Class 2: Statistical computing using R (programming)

You must read this chapter while sitting at the computer with an R window open. You will learn by typing in code from this chapter. You can learn further by playing with the code. Computing is central to modern statistics at all levels, from basic to advanced. If you already know how to program, great. If not, the examples in this book should give you what you'll need to get started.

We do our computing in the open-source package R, a command-based statistical software environment that you can download and operate on your own computer.

Downloading and setting up R

Start at the home of R, `http://www.r-project.org/`. Download R and install it on your computer as follows:

1. On the R webpage, click on the "download R" link. This will take you to a list of mirror sites. Choose any of these. Now click on the link under "Download and Install R" at the top of the page for your operating system (Linux, Mac, or Windows):

   - If Windows, click on "base" and then click on the Download link at the top of the next page. Open the file and run it. Click through until you reach the window for "Startup options." At this point, click on "Yes (customized startup" and then, on the next screen, click on "SDI (separate windows)," Then click through from there.
   - If Mac, double click on the installer package downloaded. Follow the directions in the installer (default settings in the installer should be fine). You may need administrator privileges to install R properly. If you're the only user on your machine, don't worry: you are the administrator.
   - If linux, use the installer that is compatible with your linux distribution.

If your installation of R above has been successful, there will be a R icon on your desktop. Click on the icon to start an R session. You should get a copyright message and a prompt: "`>`". To check that R is working, type `2 + 5` and hit Enter. You should get `[1] 7`. From now on, when we say "type" `_____`, we mean type (or copy-and-paste) `_____` into R and hit Enter.

The next step is to install `diff`: a package in R with special routines that will be used in this course. In the R window, type `install.packages ("diff")` and hit Enter. A window will open asking you to choose a CRAN mirror. Any mirror is fine: double-click and the package will be installed.

Type `library ("diff")` to load in the `diff` package. Every time you restart R, you will need to do `library ("diff")`. But you do not need to *install* the package again. Once is enough.

Your working directory

Choose a *working directory* on your computer where you will do your R work. Suppose your working directory is `c:/myfiles/stat/`. Then you should put all your data files in this directory, and all the files and graphs you save in R will appear here too.

Now type `getwd()`. This shows your current R working directory. Change your working directory by typing `setwd ("c:/myfiles/stat/")`, or whatever you would like to use.

Try a few things, typing these one line at a time and looking at the results on the console:

```
getwd ()
1/3
sqrt(2)
curve (x^2 + 5, from=-2, to=2)
```

These will return, respectively, your working directory (for example, `c:/myfiles/stat/`), `0.3333333`, `1.414214`, and a new graphics window plotting the curve $y = x^2 + 5$.

When working in R, we recommend that you have three windows open: the R console, the R graphics window (which opens automatically when you make a graph, as above), and a text editor open to a file with a name such as `script.R`. Text editors include Notepad on Windows, TextEdit on Mac OS X, or Emacs on linux. Type your R commands into script.R and copy-and-paste them into the R console. The advantage of this approach (compared to simply typing into R directly) is that your commands are all in a file you can save. So your work will be reproducible.

Finally, quit your R session by typing `q()`. If you will be working in the same working directory every time, you should set your default working directory. In Windows, this is done by right-clicking on the R icon on the desktop, going into "Properties" and setting the "Start in" field to the directory of choice. On a Mac, open up the "Preferences" in R; within Preferences, there should be a button to "Change working directory" which allows you to choose the default working directory.

Calling functions and getting help

Open R, type `library ("diff")`, and play around, using the assignment function ("`<-`"). To start, type the following lines into your script.R file and copy-and-paste them into the R window:

```
a <- 3
print (a)
b <- 10
print (b)
a + b
a*b
exp(a)
10^a
log(b)
log10(b)
a^b
round (3.435, 0)
round (3.435, 1)
round (3.435, 2)
```

R is based on *functions*, which include mathematical operations (`exp()`, `log()`, `sqrt()`, and so forth) and lots of other routines (`print()`, `round()`, ...).

