For a few years, at least, your students will know what you mean by "Space Shuttle." You can help them remember the way to create Planet Null if, at the cost of some self-dignity, you tell them, "Take the Space Shuffle to Planet Null."

3.4.4 The Sampling Distribution Under the Alternative Hypothesis

Planet Alt is the place where we implement our theories of how the world works. This will seem like an odd statement to those who are familiar with the standard introductory textbook formulation of the alternative hypothesis in its "anything but the Null" form, e.g., $H_a : \mu_1 \neq \mu_2$. Move away from that formulation, whose only point seems to be to inform whether to do a one-tailed or a two-tailed test.

Instead head off to Planet Alt.

How do you get there? You build a simulation of the world as you think it might be.

Example 3.7. To illustrate, imagine that you are interested in researching the potential relationship between vitamin D deficiency and high blood pressure. Your eventual plan is to do a study, perhaps one where you draw blood samples to measure vitamin D levels, and correlate this with high blood pressure. From your previous experience, you know that high blood pressure is particularly a problem in men aged 50 and older, and that black men seem to be more susceptible than whites.

You scour the literature, looking for data on vitaminD and blood pressure. What you find are some papers describing vitamin D levels in blacks and whites, and that, on average, blacks seem to have substantially lower vitamin D levels than whites. There's also data on high blood pressure, but no good study relating vitamin D to blood pressure. That's a problem, but it's also the source of your research opportunity.

The instructor would provide this function to the students.

You form your alternative hypothesis based on your idea of a world in which the relationship between vitamin D and blood pressure is large enough to be interesting at a clinical level, but small enough to have been missed by past research. You decide a substantial but typical deficiency in vitamin D will, on average lead to a 5mmHg increase in systolic blood pressure.

Now, to construct Planet Alt:

```
> planet.alt = function(n=10,effect.size=1) {
   race = resample( c("B","B","W","W","W"), n)
   D = pmax(0, rnorm(n,mean=(37.7+(71.8-37.7)*(race=="W")),
                       sd=(10+(20-10)*(race=="W"))))
   systolic = pmax(80, rnorm(n,mean=130,sd=8) +
                       rexp(n,rate=.15) +
                       effect.size*(30-D)/2 )
   return( data.frame(race=race,
                      D=D,
                      systolic=round(systolic)) )
 }
```

The internals of such a function is not for the faint of heart. You'll see that it creates people that are randomly of race B or W, and in a proportion that you might choose if you were sampling with the intent of examining seriously the role that race plays. The vitamin D level is set, according to your findings in the literature, to have a mean of 37.7 for blacks and 71.8 for whites. The standard deviations also differ between blacks and whites. Finally, the systolic blood is set to produce a population that is a mixture of a normal distribution around 130 mmHg with an exponential tail toward high blood pressure. A drop of vitaminD levels of 10 units leads to an increase in blood pressure of 5 mmHg. There's a parameter, `effect.size` that will allow you to change this without re-programming.

You might not agree with this specific alternative hypothesis. That's fine. You can make your own Planet Alt. There are plenty to go around!

In contrast to the complexity of writing the simulation, using it is simple. Here's a quick study of size $n = 5$:

```
> planet.alt(5)
```

	race	D	systolic
1	W	54.28	118
2	B	45.09	126
3	B	23.74	136
4	B	48.27	139
5	B	34.91	137

◇

Example 3.8. In this example we construct Planet Alt for a comparison of two means.

```
> alt.2.groups <- function(n=10, effect.size=1, sd=1, baseline=0){
         n <- rep(n, length.out=2)
         sd <- rep(sd, length.out=2)
         data.frame(
                 y = c( rnorm(n[1], baseline, sd[1]), rnorm(n[2], baseline+effect.size, sd[2])),
                 group = rep(c('A','B'), n)
         )
 }

> alt.2.groups(3)
```

	y	group
1	-0.5794	A
2	-0.5298	A
3	0.6280	A

```
4 -0.3159     B
5 -0.5045     B
6  0.1617     B
```

◇

We'll discuss how to use such simulations to compute power in Section **??**.

3.5 Taking Randomness Seriously

It's tempting to think that "random" means that anything goes. This is, after all, the everyday meaning of the word. In statistics, though, the point of randomness is to get a representative sample, or to assign experimental treatments in some fairly even way. Using randomness properly means taking care and being formal about where the randomness comes from. When you finally submit the research paper you have written after your long years of labor, you don't want it to be rejected because the reviewer isn't convinced that what you called "random" was really so.

First, to demonstrate that the sort of computer-generated random sampling produces reasonable results. Perhaps it's a little too early in the course to talk about sampling distributions, but you certainly can have your class generate random samples of a reasonable size and show that sample statistics are reasonably close to the population parameter. For instance, have each student in your class do this:

```
> with(SAT, mean(ratio))

 mean
16.86

> with(sample(SAT,25), mean(ratio))

 mean
16.44
```

To Do
We'll have more to say about `do()` shortly.

Or, each student can do this several times:

```
> do(5) * with(sample(SAT,25), mean(ratio))

   mean
1 17.04
2 16.60
3 17.15
4 17.01
5 17.17
```

This raises the question, naturally enough, of what "reasonably close" means, and what it means to measure "how close." Eventually, that might lead you into consideration of differences, mean differences, mean square differences, and root mean square differences: the standard deviation and variance. But even before encountering this formality the students should be able to see that there is nothing systematically wrong with the answers they get from their random samples. Some are too high compared to the population parameter, some are too low, but as a group they are pretty well centered.

For some applications it is useful to have an alternative vocabulary. If you are fond of examples involving cards, you can use `deal()` and `shuffle()` instead of `sample()`.

```
> # These are equivalent
> sample(cards, 5)

[1] "6D"  "10S" "9H"  "JD"  "KH"

> deal(cards, 5)
```

```
[1] "3C" "JS" "JH" "AC" "QS"
```

```
> # These are equivalent
> sample(cards)
```

```
 [1] "2C"  "3H"  "QD"  "KC"  "8C"  "6S"  "9D"  "3D"  "7C"  "8H"  "10C" "7H"  "AS"
[14] "9S"  "8D"  "10S" "2S"  "4C"  "QC"  "7D"  "5S"  "4H"  "6H"  "7S"  "9C"  "KH"
[27] "6C"  "JD"  "3C"  "KS"  "AD"  "5D"  "2H"  "4S"  "AH"  "QH"  "2D"  "AC"  "4D"
[40] "JC"  "9H"  "JS"  "KD"  "JH"  "5H"  "6D"  "10H" "QS"  "10D" "8S"  "5C"  "3S"
```

```
> shuffle(cards)
```

```
 [1] "9S"  "6S"  "JS"  "7C"  "4H"  "KH"  "QH"  "6C"  "9C"  "JC"  "8C"  "5H"  "2S"
[14] "5C"  "KD"  "10H" "3S"  "8D"  "9H"  "8S"  "9D"  "2C"  "AC"  "2D"  "4C"  "7D"
[27] "5S"  "3C"  "10D" "8H"  "2H"  "6D"  "JD"  "4S"  "KC"  "AH"  "QD"  "QC"  "3D"
[40] "AS"  "10C" "KS"  "10S" "QS"  "7S"  "6H"  "AD"  "3H"  "4D"  "7H"  "JH"  "5D"
```

And for sampling with replacement, we offer `resample()`:

```
> # These are equivalent
> sample(1:10, 5, replace=TRUE)
```

```
[1] 8 8 8 7 4
```

```
> resample(1:10, 5)
```

```
[1] 2 4 6 4 4
```

3.5.1 Illuminating Sampling Mistakes

It's helpful at this point to have some examples where an informal or careless approach to randomness leads to misleading results.

Here are some that can be visually compelling:

1. Select books off a library shelf and count how many pages are in each book. When students pick books haphazardly, they get a systematic over-estimate of the page length.

   A random sample of 25 is sufficient to see this for the large majority of students in the class. Going up to 50, though tedious, makes the result even more compelling.

2. A simulation of a real-world study of Alzheimer's disease. The sampling process was to go to nursing homes on a given day and randomly pick out patients who were diagnosed with Alzheimer's. Their files were flagged and, after a few years, the data were examined to find the mean survival time, from when they entered the nursing home to when they died.

3. Sample people and find out how many siblings are in their family (including themselves). Use this data to estimate family size. Since larger families have more children, we will over-sample larger families and over-estimate family size. Section 3.5.4 demonstrates how to fake a data set for this activity. But for the present, we can use the `Galton` dataset and pretend to take a sample of, say, size 300 kids from the whole group of children that Galton had assembled.

   ```
   > data(Galton)
   > with(sample(Galton,300), mean(nkids))
   ```

   ```
    mean
   6.163
   ```

   ```
   > with(sample(Galton,300), mean(nkids))
   ```

   ```
    mean
   6.153
   ```

   ```
   > do(5) * with(sample(Galton,300), mean(nkids))
   ```

```
   mean
1 6.093
2 6.100
3 6.180
4 6.093
5 6.127
```

The samples all give about the same answer: about 6 kids in a family. But this is misleading! The case in the `Galton` data is a child, and families with more children have their family size represented proportionally more.

There are several ways to convert the `Galton` to a new data set where the case is a family. The key information is contained in the `family` variable, which has a unique value for each family. The `duplicated` function identifies levels that are repeats of earlier ones and so the following will do the job (although is not necessarily something that you want to push onto students):

```
> families = subset(Galton, !duplicated(family))
```

Of course, you will want to show that this has done the right job. Compare

```
> nrow(families)
[1] 197
```

to

```
> length(with(Galton, unique(family)))
[1] 197
```

Now check out the average family size when the average is across families, not across kids in families:

```
> with( families, mean(nkids) )
 mean
4.563
```

(Note: In a higher-level course, you could ask students to determine a method to correct for this bias.)

4. Using `googleMap()` (see Section 5.2.1) we can compare uniform sampling of longitude and latitude with correct sampling that takes into account that the earth is a sphere, not a cylinder.

In each of these cases, follow up the haphazard or systematically biased sample with a formally generated sample (which is faster to do, in any event) and show that the formally generated sample gives results that are more representative of the population than the haphazardly sampled data.

3.5.2 Sampling from Distributions

Later in the course, if you cover modeling with distributions, you can sample from these distributions instead of from data sets (treated like populations). R includes set of functions beginning with the letter `r` followed by the (abbreviated) name of a family of distributions (see Table **??** in section **??** for a list of probability distributions available within base R). These functions all generate a random sample from the distribution given a sample size and any necessary parameters for the distribution.

```
> rnorm(20, mean=500, sd=100)              # fake SAT scores
 [1] 398.7 411.2 610.9 560.6 574.2 493.1 521.4 394.1 641.6 417.4 622.5 480.6 462.1
[14] 416.1 524.6 461.1 479.3 609.1 320.0 534.4
> rexp(20, rate=1/5)                       # fake lifetime data
 [1]  2.5392 23.7481  6.6444  4.9789  7.4664  0.6798  3.8735  1.4669  3.7953  2.9426
[11]  5.8247  0.0562  0.3942  3.3576  6.6171  8.6953  1.9079  1.3481  3.8390  5.5996
```

```
> rbinom(20, 10, .5)                    # how many heads in 10 coin tosses?
 [1] 5 8 5 3 4 8 6 6 3 5 6 4 3 7 6 7 6 6 7 4
> rgeom(20, .1)                         # how long until Freddy misses his next free throw?
 [1]  0  5  9 19 39  0 10  9  1  6  5 20  6 11  0 16  0 17 40 11
```

The mosaic package offers rdata() as an alternative to resample() with syntax that mirrors these functions for sampling from a distribution.

```
> rdata(20, 1:4)              # samples with replacement
 [1] 4 2 4 2 4 3 2 2 4 1 4 4 1 1 1 4 1 4 4 1
> set.seed(123)
> # these are equivalent;  note the order of the arguments.
> resample(HELP, 2)
    age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female  sex g1b
131  49            1    yes   22  5          1      126        0   4      0 male yes
358  36            1    yes   36  3          3      362        0  NA      0 male  no
    homeless i1  i2  id indtot linkstatus link   mcs   pcs pss_fr racegrp satreat
131 homeless 64 179 144     42          1  yes 45.49 38.14      5   black      no
358   housed 25  42 430     37          0   no 45.86 14.07      8   white      no
    sexrisk substance treat
131       6   alcohol   yes
358       4   alcohol    no
> rdata(2, HELP)
    age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female    sex
186  43           NA   <NA>   36  1         NA       18        0  NA      1 female
401  29           NA   <NA>   28  2         NA      118        2   1      0   male
    g1b homeless i1 i2  id indtot linkstatus link   mcs   pcs pss_fr racegrp satreat
186 yes   housed 58 58 219     40          1  yes 36.10 37.04     11   black     yes
401  no homeless 43 54 161     43          1  yes 28.48 45.82      7   white      no
    sexrisk substance treat orig.ids
186       2   alcohol   yes      186
401       6   alcohol   yes      401
> set.seed(123)
> # these are equivalent; sampling without replacement now.
> sample(HELP, 2)
    age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female  sex g1b
131  49            1    yes   22  5          1      126        0   4      0 male yes
357  26            1    yes   23  4        106      410        0  NA      0 male  no
    homeless i1  i2  id indtot linkstatus link   mcs   pcs pss_fr racegrp satreat
131 homeless 64 179 144     42          1  yes 45.49 38.14      5   black      no
357   housed  6   6 428     15          0   no 38.28 36.49      5   black      no
    sexrisk substance treat orig.row
131       6   alcohol   yes      131
357       3    heroin    no      357
> rdata(2, HELP, replace=FALSE)
    age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female    sex
186  43           NA   <NA>   36  1         NA       18        0  NA      1 female
400  31           NA   <NA>   45  5         NA      365        5  NA      0   male
    g1b homeless i1 i2  id indtot linkstatus link  mcs   pcs pss_fr  racegrp satreat
186 yes   housed 58 58 219     40          1  yes 36.1 37.04     11    black     yes
400 yes   housed 26 26 159     33          0   no 15.6 47.66      4 hispanic     yes
    sexrisk substance treat orig.ids
186       2   alcohol   yes      186
400       2    heroin   yes      400
```

3.5.3 do()

The heart of a simulation is doing something over and over. The do() function simplifies the syntax for this and improves the output format (usually returning a data frame). Here's a silly example.

```
> do(3) * "hello"
```

```
  result
1  hello
2  hello
3  hello
```

That's a silly example, because we did the same thing each time. If we add randomness, then each replication is (potentially) different:

```
> do(3) * rflip(10)                # 3 times we flip 10 coins
```

```
   n heads tails
1 10     6     4
2 10     8     2
3 10     1     9
```

What makes do() clever is that is knows about several types of things we might like to repeat and it tries to keep track of the most important summary statistics for us. If we fit a linear model, for example, do() keeps track of the regression coefficients, $\hat{\sigma}$, and r^2. That makes it useful even if we just want to summarize our data.

```
> do(1) * lm( age ~ sex, HELP)            # using the data as is
```

```
  Intercept sexmale sigma r-squared
1     36.25 -0.7841 7.712   0.00187
```

But it really shines when we start looking at sampling distributions.

```
> do(3) * lm( age ~ shuffle(sex), HELP)   # simulation under a null hypothesis
```

```
  Intercept sexmale sigma r-squared
1     36.28 -0.8208 7.711 0.0020493
2     34.64  1.3205 7.698 0.0053031
3     35.83 -0.2335 7.718 0.0001658
```

3.5.4 Generating Fake Data For Sampling Activities

If you don't have actual data at your disposal, you can sometimes simulate data to illustrate your point.

Example 3.9. (Fake Families) We'll simulate 5000 families using a Poisson distribution with a mean of 3 to generate family size. (You can make up whatever elaborate story you like for the population you are considering.)

This example is intended for instructors, not for students.

```
> families <- data.frame(familyid=1:5000, children=rpois(5000,3))
```

Now we generate the people in these families

```
> people <- data.frame(
                familyid = with(families, rep( familyid, children)),
                sibs = with(families, rep( children, children))
                )
```

Computing the mean "family size" two different ways reveals the bias of measuring family size by sampling from children rather than from families.

```
> with(families, mean(children))
```

```
 mean
2.993
> with(people, mean(sibs))
 mean
3.979
```

If the result seems mysterious, the following tables might shed some light.

```
> with(families, table(children))
children
   0    1    2    3    4    5    6    7    8    9   10   12
 253  725 1141 1120  848  510  247   95   45   11    4    1
> with(people, table(sibs))
sibs
   1    2    3    4    5    6    7    8    9   10   12
 725 2282 3360 3392 2550 1482  665  360   99   40   12
```

◇

Once built, you can provide these data frames to your students for a sampling exercise. (See Section 5.1.)

Now that we have discussed various methods for generating random data, it's time to put those skills to good use computing p-values.

3.6 Exercises, Problems, and Activities

3.1

a) Write an alternative-hypothesis simulation where there are two groups: A and B. In your alternative hypothesis, the population mean of A is 90 and the population mean of B is 110. Let the standard deviation be 30 for group A and 35 for group B.

b) You've got two different suppliers of electronic components, C and D. You hypothesize that one of them, D, is somewhat defective. Their components have a lifetime that's distributed exponentially with a mean lifetime of 1000. Supplier C is better, you think, and meets the specified mean lifetime of 2000.

Write an alternative hypothesis simulation of the two suppliers.

3.2 What's the distribution of p-values under the Null Hypothesis?

Many students seem to think that if the Null is true, the p-value will be large. It can help to show what's really going on.

4 Some Biology Specific Applications of R

Several of the examples in this chapter are modifications of things found in the online book *A Little Book of R for Bioinformatics* available via

`http://readthedocs.org/docs/a-little-book-of-r-for-bioinformatics/`

Data used in this chapter are available at

`http://www.calvin.edu/~rpruim/talks/SC11/Seattle/Data/`

and in the `Data` folder inside the `SC11materials` folder in the RStudio server on dahl.

In this chapter we will make use of several specialty package. Check that they are installed and loaded, and install or load them as necessary before continuing.

```
> library(seqinr)
> library(ape)
> library(phangorn)
```

```
> source("http://bioconductor.org/biocLite.R")
> biocLite()                # install core of Bioconductor (could take a while)
> biocLite('Biostrings')    # install Biostrings package from Bioconductor
```

```
> library(Biostrings)       # don't forget to load it after installation
```

4.1 Working With Sequence Data

4.1.1 Reading from FASTA files

FASTA file format is one common way that sequence data is stored. Here is a toy example that shows a simple FASTA file with 8 sequences:

```
>1
GTTAGAG
>2
GTTAGCG
>3
ATCATTA
>4
ATCAATA
>5
```

```
GATCGAC
>6
GATCGAG
>7
AATCAAG
>8
AATCAAA
```

Suppose you have a FASTA file and you want to manipulate the data it contains in R. The `seqinr` package includes a function for reading in such data.

```
> read.fasta('http://www.calvin.edu/~rpruim/talks/SC11/Data/8planets.fasta') -> planets
```

The variable `planets` now contains the eight planet sequences *in a list*.

```
> class(planets)
[1] "list"
> length(planets)
[1] 8
```

We can access individual elements of a list in R using the double-square-bracket operator. Each element includes the nucleotide sequence as well as some additional information.

```
> planets[[1]] -> mercury; mercury
[1] "g" "t" "t" "a" "g" "a" "g"
attr(,"name")
[1] "1"
attr(,"Annot")
[1] ">1  "
attr(,"class")
[1] "SeqFastadna"
> class(mercury)
[1] "SeqFastadna"
> length(mercury)
[1] 7
> table(mercury)
mercury
a g t
2 3 2
```

Digging Deeper
This is part of the object oriented nature of R. When an object is printed, R looks for a function called `print.class()`, where `class` is the class of the object. If it find this function, that's what it uses. If not, then it uses `print.default()`.

Each sequence is stored in a SeqFastadna object, but the default display of these objects is pretty ugly. If we like, we can change what displays when an object of class SeqFastadna is printed by writing a function.

```
> print.SeqFastadna <- function(x,...) {
        cat( paste( attr(x,'name'),' [', attr(x,'Annot'), "]: ", sep="" ) );
        if ( length(x) <= 10 ) {
                cat( x )
        } else {
                cat( paste( x[1: 10] ) )
                cat('...')
        }
        cat('\n')
 }
> mercury
1 [>1  ]: g t t a g a g
```

Alternatively, we can use the `c2s()` function to convert the main sequence data from a sequence of characters to a single string.

```
> c2s(mercury)
[1] "gttagag"
```

A `s2c()` function exists for going in the other direction as well.

```
> s2c('aaccggtt')
[1] "a" "a" "c" "c" "g" "g" "t" "t"
```

Here is a more interesting example, this time using amino acid sequences:

```
> read.fasta('http://www.calvin.edu/~rpruim/talks/SC11/Data/toxins.fasta') -> toxins
> length(toxins)
[1] 60
> toxins[[1]]                  # uses our newly defined print() method
KA125_LYCMC [>KA125_LYCMC]: m n k l p i l i f m...
> table(toxins[[1]])
- a c d e f g h i k l m n p q r s t v y
6 1 8 2 1 2 4 2 4 4 4 4 2 2 1 3 8 3 2 3
> sapply(toxins, length)
KA125_LYCMC KA127_LYCMC KA121_TITSE KA122_TITTR KA124_TITST KA123_TITCO KA159_LYCMC
         66          66          66          66          66          66          66
  KAX_BUTOS KA162_MESMA KA163_MESMA  KA3B_BUTEU KAX17_LEIQH KA161_MESTA KAX11_LEIQH
         66          66          66          66          66          66          66
KAX1C_LEIQH KAX1D_LEIQH KAX12_LEIQH KAX16_MESMA KAX15_MESMA KAX14_CENLM KAX1B_CENNO
         66          66          66          66          66          66          66
KAX13_MESTA KAX1A_PARTR KAX41_TITSE KAX46_TITST KAX45_TITCO KAX44_TITCA KAX43_TITDI
         66          66          66          66          66          66          66
KAX19_CENLM KAX28_CENEL KAX29_CENEL KAX21_CENNO KAX2B_CENEL KAX2C_CENEL KAX22_CENMA
         66          66          66          66          66          66          66
KAX25_CENLM KAX23_CENLL KAX2A_CENEL KAX2D_CENSU KAX32_LEIQH KAX33_LEIQH KAX34_LEIQH
         66          66          66          66          66          66          66
KAX31_ANDMA KAX37_ORTSC KAX24_CENNO KAX64_PANIM KAX6D_HETSP KAX69_OPICA KAX62_SCOMA
         66          66          66          66          66          66          66
KAX6A_OPICA KAX6F_HEMLE KAX72_PANIM KAX66_OPICA KAX67_OPICA KAX68_OPICA KAX63_HETSP
         66          66          66          66          66          66          66
KAX6E_HADGE KAX6C_ANUPH KAX6B_OPIMA KAX52_ANDMA
         66          66          66          66
```

This output shows us the names and lengths of the 60 toxin sequences.

> **Digging Deeper**
> `sapply()` applies a function to each element of a list or vector and returns a vector of values. `lapply()` is similar but returns a list. `apply()` can apply a function to rows or columns of a matrix.

4.1.2 GC content

For DNA sequences, it is easy to compute GC content either "by hand":

```
> proptable( planets[[1]] %in% c('g','c') )
 FALSE   TRUE
0.5714 0.4286
```

or using the `GC()` function.

```
> GC( planets[[1]] )
[1] 0.4286
```

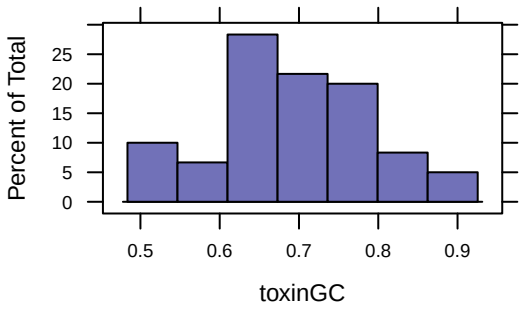
The GC() function is not only simpler, but it knows more. It can also handle amino acid sequences and can make an estimate even when there is missing or ambiguous data.

Digging Deeper
GC1(), GC2(), and GC3() can be used to get GC content in positions 1, 2, or 3 of codons.

```
> GC(toxins[[1]])
[1] 0.75
```

Now let's calculate the GC content for each toxin and display the results in a histogram.

```
> sapply(toxins, GC) -> toxinGC; toxinGC
KA125_LYCMC KA127_LYCMC KA121_TITSE KA122_TITTR KA124_TITST KA123_TITCO KA159_LYCMC
     0.7500      0.5000      0.6471      0.6471      0.6471      0.6875      0.6667
  KAX_BUTOS KA162_MESMA KA163_MESMA  KA3B_BUTEU KAX17_LEIQH KA161_MESTA KAX11_LEIQH
     0.6667      0.6316      0.6471      0.7143      0.7500      0.8000      0.6000
KAX1C_LEIQH KAX1D_LEIQH KAX12_LEIQH KAX16_MESMA KAX15_MESMA KAX14_CENLM KAX1B_CENNO
     0.6000      0.6364      0.6429      0.6250      0.7500      0.6923      0.6429
KAX13_MESTA KAX1A_PARTR KAX41_TITSE KAX46_TITST KAX45_TITCO KAX44_TITCA KAX43_TITDI
     0.9000      0.8000      0.7143      0.7143      0.7857      0.6667      0.5882
KAX19_CENLM KAX28_CENEL KAX29_CENEL KAX21_CENNO KAX2B_CENEL KAX2C_CENEL KAX22_CENMA
     0.5333      0.6923      0.6923      0.6923      0.6923      0.6923      0.6429
KAX25_CENLM KAX23_CENLL KAX2A_CENEL KAX2D_CENSU KAX32_LEIQH KAX33_LEIQH KAX34_LEIQH
     0.6429      0.7692      0.9091      0.7692      0.7692      0.7692      0.7857
KAX31_ANDMA KAX37_ORTSC KAX24_CENNO KAX64_PANIM KAX6D_HETSP KAX69_OPICA KAX62_SCOMA
     0.8333      0.8333      0.6429      0.7500      0.8462      0.5000      0.7333
KAX6A_OPICA KAX6F_HEMLE KAX72_PANIM KAX66_OPICA KAX67_OPICA KAX68_OPICA KAX63_HETSP
     0.6667      0.7692      0.6667      0.5652      0.5455      0.5000      0.7143
KAX6E_HADGE KAX6C_ANUPH KAX6B_OPIMA KAX52_ANDMA
     0.7333      0.7692      0.5294      0.9000
> histogram( ~ toxinGC)
```



4.1.3 Sliding Windows of GC Content

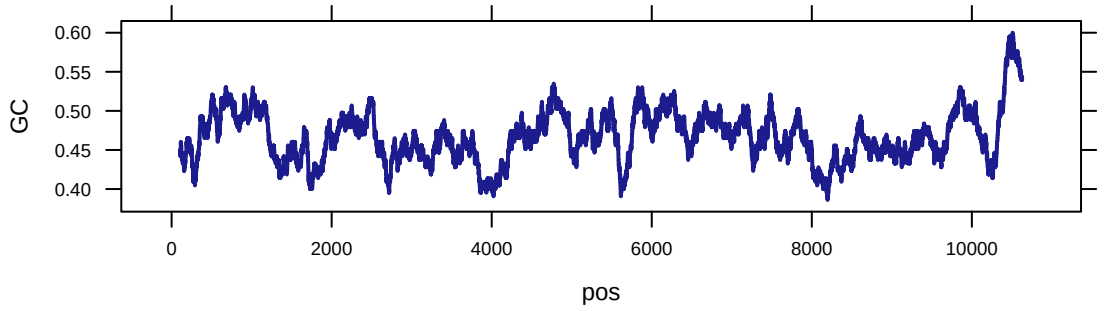
For a longer sequence, it can be interesting to see how the GC content changes across the length of the sequence. A sliding windows approach helps us do this.

```
> dengue <- read.fasta('http://www.calvin.edu/~rpruim/talks/SC11/Data/DengueFever.fasta')
> dengueseq <- dengue[[1]]
```

```
> windowGC <- function(x, windowSize=round(length(x)/100) ){
   positions <- (windowSize+1):(length(x) - windowSize)
   data.frame(
     GC=sapply( positions, function(n) {GC(x[(n-windowSize):(n+windowSize)])}),
     pos = positions
```

```
    )
}
```

```
> xyplot( GC ~ pos, data=windowGC(dengueseq) , cex=.05, type='l')
```



4.1.4 Counting k-words

A k-word is a subsequence of k consecutive letters (bases or codons).