The function `c()` combines things together into a vector. For example, type `c(4,10,-1,2.4)` in the R console or copy-and-paste the following:

```
x <- c(4,10,-1,2.4)
print (x)
```

The function `seq()` creates an equally-spaced sequence of numbers; for example, `seq(4,54,10)` returns the sequence from 4, 14, 24, 34, 44, 54. The `seq()` function works with non-integers as well: try `seq(0,1,0.1)` or `seq(2,-5,-0.4)`. For integers, `a:b` is shorthand for `seq(a,b,1)` or `seq(b,a,-1)` if `b` $<$ `a`. Let's try a few more commands:

```
c (1, 3, 5)
1:5
c(1:5, 1, 3, 5)
c(1:5, 10:20)
seq (2, 10, 2)
seq (1, 9, 2)
```

You can get help on any function using "`?`" in R. For example, type `?seq`. This should open an internet browser window with a help file for `seq()`. R help files typically have more information than you'll know what to do with, but if you scroll to the bottom of the page you'll find some examples that you can cut and paste into your console.

Whenever you are trying out a new function, we recommend using "`?`" to view the help file and running the examples at the bottom to see what happens.

Comments

You can write comments for yourself or anyone else trying to follow your programming. "`#`" tells R to ignore the rest of the line. You can intersperse lines of comments (starting with `#`) with lines of code and still be able to copy-and-paste multiple lines at a time without confusing R. It will be helpful to comment your `script.R` or other files to help you remember what steps you have taken.

Sampling and random numbers

Here's how to get a random number, uniformly distributed between 0 and 100:

```
runif (1, 0, 100)
```

And now 50 more random numbers:

```
runif (50, 0, 100)
```

Suppose we want to pick one of three colors with equal probability:

```
color <- c ("blue", "red", "green")
sample (color, 1)
```

Suppose we want to sample with unequal probabilities:

```
color <- c ("blue", "red", "green")
p <- c (0.5, 0.3, 0.2)
sample (color, 1, prob=p)
```

Or we can do it all in one line, which is more compact but less readable:

```
sample (c("blue","red","green"), 1, prob=c(0.5,0.3,0.2))
```

Data types

Numeric data. In R, numbers are stored as *numeric* data. This includes many of the examples above as well as special constants such as *pi*.

Big and small numbers. R recognizes scientific notation. A million can be typed in as `1000000` or `1e6`, but not `1,000,000`. (R is particular about certain things. Capitalization matters, "`,`" doesn't belong in numbers, and spaces usually aren't important.) Scientific notation also works for small numbers: `1e-6` is `0.000001` and `4.3e-6` is `0.0000043`.

Infinity. Type (or copy-and-paste) these into R, one line at a time, and see what happens:

```
1/0
-1/0
exp(1000)
exp(-1000)
1/Inf
Inf + Inf
-Inf - Inf
0/0
Inf - Inf
```

Those last two operations returns `NaN` (Not a Number); type `?Inf` for more on the topic. In general we try to avoid working with infinity but it is convenient to have Inf for those times when we accidentally divide by 0 or perform some other illegal mathematical operation.

Missing data. In R, `NA` is a special keyword that represents missing data. For more information, type `?NA`. Try these commands in R:

```
NA
2 + NA
NA - NA
NA / NA
NA * NA
c (NA, NA, NA)
c (1, NA, 3)
10 * c(1, NA, 3)
NA / 0
NA + Inf
is.na (NA)
is.na (c(1, NA, 3))
```

The `is.na()` function tests whether or not the argument is `NA`. The last line operates on each element of the vector and returns a vector with three values that indicate whether the corresponding input is NA.

Character strings. Let's sample a random color and a random number and put them together:

```
color <- sample (c("blue","red","green"), 1, prob=c(0.5,0.3,0.2))
number <- runif (1, 0, 100)
paste (color, number)
```

Here's something prettier:

```
paste (color, round (number,0))
```

TRUE, FALSE, and ifelse

Try typing these:

```
2 + 3 == 4
2 + 3 == 5
1 < 2
2 < 1
```

In R, the expressions `==, <, >` are *comparisons* and return a logical value, `TRUE` or `FALSE` as appropriate. Other comparisons include `<=` (less than or equal), `>=` (greater than or equal), and `!=` (not equal).