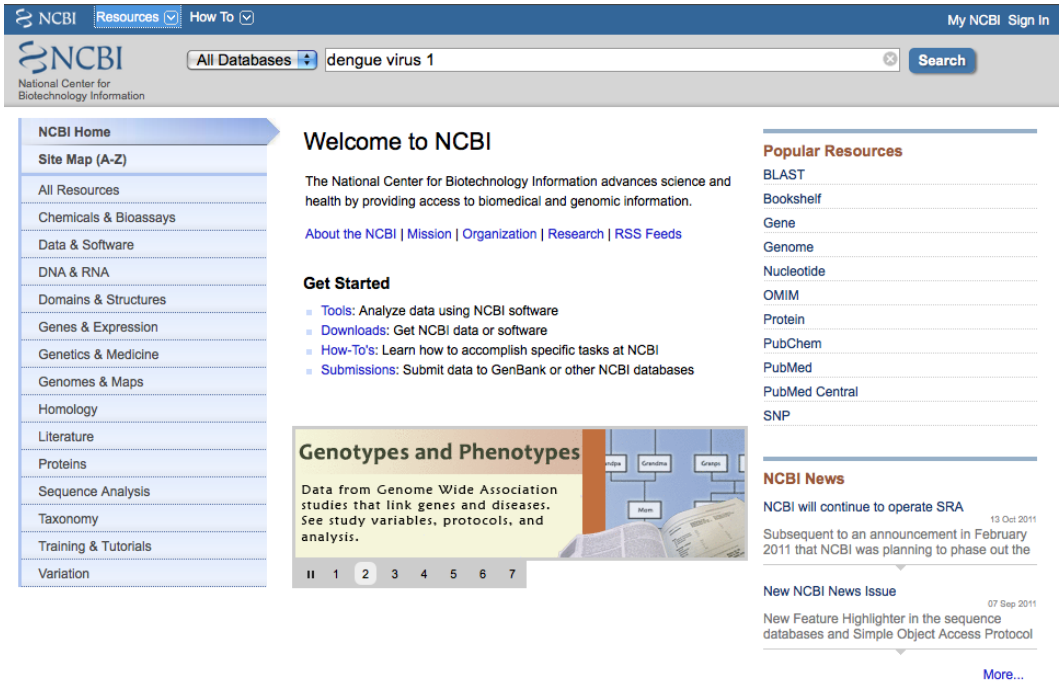
```
> alphabet = sort(unique(toxins[[5]]))
> count(toxins[[5]], 1, alphabet=alphabet) -> c1; c1[c1>0]

 -  a  c  d  e  f  g  h  k  l  m  n  p  r  s  t  w  y
26  4  8  2  1  2  3  1  3  2  1  2  1  3  2  2  1  2

> count(toxins[[5]], 2, alphabet=alphabet) -> c2; c2[c2>0]

-- -c -k -w a- af ah as cf cg cl cm cr cs cy dl dp ec fg fk g- ga gr hg ka kc la ld
22  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  2  1  1
mn nk nn pc ra rc re sr st t- tc wc yd yt
 1  1  1  1  1  1  1  1  1  1  1  1  1  1

> count(toxins[[5]], 3, alphabet=alphabet) -> c3; c3[c3>0]

--- --c --w -cg -kc -wc a-- afg ahg asr cfk cga cld cmn crc cst cyd cyt dla dpc ecy
 19   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
fgr fka g-k gas gra hg- kaf kcm kcr la- ldl mnn nkc nnk pcf rah rcy rec sre stc t--
  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
tcl wcs ydp yt-
  1   1   1   1
```

4.2 Obtaining Sequence Data

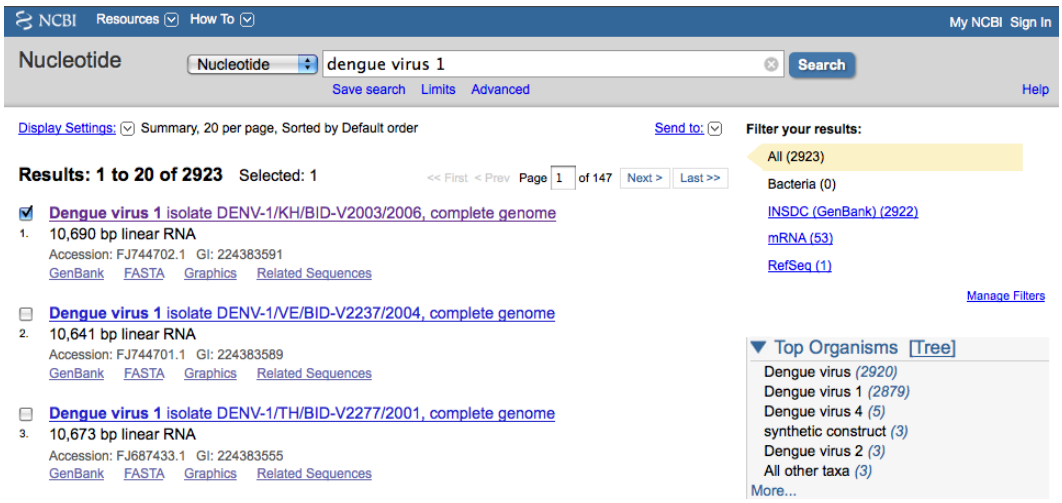
4.2.1 Via Your Browser

You can get sequence data for your favorite organisms at `http://www.ncbi.nlm.nih.gov`.

Enter your search criterion and click on `Search`.



Then choose Nucleotide (or protein) on the next page, and then select the particular sequence or sequences you want here:



Finally, click on FASTA here

and use copy and paste to save the file.

4.2.2 Searching Sequence Databases from Within R

We have already seen that you can work with any sequences you can find in fasta format, so all of the web utilities you know and love that can export in fasta format are now sources for data you can use in R. `seqinr` also provides some facilities for searching databases from inside R.

CAUTION!
While the functionality provided by `seqinr` is very useful, some of the design choices made by the authors could have been better.

The `seqinr` package was written by the group that created the ACNUC database in Lyon, France (`http://pbil.univ-lyon1.fr/databases/acnuc/acnuc.html`). The ACNUC database is a database that contains most of the data from the NCBI Sequence Database, as well as data from other sequence databases such as UniProt and Ensembl. Here is a list of available databases.

```
> choosebank()
 [1] "genbank"       "embl"          "emblwgs"       "swissprot"     "ensembl"
 [6] "hogenom"       "hogenomdna"    "hovergendna"   "hovergen"      "hogenom4"
[11] "hogenom4dna"   "homolens"      "homolensdna"   "hobacnucl"     "hobacprot"
[16] "phever2"       "phever2dna"    "refseq"        "nrsub"         "greviews"
[21] "bacterial"     "protozoan"     "ensbacteria"   "ensprotists"   "ensfungi"
[26] "ensmetazoa"    "ensplants"     "mito"          "polymorphix"   "emglib"
[31] "taxobacgen"    "refseqViruses"
```

Alas, the ACNUC sub-databases do not have a one-to-one correspondence with the NCBI sub-databases (the NCBI Protein database, NCBI EST database, NCBI Genome database, etc.)!

Three of the most important sub-databases in ACNUC which can be searched from R are:

- genbank: this contains DNA and RNA sequences from the NCBI Sequence Database, except for certain classes of sequences (eg. draft genome sequence data from genome sequencing projects)
- refseq: this contains DNA and RNA sequences from Refseq, the curated part of the NCBI Sequence Database
- refseqViruses: this contains DNA, RNA and proteins sequences from viruses from RefSeq

You can find more information about what each of these ACNUC databases contains by looking at the ACNUC website.

Searching for sequences is done by

1. choosing a databse with `choosebank()`

```
> choosebank('refseq')
```

2. forming a query and using `query()` to get the results.

   The query language allows us to search by accession, author, keyword, journal, journal reference (journal, volume, page), etc. See the documentation for `query()` to get a full description of the queries that are possible. This examples would (if we ran it) find all the human sequences in refSeq.

   ```
   > choosebank('refseq')
   > query('queryResults', "SP=Homo sapiens")
   > queryResults
   45345 SQ for SP=Homo sapiens
   ```

   CAUTION! Please don't all run this query simultaneously. Thanks, the Management.

   A more modest (and responsive) query uses a narrower search:

   ```
   > choosebank("swissprot")
   > query("twoProteins", "AC=Q9CD83 or AC=A0PQ23")
   ```

3. looking through the results of the search to identify the sequence(s) of interest.

   In this case we are interested in both.

   ```
   > twoProteins
   2 SQ for AC=Q9CD83 or AC=A0PQ23
   ```

   These objects contain information about the results of our query. For example, we can find out a bit about each sequence that was located.

   ```
   > twoProteins$req
   [[1]]
            name          length           frame          ncbicg
   "A0PQ23_MYCUA"           "212"             "0"             "1"

   [[2]]
          name         length          frame         ncbicg
   "PHBS_MYCLE"          "210"            "0"            "1"
   ```

4. retrieving the sequence(s) from the database.

   The command above does not actually fetch the sequence data, only meta-data about the sequences. Now let's fetch the sequences of interest.

   ```
   > lepraeseq <- getSequence(twoProteins$req[[1]])
   > ulceransseq <- getSequence(twoProteins$req[[2]])
   > bothseq <- getSequence(twoProteins$req)
   > class(lepraeseq)
   [1] "character"
   > class(bothseq)
   [1] "list"
   ```

   And take a little look

   ```
   > c2s(bothseq[[1]][1:50])
   [1] "MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEI"
   > c2s(lepraeseq[1:50])
   [1] "MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLANDEI"
   > c2s(ulceransseq[1:50])
   [1] "MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVANEEIVVDIINQQLL"
   ```

5. closing the connection to the database

```
> closebank()
```

The resulting sequences can be saved to a file using `write.fasta()`

```
> write.fasta( sequences=list(lepraeseq,ulceransseq), names=c('leprae','ulcerans'),
              file='TwoProteins.fasta')
```

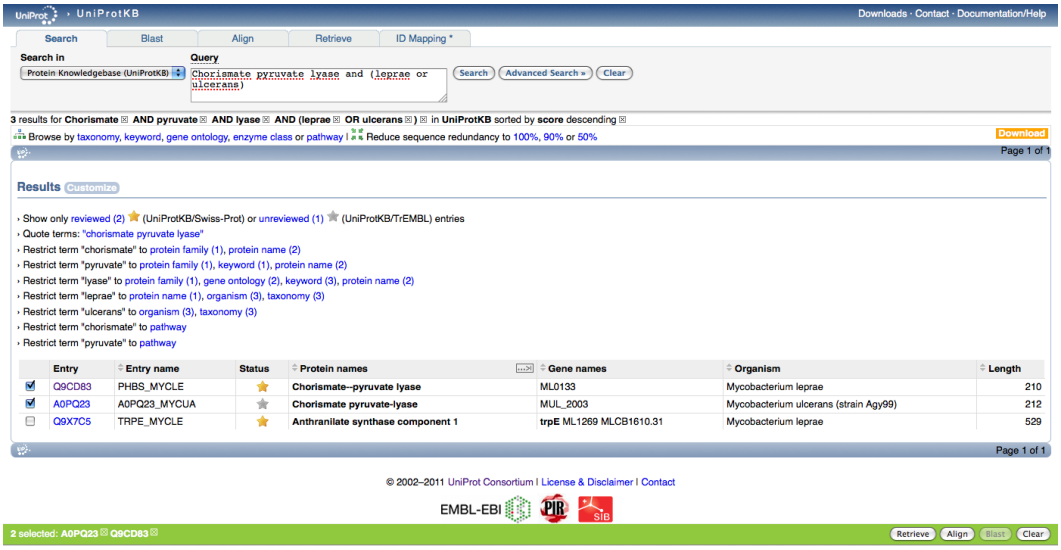
Note that `write.fasta()` expects the sequences as vectors of individual characters (so we don't use `c2s()` here) and that to provide more than one sequence, we must put them into a list rather than a vector because vectors cannot contain vectors.

4.3 Sequence Alignment

4.3.1 Pairwise Alignment

Let's compare the protein sequences of the chorismate lyase protein in two bacteria. *Mycobacterium leprae* is the bacterium which causes leprosy, while *Mycobacterium ulcerans* is a related bacterium which causes Buruli ulcer, both of which are classified by the WHO as neglected tropical diseases.

We can retrieve these sequences in FASTA format from `http://www.uniprot.org/`.

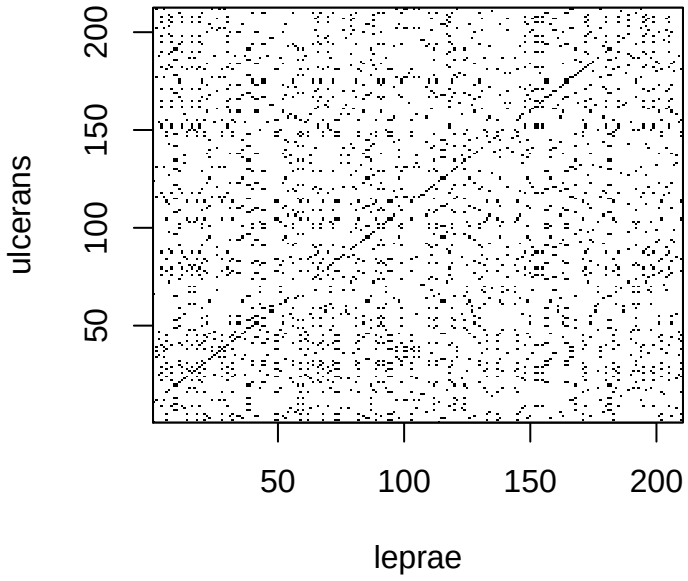


If we save the results in FASTA format, we can read them into R as before.

```
> proteins <- read.fasta('http://www.calvin.edu/~rpruim/talks/SC11/Data/TwoProteins.fasta')
> leprae <- proteins[[1]]
> ulcerans <- proteins[[2]]
```

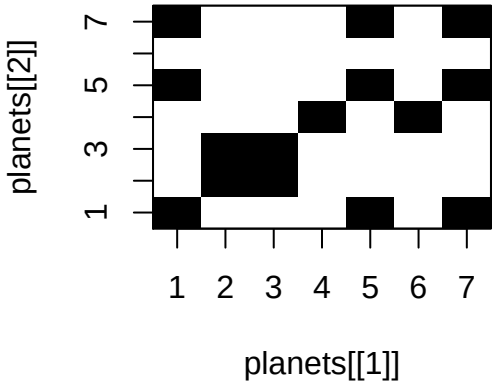
A dot plot allows us to visually inspect the alignment.

```
> dotPlot(leprae, ulcerans)
```



It can be helpful for students to look at a very small example of a dot plot to understand how they work.

```
> planets[[1]]
1 [>1  ]: g t t a g a g
> planets[[2]]
2 [>2  ]: g t t a g c g
> dotPlot(planets[[1]], planets[[2]])
```



The `pairwiseAlignment()` function in the `Biostrings` package allows us to compute a pairwise global alignment using the Needleman-Wunsch algorithm. Note that `pairwiseAlignment()` requires that the sequence be represented as a single string rather than as a vector of characters.

```
> pairwiseAlignment(c2s(leprae), c2s(ulcerans))
```

```
Global PairwiseAlignedFixedSubject (1 of 1)
pattern: [1] m----------tnrtlsreeirkldrdlrilvatngt...svfqdtpreeldrcqysndidtrsgdrfvlhgrvfkn
subject: [1] mlavlpekremtechlsdeeirklnrdlriliatngt...svfednsreepirhqrsvgtsars-------grsict
score: -76.08
```

We can tune the algorithm by selecting a substitution matrix, and penalties for opening and extending gaps. `Biostrings` contains several substitution matrices that we can use, including BLOSUM50. We must convert to upper case to use these substitution matrices.

```
> pairwiseAlignment(
   toupper(c2s(leprae)), toupper(c2s(ulcerans)),
   substitutionMatrix = 'BLOSUM50', gapOpening = -2, gapExtension = -8 )
```

```
Global PairwiseAlignedFixedSubject (1 of 1)
pattern: [1] MT-----NR--T---LSREEIRKLDRDLRILVATNGT...SVFQDTPREELDRCQYSNDIDTRSGDRFVLHGRVFKN
subject: [1] MLAVLPEKREMTECHLSDEEIRKLNRDLRILIATNGT...SVFEDNSREEPIRHQRS--VGT-SA-R---SGRSICT
score: 627
```

Alternatively, we can do a local alignment using the Smith-Waterman algorithm.

```
> pairwiseAlignment(
    toupper(c2s(leprae)), toupper(c2s(ulcerans)),
    substitutionMatrix = 'BLOSUM50', gapOpening = -2, gapExtension = -8, type='local')
```

```
Local PairwiseAlignedFixedSubject (1 of 1)
pattern:  [1] MTNRTLSREEIRKLDRDLRILVATNGTLTRVLNVVAN...QPVIITTEYFLRSVFQDTPREELDRCQYSNDIDTRSG
subject: [11] MTECHLSDEEIRKLNRDLRILIATNGTLTRILNVLAN...RPVIIITEYFLRSVFEDNSREEPIRHQRSVGTSARSG
score: 761
```

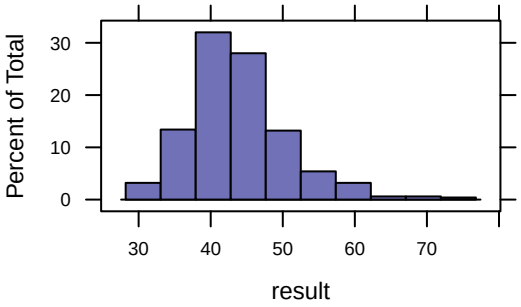
To get a sense for how good the alignment is, we can compare the alignment we would get with a random sequence that has the same allele frequency.

```
> pairwiseAlignment(
   toupper(c2s(leprae)), toupper(c2s(rdata(length(ulcerans),ulcerans))),
   substitutionMatrix = 'BLOSUM50', gapOpening = -2, gapExtension = -8, type='local',
   scoreOnly=TRUE)
```

```
[1] 38
```

```
> scores <- do(500) *
           pairwiseAlignment(toupper(c2s(leprae)), toupper(c2s(rdata(length(ulcerans),ulcerans))),
                   substitutionMatrix = 'BLOSUM50', gapOpening = -2, gapExtension = -8,
                                  type='local', scoreOnly=TRUE)
```

```
> histogram(~ result, data=scores)
```



4.3.2 Multiple Alignment

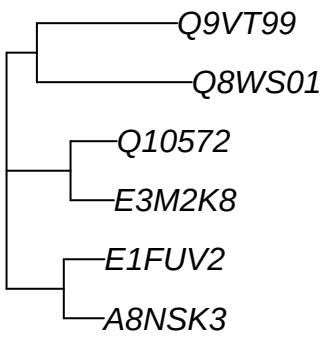
Multiple alignment is typically done in specialized software tools (e.g., MUSCLE, clustalw) rather than in R. Alternatively, R can call an alignment program and let it do the heavy lifting. Indeed, the function `seqaln()` in the `bio3d` package provides an interface to MUSCLE that works this way (provided MUSCLE is also properly installed on the same machine).

We can save the output from clustalw in a FASTA file and read in the alignments using `read.alignment()`:

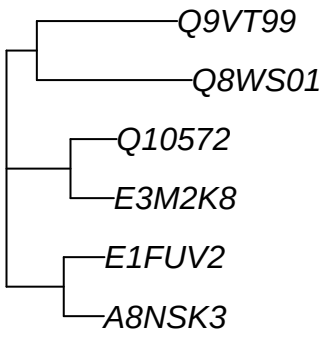
```
> # read from web using this
> # read.alignment(file='http://www.calvin.edu/~rpruim/talks/SC11/Data/viruses-aln.fasta',
> #   format='fasta') -> virus_aln
> # read from local file using this
> read.alignment(file='Data/viruses-aln.fasta', format='fasta') -> virus_aln
> dm <- dist.alignment(virus_aln)        # compute distance matrix
> dm
       Q10572 E3M2K8 E1FUV2 A8NSK3 Q9VT99
E3M2K8 0.2249
E1FUV2 0.5176 0.5176
A8NSK3 0.5238 0.5153 0.2032
Q9VT99 0.7181 0.7088 0.6670 0.6579
Q8WS01 0.7319 0.7221 0.7223 0.7223 0.7416
```

The `phangorn` package provides several algorithms for constructing phylogenetic trees from sequence data.

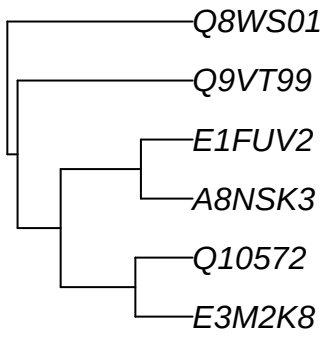
```
> plot(NJ(dm))    # neighbor joining
```



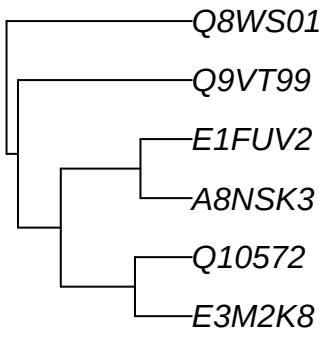
```
> plot(UNJ(dm))    # unweighted neighbor joining
```




```
> plot(upgma(dm))   # Unweighted Pair Group Method with Arithmetic Mean
```



```
> plot(wpgma(dm))   # Weighted Pair Group Method with Arithmetic Mean
```



5 Taking Advantage of the Internet

The Internet provides a wealth of data spanning the world, access to sophisticated statistical computing, and a practical means for you to communicate with your own students. In this chapter, we'll illustrate some mundane ways for you to distribute and share data and software with your students, web-based interfaces for statistical computing, as well as tools for "scraping" data from the Internet using application program interfaces (API's) or through XML (eXtensible Markup Language).

We draw your attention particularly to provocative papers by Gould [Gou10] and Nolan and Temple Lang [NT10], who highlight the importance of broadening the type of data students encounter in their first courses as well as the role of computing in modern statistics, respectively.

The wealth of data accessible to students on the internet continues to increase at what feels like an exponential rate.

5.1 Sharing With and Among Your Students

Instructors often have their own data sets to illustrate points of statistical interest or to make a particular connection with a class. Sometimes you may want your class as a whole to construct a data set, perhaps by filling in a survey or by contributing their own small bit of data to a class collection. Students may be working on projects in small groups; it's nice to have tools to support such work so that all members of the group have access to the data and can contribute to a written report.

There are now many technologies for supporting such sharing. For the sake of simplicity, we will emphasize three that we have found particularly useful both in teaching statistics and in our professional collaborative work. These are:

- A web site with minimal overhead, such as provided by Dropbox.
- The services of Google Docs.
- A web-based RStudio server for R.

The first two are already widely used in university environments and are readily accessible simply by setting up accounts. Setting up an RStudio web server requires some IT support, but is well within the range of skills found in IT offices and even among some individual faculty.

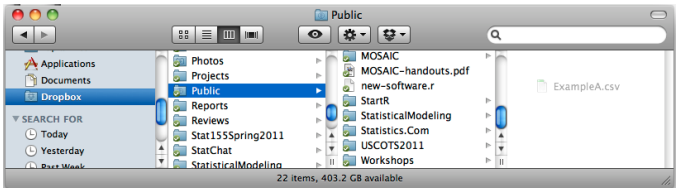


Figure 5.1: Dragging a CSV file to a Dropbox Public directory

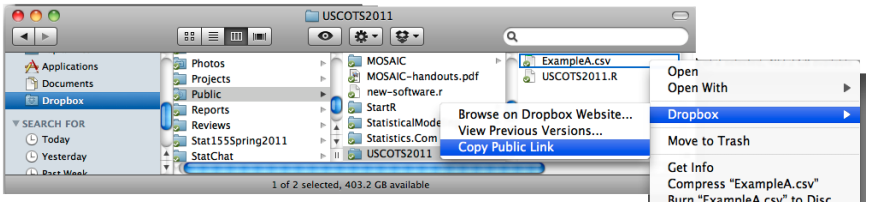


Figure 5.2: Getting the URL of a file in a Dropbox Public directory

5.1.1 Your Own Web Site

Our discussion of Dropbox is primarily for those who do not already know how to do this other ways.

You may already have a web site. We have in mind a place where you can place files and have them accessed directly from the Internet. For sharing data, it's best if this site is public, that is, it does not require a login. That rules out most "course support" systems such as Moodle or Blackboard.

The Dropbox service for storing files in the "cloud" provides a very convenient way to distribute files over the web. (Go to `dropbox.com` for information and to sign up for a free account.) Dropbox is routinely used to provide automated backup and coordinated file access on multiple computers. But the Dropbox service also provides a PUBLIC directory. Any files that you place in that directory can be accessed directly by a URL.

To illustrate, suppose you wish to share some data set with your students. You've constructed this data set in a spreadsheet and stored it as a CSV file, let's call it "example-A.csv". Move this file into the PUBLIC directory under Dropbox — on most computers Dropbox arranges things so that its directories appear exactly like ordinary directories and you'll use the ordinary familiar file management techniques as in Figure 5.1.

Dropbox also makes it straightforward to construct the web-location identifying URL for any file by using mouse-based menu commands to place the URL into the clipboard, whence it can be copied to your course-support software system or any other place for distribution to students. For a CSV file, reading the contents of the file into R can be done with the `read.csv()` function, by giving it the quoted URL:

The history feature in RStudio can be used to re-run this command in future sessions

```
> a <- read.csv("http://dl.dropbox.com/u/5098197/USCOTS2011/ExampleA.csv")
```

This technique makes it easy to distribute data with little advance preparation. It's fast enough to do in the middle of a class: the CSV file is available to your students (after a brief lag while Dropbox synchronizes). It can even be edited by you (but not by your students).

The same technique can be applied to all sorts of files: for example, R workspaces or even R scripts. Of course, your students need to use the appropriate R command: `load()` for a workspace or `source()` for a script.

It's a good idea to create a file with your course-specific R scripts, adding on to it and modifying it as the course progresses. This allows you to distribute all sorts of special-purpose functions, letting you

distribute new R material to your students. For instance, that brilliant new "manipulate" idea you had at 2am can be programmed up and put in place for your students to use the next morning in class. Then as you identify bugs and refine the program, you can make the updated software immediately available to your students.

For example, in the next section of this book we will discuss reading directly from Google Spreadsheets. It happens that we wanted to try a new technique but were not sure that it was worth including in the `mosaic` package. So, we need another way to distribute it to you. Use this statement:

```
> source("http://dl.dropbox.com/u/5098197/USCOTS2011/USCOTS2011.R")
```

Among other things, the operator `readGoogleCSV()` is defined in the script that gets sourced in by that command. Again, you can edit the file directly on your computer and have the results instantly available (subject only to the several second latency of Dropbox) to your students. Of course, they will have to re-issue the `source()` command to re-read the script.

CAUTION!
Security through Obscurity of this sort will not generally satisfy institutional data protection regulations

If privacy is a concern, for instance if you want the data available only to your students, you can effectively accomplish this by giving files names known only to your students, e.g., "Example-A78r423.csv".

5.1.2 GoogleDocs

The Dropbox technique is excellent for broadcasting: taking files you create and distributing them in a read-only fashion to your students. But when you want two-way or multi-way sharing of files, other techniques are called for, such as provided by the GoogleDocs service.

GoogleDocs allows students and instructors to create various forms of documents, including reports, presentations, and spreadsheets. (In addition to creating documents *de novo*, Google will also convert existing documents in a variety of formats.)

Once on the GoogleDocs system, the documents can be edited *simultaneously* by multiple users in different locations. They can be shared with individuals or groups and published for unrestricted viewing and even editing.