Comparisons can be used in combination with the `ifelse()` function. The first argument takes a logical statement, the second argument is an expression to be evaluated if the statement is true, and the third argument is evaluated if the statement is false. Suppose we want to pick a random number between 0 and 100 and then choose the color red if the number is below 30 or blue otherwise:

```
number <- runif (1, 0, 100)
color <- ifelse (number<30, "red", "blue")
```

Loops

A key aspect of computer programming is *looping*—that is, setting up a series of commands to be performed over and over. Start by trying out the simplest possible loop:

```
for (i in 1:10){
  print ("hello")
}
```

Or:

```
for (i in 1:10){
  print (i)
}
```

Or:

```
for (i in 1:10){
  print (paste ("hello", i))
}
```

The curly braces define what is repeated in the loop.

Here's a loop of random colors:

```
for (i in 1:10){
  number <- runif (1, 0, 100)
  color <- ifelse (number<30, "red", "blue")
  print (color)
}
```

Working with vectors

In R, a vector is a list of items. These items can include numerics, characters, or logicals. A single value is actually represented as a vector with one element. Here are some vectors:

- (1, 2, 3, 4, 5)
- (3, 4, 1, 1, 1)

- ("A", "B", "C")

Here's the R code to create these:

```
x <- 1:5
y <- c(3, 4, 1, 1, 1)
z <- c("A", "B", "C")
```

And here's a random vector of 5 random numbers between 0 and 100:

```
u <- runif (5, 0, 100)
```

Mathematical operations on vectors are done component-wise. Take a look:

```
x
y
x + y
1000*x + u
```

There are scalar operations on vectors:

```
1 + x
2 * x
x / 3
x^4
```

We can summarize vectors in various ways, including the sum and the average (called the "mean" in statistics jargon):

```
sum (x)
mean (x)
```

We can also compute weighted averages if we know the weights. We illustrate with a vector of 3 elements:

```
x <- c (100, 200, 600)
w1 <- c (1/3, 1/3, 1/3)
w2 <- c (0.5, 0.2, 0.3)
```

In the above code, the vector of weights `w1` has the effect of counting each of the three items equally; vector `w2` counts the first item more. Here are the weighted averages:

```
sum (w1*x)
sum (w2*x)
```

Or suppose we want to weight in proportion to population:

```
N <- c (310e6, 112e6, 34e6)
sum(N*x)/sum(N)
```

Or, equivalently:

```
N <- c (310e6, 112e6, 34e6)
w <- N/sum(N)
sum(w*x)
```

The `cumsum()` function does the cumulative sum. Try this:

```
a <- c(1, 1, 1, 1, 1)
cumsum (a)
a <- c(2, 4, 6, 8, 10)
cumsum (a)
```

Subscripting Vectors can be indexed by using brackets, "`[ ]`". Within the brackets we can put in a vector of elements we are interested in either as a vector of numbers or a logical vector. When using a vector of numbers, the vector can be arbitrary length, but when indexing using a logical vector, the length of the vector must match the length of the vector you are indexing. Try these:

```
a <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J")
a[1]
a[2]
a[4:6]
a[c(1,3,5)]
a[c(8,1:3,2)]
a[c(FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE)]
```

As we have seen in some of the previous examples, we can perform mathematical operations on vectors. These vectors have to be the same length, however. If the vectors are not the same length, we can subset the vectors so they are compatible. Try these:

```
x <- c (1, 1, 1, 2, 2)
y <- c (2, 4, 6)
x[1:3] + y
x[3:5] * y
y[3]^x[4]
x + y
```

The last line runs but produces a warning. These warnings should not be ignored since it isn't guaranteed that R would carry out the operation as you intended.

Writing your own functions

You can write you own function. Most of the functions we will be writing will take in one or more vectors and return a vector. Below is an example of a simple function that triples the value provided:

```
triple <- function (x) {
  return (3*x)
}
```

To call this function, type `triple(c(1,2,3))`. This function has one argument, `x`. The body of the function is within the curly braces and the arguments of the function are available for use within the braces. In our example function, we multiply `x` by 3 and we return it back to the user. If we wanted to have more than one argument, we could replace the header of the function, `function(x)`, with `function(x,y,z)`.

Optimization

Finding the peak of a parabola. Figure 2.1 shows the parabola $y = 15 + 10x - 2x^2$. As you can see, the peak is at $x = 2.5$. How can we find this solution systematically (and without using calculus, which is not required for this course)? Finding the maximum of a function is called an *optimization* problem. Here's how we do it in R.