For teaching, this has a variety of uses:

- Students working on group projects can all simultaneously have access to the report as it is being written and to data that is being assembled by the group.
- The entire class can be given access to a data set, both for reading and for writing.
- The Google Forms system can be used to construct surveys, the responses to which automatically populate a spreadsheet that can be read by the survey creators.
- Students can "hand in" reports and data sets by copying a link into a course support system such as Moodle or Blackboard, or emailing the link.
- The instructor can insert comments and/or corrections directly into the document.

An effective technique for organizing student work and ensuring that the instructor (and other graders) have access to it, is to create a separate Google directory for each student in your class (Dropbox can also be used in this manner). Set the permission on this directory to share it with the student. Anything she or he drops into the directory is automatically available to the instructor. The student can also share with specific other students (e.g., members of a project group).

Example 5.1. One exercise for students starting out in a statistics course is to collect data to find out whether the "close door" button on an elevator has any effect. This is an opportunity to introduce simple ideas of experimental design. But it's also a chance to teach about the organization of data.

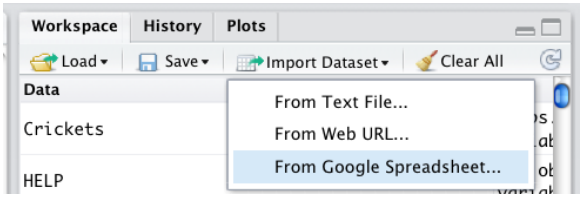
Have your students, as individuals or small groups, study a particular elevator, organize their data into a spreadsheet, and hand in their individual spreadsheet. Then review the spreadsheets in class. You will likely find that many groups did not understand clearly the distinction between cases and variables, or coded their data in ambiguous or inconsistent ways.

Work with the class to establish a consistent scheme for the variables and their coding, e.g., a variable `ButtonPress` with levels "Yes" and "No", a variable `Time` with the time in seconds from a fiducial time (e.g. when the button was pressed or would have been pressed) with time measured in seconds, and variables `ElevatorLocation` and `GroupName`. Create a spreadsheet with these variables and a few cases filled in. Share it with the class.

Have each of your students add his or her own data to the class data set. Although this is a trivial task, having to translate their individual data into a common format strongly reinforces the importance of a consistent measurement and coding system for recording data. ◇

Once you have a spreadsheet file in GoogleDocs, you will want to open it in R. Of course, it's possible to export it as a CSV file, then open it using the CSV tools in R, such as `read.csv()`. But there are easier ways that let you work with the data "live."

In the web-server version of RStudio, described below, you can use a menu item to locate and load your spreadsheet.



If you are using other R interfaces, you must first use the Google facilities for publishing documents.

1. From within the document, use the "Share" dropdown menu and choose "Publish as a Web Page."
2. Press the "Start Publishing" button in the "Publish to the web" dialog box. (See figure 5.3.)
3. In that dialog box, go to "Get a link to the published data." Choose the CSV format and copy out the link that's provided. You can then publish that link on your web site, or via course-support software. Only people with the link can see the document, so it remains effectively private to outsiders.

It turns out that communicating with GoogleDocs requires facilities that are not present in the base version of R, but are available through the `RCurl` package. In order to make these readily available to students, we have created a function that takes a quoted string with the Google-published URL and reads the corresponding file into a data frame:

```
> elev <- readGoogleCSV(
 "https://spreadsheets.google.com/spreadsheet/pub?hl=en&hl=en&key=0Am13enSalO74dEVzMGJSMU5TbTc2eWlWakppQlpj
> head(elev)
  StudentGroup      Elevator CloseButton  Time Enroute LagToPress
1           HA Campus Center           N 8.230       N          0
2           HA Campus Center           N 7.571       N          0
3           HA Campus Center           N 7.798       N          0
```

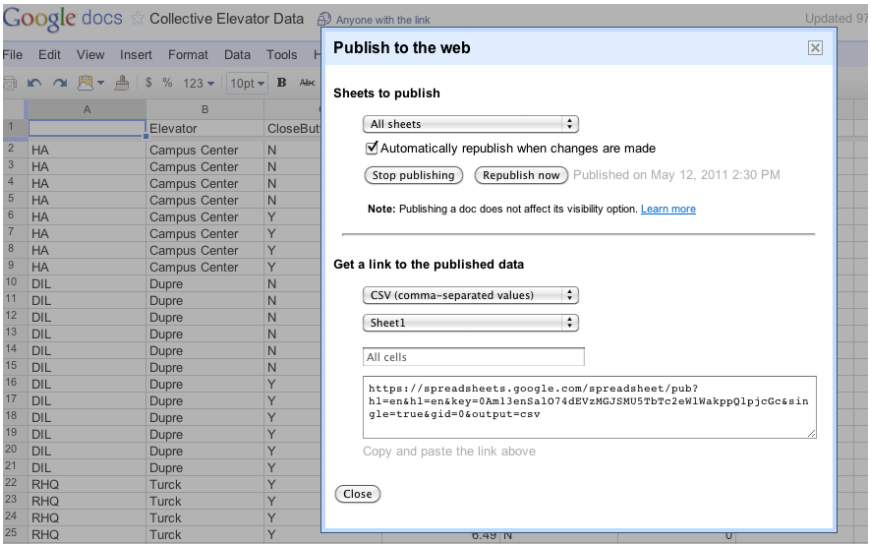


Figure 5.3: Publishing a Google Spreadsheet so that it can be read directly into R.

```
4          HA Campus Center          N 8.303      N         0
5          HA Campus Center          Y 5.811      N         0
6          HA Campus Center          Y 6.601      N         0
```

Of course, you'd never want your students to type that URL by hand; you should provide it in a copy-able form on a web site or within a course support system.

Note that the `readGoogleCSV()` function is not part of the `mosaic` package. As described previously, we make it available via an R source file that can be read into the current session of R using the `source()` command:

```
> source("http://dl.dropbox.com/u/5098197/USCOTS2011/USCOTS2011.R")
```

5.1.3 The RStudio Web Server

RStudio is available as a desktop application that provides a considerately designed interface to the standard R software that you can install on individual computers.

But there is another version of RStudio available, one that takes the form of a web server. There are some substantial advantages to using the web-server version.

- For the user, no installation is required beyond a standard web browser.
- Sessions are continued indefinitely; you can come back to your work exactly where you left it.
- A session started on one computer can be continued on another computer. So a student can move seamlessly from the classroom to the dorm to a friend's computer.
- The web-server system provides facilities for direct access to GoogleDocs.

As RStudio continues to be developed, we anticipate facilities being added that will enhance even more the ability to teach with R:

CAUTION!
These are anticipated future features.

- The ability to create URLs that launch RStudio and read in a data set all in a single click.

- The ability to share sessions simultaneously, so that more than one person can be giving commands. This will be much like Google Docs, but with the R console as the document. Imagine being able to start a session, then turn it over to a student in your classroom to give the commands, with you being able to make corrections as needed.
- The ability to clone sessions and send them off to others. For instance, you could set up a problem then pass it along to your students for them to work on.

But even with the present system, the web-based RStudio version allows you to work with students effectively during office hours. You can keep your own version of RStudio running in your usual browser, but give your visiting a student a window in a new browser: Firefox, Chrome, Safari, Internet Explorer, etc. Each new browser is effectively a new machine, so your student can log in securely to his or her own account.

5.2 Data Mining Activities

We end this chapter with several examples that do data mining via the Internet. Some of these are mere glimpses into what might be possible as tools for accessing this kind of data become more prevalent and easier to use.

5.2.1 What percentage of Earth is Covered with Water?

We can estimate the proportion of the world covered with water by randomly sampling points on the globe and inspecting them using GoogleMaps.

First, let's do a sample size computation. Suppose we want to estimate (at the 95% confidence level) this proportion within ±5%. There are several ways to estimate the necessary sample size, including algebraically solving

$$(1.96)\sqrt{\hat{p}(1-\hat{p})/n} = 0.05$$

for n given some estimated value of $\hat{p}$. The `uniroot()` function can solve this sort of thing numerically. Here we take an approach that looks at a table of values of n and $\hat{p}$ and margin of error.

```
> n <- seq(50,500, by=50)
> p.hat <- seq(.5, .9, by=0.10)
> margin_of_error <- function(n, p, conf.level=.95) {
        -qnorm( (1-conf.level)/2) * sqrt( p * (1-p) / n )
 }
> # calculate margin of error for all combos of n and p.hat
> outer(n, p.hat, margin_of_error) -> tbl
> colnames(tbl) <- p.hat
> rownames(tbl) <- n
> tbl
```

```
        0.5     0.6     0.7     0.8     0.9
50  0.13859 0.13579 0.12702 0.11087 0.08315
100 0.09800 0.09602 0.08982 0.07840 0.05880
150 0.08002 0.07840 0.07334 0.06401 0.04801
200 0.06930 0.06790 0.06351 0.05544 0.04158
250 0.06198 0.06073 0.05681 0.04958 0.03719
300 0.05658 0.05544 0.05186 0.04526 0.03395
350 0.05238 0.05132 0.04801 0.04191 0.03143
400 0.04900 0.04801 0.04491 0.03920 0.02940
450 0.04620 0.04526 0.04234 0.03696 0.02772
500 0.04383 0.04294 0.04017 0.03506 0.02630
```

From this it appears that a sample size of approximately 300–400 will get us the accuracy we desire. A class of students can easily generate this much data in a matter of minutes if each student inspects 10–20 maps. The example below assumes a sample size of 10 locations per student. This can be adjusted depending on the number of students and the desired margin of error.

1. Generate 10 random locations.

```
> positions <- rgeo(10); positions
```

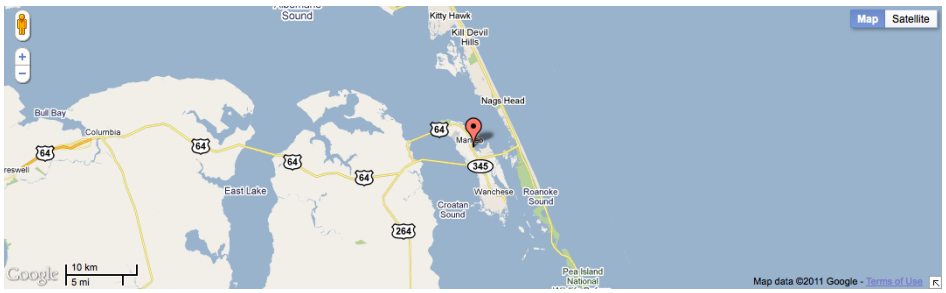
```
       lat     lon
1  -54.910 132.836
2    6.848 -17.641
3  -51.543 -73.535
4  -60.924  -7.936
5   -4.447 -57.897
6   15.100 -26.120
7  -64.686 159.347
8   66.661  46.460
9  -32.937 140.952
10 -15.802 169.742
```

2. Open a GoogleMap centered at each position.

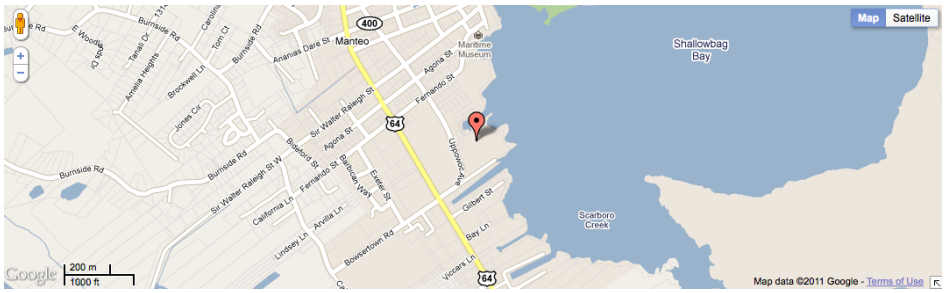
```
> googleMap(pos=positions, mark=TRUE)
```

   You may need to turn off pop-up blocking for this to work smoothly.

3. For each map, record whether the center is located in water or on land. The options `mark=TRUE` is used to place a marker at the center of the map (this is helpful for locations that are close to the coast).

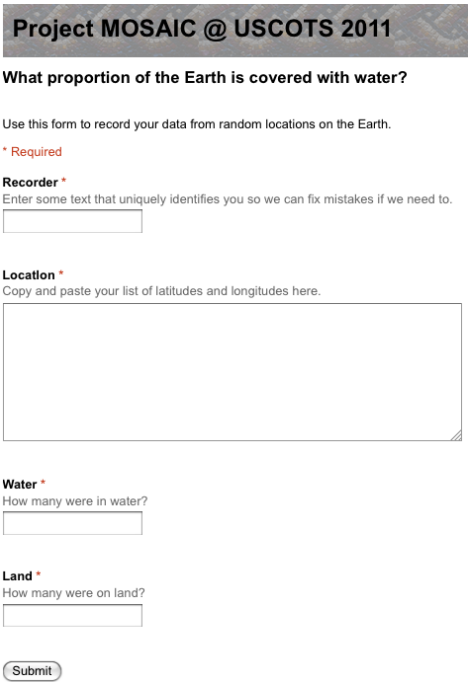


   You can zoom in or out to get a better look.



4. Record your data in a GoogleForm at

   `http://mosaic-web.org/uscots2011/google-water.html`



CAUTION!
This sort of copy-and-paste operation works better in some browsers (Firefox) than in others (Safari).

For the latitude and longitude information, simply copy and paste the output of

```
> positions
```

5. After importing the data from Google, it is simple to sum the counts across the class.

```
> sum(googleData$Water)
[1] 215
> sum(googleData$Land)
[1] 85
```

Then use your favorite method of analysis, perhaps `binom.test()`.

```
> interval(binom.test(215, 300))   # numbers of successes and trials
Method: Exact binomial test

probability of success
                0.7167

95% confidence interval:
0.662 0.767
```

5.2.2 Roadless America

The `rgeo()` function can also sample within a latitude longitude "rectangle". This allows us to sample subsets of the globe. In this activity we will estimate the proportion of the continental United States that is within 1 mile of a road.

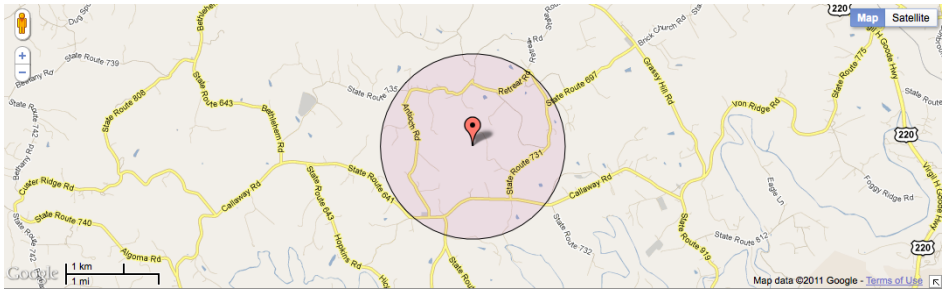
1. Generate a random sample of locations in a box containing the continental United States. Some of these points may be in Canada, Mexico, an ocean or a major lake. These will be discarded from our sample before making our estimate.

```
> positions <- rgeo(10, lonlim=c(-125,-65), latlim=c(25,50)); positions
```

```
     lat     lon
1  42.02 -113.53
2  40.55  -94.14
3  42.88  -71.02
4  45.00  -80.19
5  35.33 -117.09
6  33.62  -97.26
7  48.96 -115.82
8  40.33  -85.28
9  49.48  -98.56
10 38.88 -123.51
```

2. Open a GoogleMap centered at each position. This time we'll zoom in a bit and add a circle of radius 1 to our map.

```
> googleMap(pos=positions, mark=TRUE, zoom=12, radius=1)
```



You may need to turn off pop-up blocking for this to work smoothly.

3. For each map, record whether the center is close (to a road), far (from a road), water, or foreign. You may need to zoom in or out a bit to figure this out.

5.2.3 Variations on the Google Maps theme

There are many other quantities one could estimate using these tools. For example:

1. What proportion of your home state is within m miles of a lake? (The choice of m may depend upon your state of interest.)
2. Use two proportion procedures or chi-squared tests to compare states or continents. Do all continents have roughly the same proportion of land withing m miles of water (for some m)? Are Utah and Arizona equally roadless?
3. In more advanced classes: What is the average distance to the nearest lake (in some region)? By using concentric circles, one could estimate this from discretized data indicating, for example, whether the nearest lake is within 1/2 mile, between 1/2 mile and 1 mile, between 1 mile and 2 miles, between 2 miles, and 4 miles, between 4 miles and 10 miles, or more than 10 miles away. It may be interesting to discuss what sort of model should be used for distances from random locations to lakes. (It probably isn't normally distributed.)

A More About R

This material is more advanced than students in Intro Stats need, but is good for more advanced students and instructors to know.

A.1 Installing and Using Packages

R is open source software. Its development is supported by a team of core developers and a large community of users. One way that users support R is by providing **packages** that contain data and functions for a wide variety of tasks.

A.1.1 Installing packages from CRAN

If you need to install a package, most likely it will be on CRAN. Before a package can be used, it must be **installed** (once per computer) and **loaded** (once per R session). For example, to use `mosaic`:

```
> install.packages("mosaic")   # fetch package from CRAN to local machine.
> require(mosaic)              # load the package so it can be used.
```

If you are running on a machine where you don't have privileges to write to the default library location, you can install a personal copy of a package. If the location of your personal library is first in `R_LIBS`, this will probably happen automatically. If not, you can specify the location manually:

```
> install.packages("mosaic", lib="~/R/library")
```

On a networked machine, be sure to use a different local directory for each platform since packages must match the platform.

Installing packages on a Mac or PC is something you might like to do from the GUI since it will provide you with a list of packages from which you can select the ones of interest. Binary packages have been precompiled for a particular platform and are generally faster and easier to set up, if they are available. Source packages need to be compiled and built on your local machine. Usually this happens automatically – provided you have all the necessary tools installed on your machine – so the only disadvantage is the extra time it takes to do the compiling and building.

A.1.2 Installing other packages

Occasionally you might find a package of interest that is not available via a repository like CRAN. Typically, if you find such a package, you will also find instructions on how to install it. If not, you can usually install directly from the zipped up package file.

```
> install.packages('some-package.tar.gz',
                   repos=NULL)          # use a file, not a repository
```

A.1.3 Finding packages

There are several ways to find packages

- Ask your friends.
- Google: Put 'cran' in the search.
- Rseek: `http://rseek.org` provides a search engine specifically designed to find information about R.
- CRAN task views.

  A number of folks have put together task views that annotate a large number of packages and summarize they are good for. They are organized according to themes. Here are a few examples of available task views:

Bayesian	Bayesian Inference
Econometrics	Computational Econometrics
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays, Dynamic Graphics, Graphic Devices, and Visualization
Multivariate	Multivariate Statistics
SocialSciences	Statistics for the Social Sciences

- Bioconductor (`http://www.bioconductor.org/`) and Omegahat (`http://www.omegahat.org/R`) are another sources of packages (specify the `repos=` option to use these).
- *R Journal* (formerly *R News*) is available via CRAN and often has articles about new packages and their capabilities.
- Write your own.

  You can write your own packages, and it isn't that hard to do (but we won't cover this here).

A.2 Some Workflow Suggestions

In short: *Think like a programmer.*

- Use R interactively only to get documentation and for quick one-offs.
- Store your code in a file.

  You can execute all the code in a file using

```
> source("file.R")
```

RStudio has options for executing some or all lines in a file, too. See the buttons in the panel for any R script. (You can create a new R script file in the main file menu.)

If you work at the interactive prompt in the console and later wish you had been putting your commands into a file, you can save your past commands with

```
> savehistory("someRCommandsIalmostLost.R")
```

RStudio makes this especially easy by providing a integrated environment for working with R script files, an active R session, the R session history, etc. Many of these tasks can be done with the click of a button in RStudio.

You can selectively save portions of your history to a script file using the History panel in RStudio.

Then you can go back and edit the file.

- Use meaningful names.
- Write reusable functions.

  Learning to write your own functions will greatly increase your efficiency. (Stay tuned for details.)
- Comment your code.

  It's amazing what you can forget. The comment character in R is `#`.

A.3 Working with Data

A.3.1 Data in R packages

Data sets in the `datasets` package or any other loaded package are available via the `data()` function. Usually, the use of `data()` is unnecessary, however, since R will search most loaded packages (they must have been created with the lazy-load option) for data sets without the explicit use of `data()`. The `data()` function can be used to restore data after it has been modified or to control which package is used when data sets with the same name appear in multiple packages.

A.3.2 Loading data from flat files

R can read data from a number of file formats. The two most useful formats are `.csv` (comma separated values) and white space delimited. Excel and most statistical packages can read and write data in these formats, so these formats make it easy to transfer data between different software. R provides `read.csv()` and `read.table()` to handle these two situations. They work nearly identically except for their default settings: `read.csv()` assumes that the first line of the file contains the variable names but `read.table()` assumes that the data begins on the first line with no names for the variables, and `read.table()` will ignore lines that begin with '`#`' but `read.csv()` will not.

The default behavior can be overridden for each function, and there are a number of options that make it possible to read other file formats, to omit a specified number of lines at the top of the file, etc. If you are making the file yourself, always include meaningful names in either file format.

It is also possible to read data from a file located on the Internet. Simply replace the file name with a URL. The data read below come from [Tuf01].

```
> # need header=TRUE because there is a header line.
> # could also use read.file() without header=TRUE
> traffic <-
    read.table("http://www.calvin.edu/~rpruim/fastR/trafficTufte.txt",
    header=TRUE)
> traffic
```

```
  year cn.deaths   ny   cn   ma   ri
1 1951       265 13.9 13.0 10.2  8.0
2 1952       230 13.8 10.8 10.0  8.5
3 1953       275 14.4 12.8 11.0  8.5
4 1954       240 13.0 10.8 10.5  7.5
5 1955       325 13.5 14.0 11.8 10.0
6 1956       280 13.4 12.1 11.0  8.2
7 1957       273 13.3 11.9 10.2  9.4
8 1958       248 13.0 10.1 11.8  8.6
9 1959       245 12.9 10.0 11.0  9.0
```

Notice the use of `<-` in the example above. This is the **assignment operator** in R. It can be used in either direction (`<-` or `->`). In the first line of the example above, the results of `read.table()` are stored in a variable called `traffic`. `traffic` is a **data frame**, R's preferred container for data. (More about data types in R as we go along.)

The `na.strings` argument can be used to specify codes for missing values. The following can be useful for SAS output, for example:

```
> read.csv('file.csv', na.strings=c('NA','','.','na')) -> someData
```

because SAS uses a period (`.`) to code missing data, but R by default reads that as string data, which forces the entire variable to be of character type instead of numeric.

For convenience the `mosaic` package provides `read.file()` which uses the file name to determine which of `read.csv()`, `read.table()`, and `load()` to use and sets the defaults to

- `header=TRUE`,
- `comment.char="#"`, and
- `na.strings=c('NA','','.','na')`

for `read.csv()` and `read.table()`.

```
> traffic <- read.file("http://www.calvin.edu/~rpruim/fastR/trafficTufte.txt")
```

A.3.3 Manually typing in data

If you need to enter a small data set by hand, the `scan()` function is quick and easy. Individual values are separated by white space or new lines. A blank line is used to signal the end of the data. By default, `scan()` is expecting decimal data (which it calls **double**, for double precision), but it is possible to tell `scan()` to expect something else, like **character** data (i.e., text). There are other options for data types, but numerical and text data will usually suffice for our purposes. See `?scan` for more information and examples.

```
myData1 <- scan()
15 18
12
21 23 50 15

myData1

myData2 <- scan(what="character")
"red" "red" "orange" "green" "blue" "blue" "red"

myData2
```

Be sure when using `scan()` that you remember to save your data somewhere. Otherwise you will have to type it again.

A.3.4 Creating data frames from vectors

The `scan()` function puts data into a **vector**, not a **data frame**. We can build a **data frame** for our data as follows.

```
> myDataFrame <- data.frame(color=myData2, number=myData1)
> myDataFrame
   color number
1    red     15
2    red     18
3 orange     12
4  green     21
5   blue     23
6   blue     50
7    red     15
```

A.3.5 Getting data from mySQL data bases

The `RMySQL` package allows direct access to data in MySQL data bases. This can be convenient when dealing with subsets of very large data sets. A great example of this is the 12 gigabytes of data from the Airline on-time performance dataset included in the 2009 Data Expo (`http://stat-computing.org/dataexpo/2009`). There is an online document describing this type of manipulation.

A.3.6 Generating data

The following code shows a number of ways to generate data systematically.

```
> x <- 5:20; x                  # all integers in a range
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> # structured sequences
> seq(0, 50, by=5)
 [1]  0  5 10 15 20 25 30 35 40 45 50
> seq(0, 50, length=7)
[1]  0.000  8.333 16.667 25.000 33.333 41.667 50.000
> rep(1:5, each=3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> rep(1:5, times=3)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> c(1:5, 10, 3:5)               # c() concatenates vectors
[1]  1  2  3  4  5 10  3  4  5
```

R can also sample from several different distributions.

```
> rnorm(10, mean=10, sd=2)  # random draws from normal distribution
 [1]  7.062  8.766 13.322  9.800  7.999 10.770  8.862 10.212  8.166  7.381
> x <- 5:20                 # all integers in a range
> sample(x, size=5)         # random sample of size 5 from x (no replacement)
```

```
[1] 18 19 16  8 12
```

Functions for sampling from other distributions include `rbinom()`, `rchisq()`, `rt()`, `rf()`, `rhyper()`, etc. See Section ?? for more information.

A.3.7 Saving Data

`write.table()` and `write.csv()` can be used to save data from R into delimited flat files.

```
> ddd <- data.frame(number=1:5, letter=letters[1:5])
> args(write.table)
function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
    eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"), fileEncoding = "")
NULL
> write.table(ddd, "ddd.txt")
> write.csv(ddd, "ddd.csv")
> # this system call should work on a Mac or Linux machine
> system("head -20 ddd.txt ddd.csv")
```

Data can also be saved in native R format. Saving data sets (and other R objects) using `save()` has some advantages over other file formats:

- Complete information about the objects is saved, including attributes.
- Data saved this way takes less space and loads much more quickly.
- Multiple objects can be saved to and loaded from a single file.

The downside is that these files are only readable in R.

```
> abc <- "abc"
> ddd <- data.frame(number=1:5, letter=letters[1:5])
> save(ddd, abc, file="ddd.rda")   # saves both objects in a single file
> load("ddd.rda")                  # loads them both
```

For more on importing and exporting data, especially from other formats, see the *R Data Import/Export* manual available on CRAN.

A.4 Primary R Data Structures

A.4.1 Modes and other attributes

In R, data is stored in objects. Each **object** has a *name*, *contents*, and also various *attributes*. Attributes are used to tell R something about the kind of data stored in an object and to store other auxiliary information. Two important attributes shared by all objects are **mode** and **length**.

```
> w <- 2.5; x <- c(1,2); y <- "foo"; z <- TRUE; abc <- letters[1:3]
> mode(w); length(w)
[1] "numeric"
[1] 1
> mode(x); length(x)
```

```
[1] "numeric"
[1] 2
> mode(y); length(y)
[1] "character"
[1] 1
> y[1]; y[2]             # not an error to ask for y[2]
[1] "foo"
[1] NA
> mode(z); length(z)
[1] "logical"
[1] 1
> abc
[1] "a" "b" "c"
> mode(abc); length(abc)
[1] "character"
[1] 3
> abc[3]
[1] "c"
```

Each of the objects in the example above is a **vector**, an ordered container of values that all have the same mode.[1] The `c()` function concatenates vectors (or lists). Notice that `w`, `y`, and `z` are vectors of length 1. Missing values are coded as `NA` (not available). Asking for an entry "off the end" of a vector returns `NA`. Assigning a value "off the end" of a vector results in the vector being lengthened so that the new value can be stored in the appropriate location.

There are important ways that R has been optimized to work with vectors since they correspond to variables (in the sense of statistics). For categorical data, a **factor** is a special type of vector that includes an additional attribute called *levels*. A factor can be ordered or unordered (which can affect how statistics are done and graphs are made) and its elements can have mode `numeric` or `character`.

A **list** is similar to a vector, but its elements may be of different modes (including `list`, `vector`, etc.). A **data frame** is a list of vectors (or factors), each of the same length, but not necessarily of the same mode. This is R's primary way of storing data sets. An **array** is a multi-dimensional table of values that all have the same mode. A **matrix** is a 2-dimensional array.