1. Graph the function, in this case, `curve(15+10*x-2*x^2,from=___,to=___)`, where ___ are numbers. Play around with the "from" and "to" arguments until the maximum appears.
2. Write it as a function:

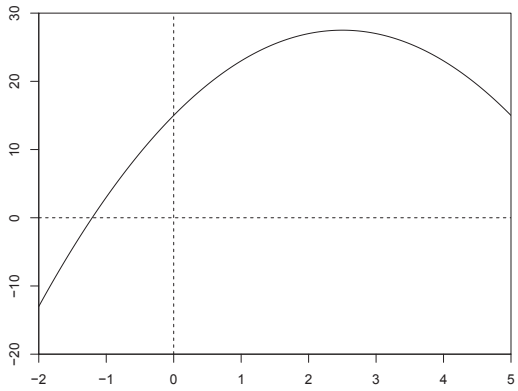

Figure 2.1 *The parabola* $y = 15 + 10x - 2x^2$. *The maximum is at* $x = 2.5$.

```
parabola <- function (x) {
  return (15 + 10*x - 2*x^2)
}
```

3. Call `optimize()`. We can find the maximum of a function in R through the `optimize()` function which takes the function to optimize as the first argument, the interval to optimize over as the second argument, and an optional argument indicating whether or not you are searching for the function's maximum. For the example above, we can make this call:

   ```
   optimize (parabola, interval=c(-10, 10), maximum=TRUE)
   ```

   This returns two values, x and $f(x)$. The value labeled `maximum` is the x at which the function is optimized and the `objective` is its corresponding $f(x)$ value.

4. Check the solution on the graph to see that it makes sense.

Restaurant pricing Suppose you own a small restaurant and are trying to decide how much to charge for dinner. For simplicity, suppose that dinner will have a single price, and that your marginal cost per dinner is \$11. From a marketing survey, you have estimated that, if you charge \$$s$ for dinner, the average number of customers per night you will get is $5000/s^2$. To get an idea of what this function looks like, you can make a graph showing the estimated number of customers per night, at prices ranging from \$10 to \$100:

```
curve (5000/x^2, from=10, to=100)
```

Make this graph yourself right now. Given the above information, how much should you charge, if your goal is to maximize profit?

If dinner costs \$$X$, your net profit is # customers times net profit per customer; that is, $(5000/s^2) \cdot (s - 11)$. To get a sense of where this is maximized, we can first make a graph:

```
net.profit <- function (s){
  (5000/s^2)*(s-11)
}
```

```
curve (net.profit (x), from=10, to=100, xlab="Price of dinner",
  ylab="Net profit per night")
```

From a visual inspection of this curve, the peak appears to be around $X = 20$. There are two ways we can more precisely determine where net profit is maximized.

First, the brute-force approach. The optimum is clearly somewhere between 10 and 100, so let's just compute the net profit at a grid of points in this range:

```
x <- seq (10, 100, .1)
y <- net.profit (x)
```

The maximum value of net profit is then simply `max(y)`, which equals 113.6, and we can find the value of x where profit is maximized:

```
x[y==max(y)]
```

It turns out to be 22. (The value 22 is `x[121]`. Here, we subscript the vector `x` with a logical vector `y==max(y)`, which is a vector of 900 `FALSE`'s and one `TRUE` at the maximum.)

We can also use the `optimize()` function:

```
optimize (net.profit, interval=c(10,100), maximum=TRUE)
```

The argument `interval` gives the range over which the optimization is computed, and we set `maximum=TRUE` to maximize (rather than minimize) the net profit.

Class 3: Working with data in R (reading data and making graphs)

Reading data

Let's read some data into R. The file heads.csv has data from a coin flipping experiment done in a previous class. Each student flipped a coin 10 times and we have a count of the number of students who saw exactly 0, 1, 2, ..., 10 heads. The file is located at the website `http://applied.stat.columbia.edu/diff/data/`. Start by going to this location, downloading the file, and saving it as heads.csv in your working directory (for example, `c:/myfiles/stat/`).

Now read the file into R:

```
heads <- read.csv ("heads.csv")
```

R can also read the file directly from the web rather than saving it to your computer first. For example:

```
heads <- read.csv ("http://applied.stat.columbia.edu/diff/data/heads.csv")
```

Typing the name of any object in R displays the object itself. So type `heads` and look at what comes out.