The access operators (`[ ]` for vectors, matrices, arrays, and data frames, and `[[ ]]` for lists) are actually *functions* in R. This has some important consequences:

- Accessing elements is slower than in a language like C/C++ where access is done by pointer arithmetic.
- These functions also have named arguments, so you can see code like the following

```
> xm <- matrix(1:16, nrow=4); xm
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

[1] There are other modes in addition to the ones shown here, including `complex` (for complex numbers), `function`, `list`, `call`, and `expression`.

```
> xm[5]
[1] 5
> xm[,2]                    # this is 1 dimensional (a vector)
[1] 5 6 7 8
> xm[,2, drop=FALSE]         # this is 2 dimensional (still a matrix)
     [,1]
[1,]    5
[2,]    6
[3,]    7
[4,]    8
```

Many objects have a **dim attribute** that stores the dimension of the object. You can change it to change the shape (or even the number of dimensions) of a vector, matrix, or array. You can see all of the non-intrinsic attributes (mode and length are intrinsic) using `attributes()`, and you can set attributes (including new ones you make up) using `attr()`. Some attributes, like dimension, have special functions for accessing or setting. The `dim()` function returns the dimensions of an object as a vector. Alternatively the number of rows and columns can be obtained using `nrow()` and `ncol()`.

```
> ddd <- data.frame(number=1:5, letter=letters[1:5])
> attributes(ddd)
$names
[1] "number" "letter"

$row.names
[1] 1 2 3 4 5

$class
[1] "data.frame"
> dim(ddd)
[1] 5 2
> nrow(ddd)
[1] 5
> ncol(ddd)
[1] 2
> names(ddd)
[1] "number" "letter"
> row.names(ddd)
[1] "1" "2" "3" "4" "5"
```

A.4.2 What `is` it?

R provides a number of functions for testing the mode or class of an object.

```
> mode(xm); class(xm)
[1] "numeric"
[1] "matrix"
> c(is.numeric(xm), is.character(xm), is.integer(xm), is.logical(xm))
[1]  TRUE FALSE  TRUE FALSE
> c(is.vector(xm), is.matrix(xm), is.array(xm))
[1] FALSE  TRUE  TRUE
```

A.4.3 Changing modes (coercion)

If R is expecting an object of a certain mode or class but gets something else, it will often try to **coerce** the object to meet its expectations. You can also coerce things manually using one of the many `as.???()` functions.

```
> apropos("^as\\.")[1:10]      # just a small sample
 [1] "as.array"               "as.array.default"       "as.call"
 [4] "as.category"            "as.character"           "as.character.condition"
 [7] "as.character.Date"      "as.character.default"   "as.character.error"
[10] "as.character.factor"
> xm
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> # convert numbers to strings (this drops attributes, including dimension)
> as.character(xm)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15" "16"
> # convert matrix to vector
> as.vector(xm)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
> as.logical(xm)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> alpha <- c("a", "1", "b", "0.5")
> mode(alpha)
[1] "character"
> as.numeric(alpha)      # can't do the coercion, so NAs are introduced
[1]  NA 1.0  NA 0.5
> as.integer(alpha)      # notice coercion of 0.5 to 0
[1] NA  1 NA  0
```

A.5 More About Vectors

Vectors are so important in R that they deserve some additional discussion. In Section A.3.6 we learned how to generate some simple vectors. Here we will learn about some of the operations and functions that can be applied to vectors.

A.5.1 Names and vectors

We can give each position in a vector a name. This can be very handy for certain uses of vectors.

```
> myvec <- 1:5; myvec
[1] 1 2 3 4 5
> names(myvec) <- c('one','two','three','four','five'); myvec
  one   two three  four  five
    1     2     3     4     5
```

Names can also be specified as a vector is created using `c()`.

```
> another <- c(mean=10, sd=2, "trimmed mean"=9.7); another
```

```
        mean           sd trimmed mean
        10.0          2.0          9.7
```

A.5.2 Vectorized functions

Many R functions and operations are "vectorized" and can be applied not just to an individual value but to an entire vector, in which case they are applied componentwise and return a vector of transformed values. Most traditional mathematics functions are available and work this way.

```
> x <- 1:5; y <- seq(10, 60, by=10); z <- rnorm(10); x; y
[1] 1 2 3 4 5
[1] 10 20 30 40 50 60
> y + 1
[1] 11 21 31 41 51 61
> x * 10
[1] 10 20 30 40 50
> x < 3
[1]  TRUE  TRUE FALSE FALSE FALSE
> x^2
[1]  1  4  9 16 25
> log(x); log(x, base=10)            # natural and base 10 logs
[1] 0.0000 0.6931 1.0986 1.3863 1.6094
[1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

Vectors can be combined into a matrix using `rbind()` or `cbind()`. This can facilitate side-by-side comparisons.

```
> # compare round() and signif() by binding rowwise into matrix
> rbind(round(z, digits=2), signif(z, digits=2))
     [,1] [,2] [,3] [,4]  [,5] [,6]  [,7] [,8]   [,9] [,10]
[1,] 0.82 1.23 1.77  1.2 -0.25 0.16 0.030 1.29 -0.010 -0.58
[2,] 0.82 1.20 1.80  1.2 -0.25 0.16 0.031 1.30 -0.011 -0.58
```

A.5.3 Functions that act on vectors as vectors

Other functions, including many statistical functions, are designed to work on the vector as a vector. Often these return a single value (technically a vector of length 1), but other return types are used as appropriate.

```
> x <- 1:10; z <- rnorm(100)
> mean(z); sd(z); var(z); median(z)  # basic statistical functions
    mean
-0.06371
   sd
1.005
 var
1.01
```

```
 median
0.03395
> range(z)                          # range returns a vector of length 2
[1] -3.256  2.012
> sum(x); prod(x)                       # sums and products
[1] 55
[1] 3628800
> z <- rnorm(5); z
[1] -2.5012 -1.3353  1.4802  0.1213  3.0001
> sort(z); rank(z); order(z)            # sort, rank, order
[1] -2.5012 -1.3353  0.1213  1.4802  3.0001
[1] 1 2 4 3 5
[1] 1 2 4 3 5
> rev(x)                                # reverse x
 [1] 10  9  8  7  6  5  4  3  2  1
> diff(x)                               # pairwise differences
[1] 1 1 1 1 1 1 1 1 1
> cumsum(x)                             # cumulative sum
 [1]  1  3  6 10 15 21 28 36 45 55
> cumprod(x)                            # cumulative product
 [1]       1       2       6      24     120     720    5040   40320  362880 3628800
> sum(x); prod(x)                       # sums and products
[1] 55
[1] 3628800
```

Whether a function is vectorized or treats a vector as a unit depends on its implementation. Usually, things are implemented the way you would expect. Occasionally you may discover a function that you wish were vectorized and is not. When writing your own functions, give some thought to whether they should be vectorized, and test them with vectors of length greater than 1 to make sure you get the intended behavior.

Some additional useful functions are included in Table A.1.

A.5.4 Recycling

When vectors operate on each other, the operation is done componentwise, recycling the shorter vector to match the length of the longer.

```
> x <- 1:5; y <- seq(10, 70, by=10)
> x + y
[1] 11 22 33 44 55 61 72
```

In fact, this is exactly how things like `x + 1` actually work. If `x` is a vector of length n, then `1` (a vector of length 1) is first recycled into a vector of length n; then the two vectors are added componentwise. Some vectorized functions that take multiple vectors as arguments will first use recycling to make them the same length.

Table A.1: Some useful R functions.

`cumsum()`
`cumprod()`
`cummin()`
`cummax()`
	Returns vector of cumulative sums, products, minima, or maxima.

`pmin(x,y,...)` `pmax(x,y,...)`	Returns vector of parallel minima or maxima where ith element is max or min of `x[i]`, `y[i]`, ...
`which(x)`	Returns a vector of indices of elements of `x` that are true. Typical use: `which(y > 5)` returns the indices where elements of `y` are larger than 5.
`any(x)`	Returns a `logical` indicating whether any elements of `x` are true. Typical use: `if ( any(y > 5) ) { ...}`.
`na.omit(x)`	Returns a vector with missing values removed.
`unique(x)`	Returns a vector with repeated values removed.
`table(x)`	Returns a table of counts of the number of occurrences of each value in `x`. The table is similar to a vector with names indicating the values, but it is not a vector.
`paste(x,y,...,` `sep=" ")`	Pastes `x` and `y` together componentwise (as strings) with `sep` between elements. Recycling applies.

A.5.5 Accessing elements of vectors

R allows for some very interesting and useful methods for accessing elements of a vector that combine the ideas above. First, recall that the `[ ]` operator is actually a function. Furthermore, it is vectorized.

```
> x <- seq(2, 20, by=2)
> x[1:5]; x[c(1, 4, 7)]
```

```
[1]  2  4  6  8 10
```

```
[1]  2  8 14
```

`[ ]` accepts `logicals` as arguments well. The boolean values (recycled, if necessary) are used to select or deselect elements of the vector.

```
> x <- seq(2, 20, by=2)
> x[c(TRUE, TRUE, FALSE)]      # skips every third element (recycling!)
```

```
[1]  2  4  8 10 14 16 20
```

```
> x[x > 10]                    # more typical use of boolean in selection
```

```
[1] 12 14 16 18 20
```

Negative indices are used to omit elements.

```
> x <- seq(2, 20, by=2)
> x[c(TRUE,TRUE,FALSE)]        # skips every third element (recycling!)
```

```
[1]  2  4  8 10 14 16 20
```

```
> x[x > 10]                    # more typical use of boolean in selection
```

```
[1] 12 14 16 18 20
```

Here are some more examples.

```
> notes <- toupper(letters[1:7]); a <- 1:5; b <- seq(10, 100, by=10)
> toupper(letters[5:10])
[1] "E" "F" "G" "H" "I" "J"
> paste(letters[1:5], 1:3, sep='-')
[1] "a-1" "b-2" "c-3" "d-1" "e-2"
> a+b
 [1]  11  22  33  44  55  61  72  83  94 105
> (a+b)[ a+b > 50]
[1]  55  61  72  83  94 105
> length((a+b)[a+b > 50])
[1] 6
> table(a+b > 50)
FALSE  TRUE
    4     6
```

A.6 Manipulating Data Frames

A.6.1 Adding new variables to a data frame

We can add additional variables to an existing data frame by simple assignment.

```
> summary(iris)
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
 Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa    :50
 1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
 Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
 Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
 3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
 Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
> iris$SLength <- cut(iris$Sepal.Length, 4:8)    # cut places data into bins
> summary(iris)
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
 Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1   setosa    :50
 1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3   versicolor:50
 Median :5.80   Median :3.00   Median :4.35   Median :1.3   virginica :50
 Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
 3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
 Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
  SLength
 (4,5]:32
 (5,6]:57
 (6,7]:49
 (7,8]:12
```

It is an error to add a vector of the wrong length.

The CPS data frame contains data from a Current Population Survey (current in 1985, that is). Two of the variables in this data frame are age and educ. We can estimate the number of years a worker has been in the workforce if we assume they have been in the workforce since completing their education and that their age at graduation is 6 more than the number of years of education obtained. We can this as a new variable in the data frame simply by assigning to it:

```
> CPS$workforce.years <- with(CPS, age - 6 - educ)
> favstats(CPS$workforce.years)
 min Q1 median Q3 max  mean    sd   n missing
  -4  8     15 26  55 17.81 12.39 534       0
```

In fact this is what was done for all but one of the cases to create the exper variable that is already in the CPS data.

```
> with(CPS, table(exper - workforce.years))
  0   4
533   1
```

A.6.2 Dropping variables

Since we already have educ, there is no reason to keep our new variable. Let's drop it. Notice the clever use of the minus sign.

```
> CPS1 <- subset(CPS, select = -workforce.years)
```

Any number of variables can be dropped or kept in this manner by supplying a vectors of variables names.

```
> CPS1 <- subset(CPS, select = -c(workforce.years,exper))
```

If we only want to work with the first few variables, we can discard the rest in a similar way. Columns can be specified by number as well as name (but this can be dangerous if you are wrong about where the columns are):

```
> CPSsmall <- subset(CPS, select=1:4)
> head(CPSsmall,2)
  wage educ race sex
1  9.0   10    W   M
2  5.5   12    W   M
```

A.6.3 Renaming variables

Both the column (variable) names and the row names of a data frames can be changed by simple assignment using names() or row.names().

```
> ddd                       # small data frame we defined earlier
  number letter
1      1      a
2      2      b
3      3      c
4      4      d
5      5      e
> row.names(ddd) <- c("Abe","Betty","Claire","Don","Ethel")
> ddd                       # row.names affects how a data.frame prints
       number letter
Abe         1      a
Betty       2      b
Claire      3      c
Don         4      d
Ethel       5      e
```

More interestingly, it is possible to reset just individual names with the following syntax.

```
> row.names(ddd)[2] <- "Bette"         # misspelled a name, let's fix it
> row.names(ddd)
[1] "Abe"    "Bette"  "Claire" "Don"    "Ethel"
```

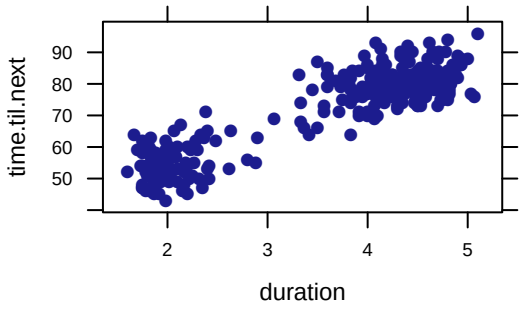
The `faithful` data set (in the `datasets` package, which is always available) has very unfortunate names.

```
> names(faithful)
[1] "eruptions" "waiting"
```

The measurements are the duration of an euption and the time until the subsequent eruption, so let's give it some better names.

```
> names(faithful) <- c('duration', 'time.til.next')
> head(faithful, 3)
  duration time.til.next
1    3.600           79
2    1.800           54
3    3.333           74
```

```
> xyplot(time.til.next ~ duration, faithful)
```



If the variable containing a data frame is modified or used to store a different object, the original data from the package can be recovered using `data()`.

```
> data(faithful)
> head(faithful, 3)
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
```

If we want to rename a variable, we can do this using `names()`. For example, perhaps we want to rename `educ` (the second column) to `education`.

```
> names(CPS)[2] <- 'education'
> CPS[1,1:4]
  wage education race sex
1    9        10    W   M
```

See Section A.5 for information that will make it clearer what is going on here.