Now for a longer file: download `nycData.csv` from `http://applied.stat.columbia.edu/diff/data/`, save it in your working directory, and read it in:

```
nycData <- read.csv ("nycData.csv")
```

This data is a subset of the census data from New York City zip codes. If you now type `nycData` from the console, the file will scroll off the screen: it's too big! Instead, let's just look at the first 5 rows:

```
nycData[1:5, ]
```

We'll talk in a bit about what to do with these data, but first we'll consider some other issues involving data input.

What if you have tabular data separated by spaces and tabs, rather than columns? For example, mile.txt from `http://applied.stat.columbia.edu/diff/`. You just need to call the `read.table()` function:

```
mile <- read.table ("mile.txt", header=TRUE)
mile[1:5, ]
```

The `header=TRUE` argument is appropriate here because the first line of the file mile.txt is a "header," that is, a list of column names. For a "naked" data file `hello.txt` with no header, we would simply call `read.table("hello.txt")` with no header argument.

Writing data

You can save data into a file using `write` instead of `read`. For example, to write the R object `heads` into a comma-separated file `output1.csv`, we would type `write.csv(heads,"output1.csv")`. To write it into a space-separated file output2.txt, it's just `write.table(heads,"output2.txt")`.

Examining data

At this point, we should have two variables in R, `heads` and `nycData`. (We can see what variables are in our session by typing `ls()`.)

Data frames, vectors, and subscripting. Most of the functions used to read data return a data structure called a `data.frame`. You can see this by typing `class(heads)`. Each column of a `data.frame` is a vector (see page 16 for more details). We can access the first column of `heads` by typing `heads[,1]`. `data.frame`s are indexed using two vectors inside "`[`" and "`]`"; the two vectors are separated by a comma. The first vector indicates which rows you are interested in and the second vector indicates what columns you are interested in. For example, `heads[6,1]` shows the number of heads observed and `heads[6,2]` shows the number of students that observed that number of heads. Leaving it blank is shorthand for all. Try:

```
heads[6,]
heads[1:3,]
heads[,1]
heads[,1:2]
heads[,]
```

To find the number of columns in a `data.frame`, use the function `length()`. To find the number of rows, use `nrow()`. We can also find the names of the columns by using `names()`. Try these:

```
length (heads)
nrow (heads)
names (heads)
length (nycData)
nrow (nycData)
names (nycData)
```

We can also index a column within a `data.frame` by using `$`. The call to `names(nycData)` showed the names of the 7 columns in the data set. We can access the zip codes by typing `nycData$zipcode` or the population of each of the zip codes by typing `nycData$pop.total`. As we mentioned before, these columns are vectors. We can index these vectors like any other vectors. Try:

```
nycData$zipcode[1:5]
nycData$pop.total[1:5]
```

(We often store individual columns of a larger data set as their own variable for ease of use. For example, we might type `zipcode <- nycData$zipcode` so zipcode is readily accessible.)

Other useful commands. Sometimes it is useful to create blank data structures before filling them in with data.

- For a vector, type `rep(NA,10)` to create a blank vector of length 10.
- For a matrix, type `array(NA,c(10,2))` to create a blank matrix of 10 rows and 2 columns.

We index a `matrix` the same way we index a `data.frame` using "`[`" and "`]`".

Let's use the `heads` data and get the frequency of each of the observations. Try:

```
frequency <- heads$count / sum (heads$count)
```

We can attach frequency to our data by using `cbind()`:

```
heads <- cbind (heads, frequency)
```

This `cbind()` binds columns of data together. If we wanted to add another observation to the dataset (binding in a row), we would use `rbind()`.

Cleaning a data file before reading it into R

Often times, data will need to be cleaned before it can be read into R. We illustrate some of the techniques that you might use with an example. Suppose we are interested in the violent crime rate versus the property crime rate by state. This data is available from the Census (Table 304) at `http://www.census.gov/compendia/statab/cats/law_enforcement_courts_prisons.html`. Two data formats are provided, Excel and PDF, but both are not directly readable by R. We will need to take a few steps to clean the data before reading it in R.

- Open the original data. We will open it as an Excel file. (You can do this from the PDF, but for this example, it's a little easier using the Excel file. If you don't have Excel, there are open source options for spreadsheets that should be just as effective).
- Create a new file. This is the final file we will save. Save this file as `crime.csv` in your working directory.
- Create a header row. In `crime.csv`, create a header row with the appropriate labels. For this example, we want 3 columns: `state`, `violent crime`, `property crime`. Type these into the first three boxes of the first row.
- Copy the data.
  - `state`. Highlight the states from the original data. We don't want the whole column since it contains merged cells, special formatting, and its own header row. Copy from the original data file and paste into `crime.csv` under the `state` heading. Don't edit these values yet. We want to copy all three columns first. (We omitted the first row, United States, and just copied over the states.)
  - `violent crime`. Highlight and copy the column for total violent crimes in 2008. Paste these under the `violent crime` heading in `crime.csv`.
  - `property crime`. Highlight and copy the column for total property crimes in 2008. Paste these under the `property crime` heading in `crime.csv`. At this point, we can close the original data file if these three columns are all that we are concerned with.
- Remove any blank rows. If you've followed along so far, you'll find row 10 is blank. Delete the whole row. (Don't just delete one of the values or the values won't line up.)
- Clean the data.
  - Check the values. The lines for some of the states (D.C., Illinois, and Minnesota) aren't quite right. Fix them.
  - Check for missing values. The violent crime numbers have two `(NA)`'s. Delete these values; we want these cells to be blank. Make sure not to shift the column in any way while deleting these values.
  - Check the numbers. Things are looking pretty good at this point, but we will have one more problem with this data. A lot of the numbers have commas in them. Remember, numbers in R doesn't have commas. We need to remove the commas from the numbers. There are two ways to do this. We can either edit each number by hand or let Excel help us. If editing by hand, make sure to double check your changes. If you're using Excel, we want the `violent crime` and `property crime` columns to be formatted as numbers without using a thousand separator. To do this, right-click in the highlighted region and click on "Format Cells." Under the "Number" tab, click on the "Number" category.

```
100022  31659123     121222113121432 22  2 3 411179797979797 1  4 503100100...
100081  486 2122     111222141122221222  2 1 997979797979797 1  4  01 25 25...
100091   1371123     1232122111113111314 1 0                          30100...
100101  15684222     133122121113232 22  1 0                           10 40...
100111  25371122     122222111111421222  2 2 4 6979797979797 1  2 853 30 95...
100202   2389013     1111221412   22 314 2 0                        100100...
100281   7884021     2232132422   42 22  2 0                         80 60...
100351  15684221     233223242112212 32  2 0                         75100...
100571  88341221     243233321113452 12  2 0                         90 98...
100641  15684223     122112211113432 22  2 0                        100 75...
```

Figure 3.1 *First ten lines of the file* `06666-0001-Data.txt`*, which has data from the Work, Family, and Well-Being survey.*

Check and uncheck the box for "Use 1000 Separator (,)." (If you find yourself in a situation where negative numbers are denoted with parentheses or are colored red instead of a "-" sign, you can have Excel format those numbers for you.)

- Save the data. Make sure the data is saved as a comma separated file. You can now read this file into R by typing `read.csv("crime.csv")`. (A csv file is a plain text file where the columns are separated by commas and rows are separated by new lines. If you open up `crime.csv` in a text editor, you should be able to see why having commas in numbers could be hard for R to figure out.)

We have included `crime.csv` in the `http://applied.stat.columbia.edu/diff/data/` folder as a comparison. If you have followed the steps above, you should have created a file identical to the one provided.

Reading in survey data, one question at a time

For an example later in the course, we study the heights, weights, and incomes of a random sample of Americans. The data come from the Work, Family, and Well-Being survey conducted by Catherine Ross in 1990. We downloaded the data file, `06666-0001-Data.txt`, and the codebook `06666-0001-Codebook.txt` from the Inter-university Consortium for Political and Social Research.[1]

Figure 3.1 shows the first ten lines of the data, and Figure 3.2 shows the relevant portion of the codebook. Our first step is to read in the data, for each question pulling out the appropriate columns from the file:

```
height.feet <- read.columns ("wfw90.dat", 144)
height.inches <- read.columns ("wfw90.dat", 145:146)
weight <- read.columns ("wfw90.dat", 147:149)
income.exact <- read.columns ("wfw90.dat", 203:208)
income.approx <- read.columns ("wfw90.dat", 209:210)
sex <- read.columns ("wfw90.dat", 219)
```

The data did not come in a convenient comma-separated or tab-separated format, so we used the function `read.columns` to read the coded responses, one question at a time.