If we don't know the column number (or generally to make our code clearer), a few more keystrokes produces

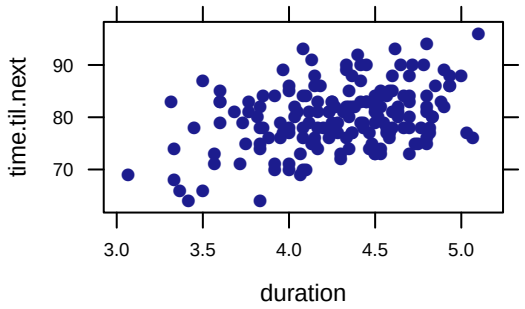
```
> names(CPS)[names(CPS) == 'education'] <- 'educ'
> CPS[1,1:4]
```

```
  wage educ race sex
1    9   10    W   M
```

A.6.4 Creating subsets

We can also use `subset()` to reduce the size of a data set by selecting only certain rows.

```
> data(faithful)
> names(faithful) <- c('duration', 'time.til.next')
> # any logical can be used to create subsets
> faithfulLong <- subset(faithful, duration > 3)
> xyplot( time.til.next ~ duration, faithfulLong )
```



Of course, if all we want to do is produce a graph, there is no reason to create a new data frame. The plot above could also be made with

```
> xyplot( time.til.next ~ duration, faithful, subset=duration > 3 )
```

A.6.5 Merging datasets

The `fusion1` data frame in the `fastR` package contains genotype information for a SNP (single nucleotide polymorphism) in the gene *TCF7L2*. The `pheno` data frame contains phenotypes (including type 2 diabetes case/control status) for an intersecting set of individuals. We can merge these together to explore the association between genotypes and phenotypes using `merge()`.

```
> require(fastR)
> head(fusion1,3)
```

```
     id     marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose
1  9735 RS12255372        1       3       3       GG     0     0     2     0
2 10158 RS12255372        1       3       3       GG     0     0     2     0
3  9380 RS12255372        1       3       4       GT     0     0     1     1
```

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

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

```
> # merge fusion1 and pheno keeping only id's that are in both
> fusion1m <- merge(fusion1, pheno, by.x='id', by.y='id', all.x=FALSE, all.y=FALSE)
> head(fusion1m, 3)
    id     marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose     t2d
1 1002 RS12255372        1       3       3       GG     0     0     2     0    case
2 1009 RS12255372        1       3       3       GG     0     0     2     0    case
3 1012 RS12255372        1       3       3       GG     0     0     2     0 control
    bmi sex   age smoker chol waist weight height    whr sbp dbp
1 32.86   F 70.76 former 4.57 112.0   85.6  161.4 0.9868 135  77
2 27.39   F 53.92  never 7.32  93.5   77.4  168.1 0.9397 158  88
3 30.47   M 53.86 former 5.02 104.0   94.6  176.2 0.9327 143  89
```

In this case, since the values are the same for each data frame, we could collapse `by.x` and `by.y` to `by` and collapse `all.x` and `all.y` to `all`. The first of these specifies which column(s) to use to identify matching cases. The second indicates whether cases in one data frame that do not appear in the other should be kept (`TRUE`) or dropped (filling in `NA` as needed) or dropped from the merged data frame.

Now we are ready to begin our analysis.

```
> xtabs(~t2d + genotype + marker, fusion1m)

, , marker = RS12255372

         genotype
t2d        GG  GT  TT
  case    737 375  48
  control 835 309  27
```

A.6.6 Slicing and dicing

`reshape()` provides a flexible way to change the arrangement of data. It was designed for converting between long and wide versions of time series data and its arguments are named with that in mind.

A common situation is when we want to convert from a wide form to a long form because of a change in perspective about what a unit of observation is. For example, in the `traffic` data frame, each row is a year, and data for multiple states are provided.

```
> traffic
  year cn.deaths   ny   cn   ma   ri
1 1951       265 13.9 13.0 10.2  8.0
2 1952       230 13.8 10.8 10.0  8.5
3 1953       275 14.4 12.8 11.0  8.5
4 1954       240 13.0 10.8 10.5  7.5
5 1955       325 13.5 14.0 11.8 10.0
6 1956       280 13.4 12.1 11.0  8.2
7 1957       273 13.3 11.9 10.2  9.4
8 1958       248 13.0 10.1 11.8  8.6
9 1959       245 12.9 10.0 11.0  9.0
```

We can reformat this so that each row contains a measurement for a single state in one year.

```
> longTraffic <-
        reshape(traffic[,-2], idvar="year", ids=row.names(traffic),
        times=names(traffic)[3:6], timevar="state",
        varying=list(names(traffic)[3:6]),
        v.names="deathRate",
        direction="long")
> head(longTraffic)
```

```
        year state deathRate
1951.ny 1951    ny      13.9
1952.ny 1952    ny      13.8
1953.ny 1953    ny      14.4
1954.ny 1954    ny      13.0
1955.ny 1955    ny      13.5
1956.ny 1956    ny      13.4
```

And now we can reformat the other way, this time having all data for a given state form a row in the data frame.

```
> stateTraffic <- reshape(longTraffic, direction='wide',
                          v.names="deathRate", idvar="state", timevar="year")
> stateTraffic
        state deathRate.1951 deathRate.1952 deathRate.1953 deathRate.1954
1951.ny    ny           13.9           13.8           14.4           13.0
1951.cn    cn           13.0           10.8           12.8           10.8
1951.ma    ma           10.2           10.0           11.0           10.5
1951.ri    ri            8.0            8.5            8.5            7.5
        deathRate.1955 deathRate.1956 deathRate.1957 deathRate.1958 deathRate.1959
1951.ny           13.5           13.4           13.3           13.0           12.9
1951.cn           14.0           12.1           11.9           10.1           10.0
1951.ma           11.8           11.0           10.2           11.8           11.0
1951.ri           10.0            8.2            9.4            8.6            9.0
```

In simpler cases, `stack()` or `unstack()` may suffice. `Hmisc` also provides `reShape()` as an alternative to `reshape()`.

A.7 Functions in R

Functions in R have several components:

- a **name** (like `histogram`)[2]
- an ordered list of named **arguments** that serve as inputs to the function

  These are matched first by name and then by order to the values supplied by the call to the function. This is why we don't always include the argument name in our function calls. On the other hand, the availability of names means that we don't have to remember the order in which arguments are listed.

  Arguments often have **default values** which are used if no value is supplied in the function call.
- a **return value**

  This is the output of the function. It can be assigned to a variable using the assignment operator (`=`, `<-`, or `->`).
- **side effects**

  A function may do other things (like make a graph or set some preferences) that are not necessarily part of the return value.

When you read the help pages for an R function, you will see that they are organized in sections related to these components. The list of arguments appears in the **Usage** section along with any

[2] Actually, it is possible to define functions without naming them; and for short functions that are only needed once, this can actually be useful.

default values. Details about how the arguments are used appear in the **Arguments** section. The return value is listed in the **Value** section. Any side effects are typically mentioned in the **Details** section.

Now let's try writing our own function. Suppose you frequently wanted to compute the mean, median, and standard deviation of a distribution. You could make a function to do all three to save some typing. Let's name our function `favstats()`. `favstats()` will have one argument, which we are assuming will be a vector of numeric values.[3] Here is how we could define it:

```
> favstats <- function(x) {
     mean(x)
     median(x)
     sd(x)
 }
> favstats((1:20)^2)
```

```
   sd
127.9
```

The first line says that we are defining a function called `favstats()` with one argument, named `x`. The lines surrounded by curly braces give the code to be executed when the function is called. So our function computes the mean, then the median, then the standard deviation of its argument.

But as you see, this doesn't do exactly what we wanted. So what's going on? The value returned by the last line of a function is (by default) returned by the function to its calling environment, where it is (by default) printed to the screen so you can see it. In our case, we computed the mean, median, and standard deviation, but only the standard deviation is being returned by the function and hence displayed. So this function is just an inefficient version of `sd()`. That isn't really what we wanted.

We can use `print()` to print out things along the way if we like.

```
> favstats <- function(x) {
     print(mean(x))
     print(median(x))
     print(sd(x))
 }
> favstats((1:20)^2)
```

```
 mean
143.5
median
 110.5
   sd
127.9
```

Alternatively, we could use a combination of `cat()` and `paste()`, which would give us more control over how the output is displayed.

```
> altfavstats <- function(x) {
     cat(paste("  mean:", format(mean(x),4),"\n"))
     cat(paste(" edian:", format(median(x),4),"\n"))
     cat(paste("    sd:", format(sd(x),4),"\n"))
 }
> altfavstats((1:20)^2)
```

```
  mean: 143.5
 edian: 110.5
    sd: 127.9
```

Either of these methods will allow us to see all three values, but if we try to store them ...

[3] There are ways to check the **class** of an argument to see if it is a data frame, a vector, numeric, etc. A really robust function should check to make sure that the values supplied to the arguments are of appropriate types.

```
> temp <- favstats((1:20)^2)

 mean
143.5
median
 110.5
   sd
127.9

> temp

   sd
127.9
```

A function in R can only have one return value, and by default it is the value of the last line in the function. In the preceding example we only get the standard deviation since that is the value we calculated last.

We would really like the function to return all three summary statistics. Our solution will be to store all three in a vector and return the vector.[4]

```
> favstats <- function(x) {
         c(mean(x), median(x), sd(x))
 }
> favstats((1:20)^2)

  mean median     sd
 143.5  110.5  127.9
```

Now the only problem is that we have to remember which number is which. We can fix this by giving names to the slots in our vector. While we're at it, let's add a few more favorites to the list. We'll also add an explicit `return()`.

```
> favstats <- function(x) {
     result <- c(min(x), max(x), mean(x), median(x), sd(x))
     names(result) <- c("min","max","mean","median","sd")
     return(result)
 }
> favstats((1:20)^2)

   min    max   mean median     sd
   1.0  400.0  143.5  110.5  127.9

> summary(Sepal.Length~Species, data=iris, fun=favstats)

Sepal.Length    N=150

+-------+----------+---+---+---+-----+------+------+
|       |          |N  |min|max|mean |median|sd    |
+-------+----------+---+---+---+-----+------+------+
|Species|setosa    | 50|4.3|5.8|5.006|5.0   |0.3525|
|       |versicolor| 50|4.9|7.0|5.936|5.9   |0.5162|
|       |virginica | 50|4.9|7.9|6.588|6.5   |0.6359|
+-------+----------+---+---+---+-----+------+------+
|Overall|          |150|4.3|7.9|5.843|5.8   |0.8281|
+-------+----------+---+---+---+-----+------+------+

> aggregate(Sepal.Length~Species, data=iris, FUN=favstats)

     Species Sepal.Length.min Sepal.Length.max Sepal.Length.mean Sepal.Length.median
1     setosa           4.3000           5.8000            5.0060              5.0000
2 versicolor           4.9000           7.0000            5.9360              5.9000
3  virginica           4.9000           7.9000            6.5880              6.5000
  Sepal.Length.sd
```

[4] If the values had not all been of the same mode, we could have used a list instead.

```
1          0.3525
2          0.5162
3          0.6359
```

Notice the use of `::` to select the `favstats()` function from the `mosaic` package rather than the one we just defined.

Notice how nicely this works with `aggregate()` and with the `summary()` function from the `Hmisc` package. You can, of course, define your own favorite function to use with `summary()`. The `favstats()` function in the `mosaic` package includes the quartiles, mean, standard, deviation, sample size and number of missing observations.

```
> mosaic::favstats(rnorm(100))
    min      Q1  median     Q3   max    mean     sd   n missing
 -2.371 -0.6595 0.02846 0.7654 2.249 0.01931 0.9744 100       0
```

Exercises

A.1 Using `faithful` data frame, make a scatter plot of eruption duration times vs. the time since the previous eruption.

A.2 The `fusion2` data set in the `fastR` package contains genotypes for another SNP. Merge `fusion1`, `fusion2`, and `pheno` into a single data frame.

Note that `fusion1` and `fusion2` have the same columns.

```
> names(fusion1)
 [1] "id"       "marker"   "markerID" "allele1"  "allele2"  "genotype" "Adose"
 [8] "Cdose"    "Gdose"    "Tdose"
> names(fusion2)
 [1] "id"       "marker"   "markerID" "allele1"  "allele2"  "genotype" "Adose"
 [8] "Cdose"    "Gdose"    "Tdose"
```

You may want to use the `suffixes` argument to `merge()` or rename the variables after you are done merging to make the resulting data frame easier to navigate.

Tidy up your data frame by dropping any columns that are redundant or that you just don't want to have in your final data frame.

Bibliography

[Fis25] R. A. Fisher, *Statistical methods for research workers*, Oliver & Boyd, 1925.

[Fis70] ———, *Statistical methods for research workers*, 14th ed., Oliver & Boyd, 1970.

[GN02] A Gelman and D Nolan, *Teaching statistics: a bag of tricks*, Oxford University Press, 2002.

[Gou10] R. Gould, *Statistics and the modern student*, International Statistical Review **78** (2010), no. 2, 297–315.

[Kap09] D. Kaplan, *Statistical modeling: A fresh approach*, CreateSpace.com, 2009.

[MH08] B. D. Mccullough and David A. Heiser, *On the accuracy of statistical procedures in microsoft excel 2007*, Computational Statistics & Data Analysis **52** (2008), 4570–4578, available online at `www.pages.drexel.edu/~bdm25/excel2007.pdf`.

[NT10] D. Nolan and D. Temple Lang, *Computing in the statistics curriculum*, The American Statistician **64** (2010), no. 2, 97–107.

[Sal01] D. Salsburg, *The lady tasting tea: How statistics revolutionized science in the twentieth century*, W.H. Freeman, New York, 2001.

[Tuf01] E. R. Tufte, *The visual display of quantitative information*, 2nd ed., Graphics Press, Cheshire, CT, 2001.