Cleaning data within R

We now must put the data together in a useful form, doing the following for each variable of interest:

[1] Information on the survey is at `http://dx.doi.org/10.3886/ICPSR06666`, and can be downloaded if you create an account with the ICPSR.

```
    HEIGHT     144-146   F3.0     Q.46 HEIGHT IN INCHES
    WEIGHT     147-149   F3.0     Q.47 WEIGHT
. . .
    EARN1      203-208   F6.0     Q.61  PERSONAL INCOME - EXACT AMOUNT
    EARN2      209-210   F2.0     Q.61  PERSONAL INCOME - APPROXIMATION
    SEX        219       F1.0     Q.63  GENDER OF RESPONDENT
. . .
    HEIGHT
    46. What is your height without shoes on?
        __________ ft.  __________in.
    WEIGHT
    47. What is your weight without clothing?
        __________ lbs.
. . .
    61a. During 1989, what was your personal income from your own wages,
    salary, or other sources, before taxes?
    EARN1
    $ __________--> (SKIP TO Q-62a)
    DON'T KNOW . . . 98
    REFUSED  . . . . 99
```

Figure 3.2 *Selected rows of the file* `wfwcodebook.txt`*, which first identifies the columns in the data corresponding to each survey question and then gives the question wordings.*

1. Look at the data
2. Identify errors or missing data
3. Transform or combine raw data into summaries of interest.

We start with height, typing: `table (height.feet, height.inches)`. Here is the result:

```
height.feet   0   1   2   3   4   5   6   7   8   9  10  11  98  99
          4   0   0   0   0   0   0   0   0   0   1   3  17   0   0
          5  66  56 144 173 250 155 247 127 174 105 145  90   0   0
          6 129  59  46  20   8   5   1   0   0   0   1   0   0   0
          7   0   0   0   0   0   0   0   1   0   0   0   0   0   0
          9   0   0   0   0   0   0   0   0   0   0   0   0   2   6
```

Most of the data look fine, but there are some people with 9 feet and 98 or 99 inches (missing data codes) and one person who is 7 feet 7 inches tall (probably a data error). We recode these problem cases as missing:

```
height.inches[height.inches>11] <- NA
height.feet[height.feet>=7] <- NA
```

And then we define a combined height variable:

```
height <- 12*height.feet + height.inches
```

We do the same thing for sex:

```
table (sex)
```

which simply yields:

```
sex
   1    2
 749 1282
```

No problems. But we prefer to have a variable coded as 0=male, 1=female, so we define a new variable, `female`:

```
female <- sex - 1
```

Next, we type `table (weight)` and get the following:

```
weight
 80  85  87  89  90  92  93  95  96  98  99 100 102 103 104 105 106 107 108 110
  1   1   1   1   1   1   1   2   2   3   1  12   5   4   3  16   1   5   7  46
111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130
  4  15   5   5  42   5   4  21   4  72   4  14  20  11  61  11   3  25   8 106
131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
  4  16   9   5  85   9  10  15   4  94   1  12   2   4  74   2   7   8   5 121
151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
  2   4   8   7  49   3   6  14   1  88   1   8   4   9  65   2   2   8   2  81
171 172 173 174 175 176 178 180 181 182 183 184 185 186 187 188 189 190 192 193
  2  10   4   2  58   5   4  78   3   4   2   4  62   1   5   1   3  46   1   3
194 195 196 197 198 199 200 201 202 203 205 206 207 208 209 210 211 212 214 215
  4  26   3   2   3   2  57   1   2   2  11   2   3   2   3  36   1   2   3  10
217 218 219 220 221 222 223 225 228 230 231 235 237 240 241 244 248 250 255 256
  1   1   1  21   2   1   1  13   2  17   1   3   1  13   1   1   1  10   3   1
260 265 268 270 275 280 295 312 342 998 999
  2   3   1   3   1   3   1   1   1   6  36
```

Everything looks fine until the end. 998 and 999 must be missing data, which we duly code as such:

```
weight[weight>500] <- NA
```

Coding the income responses is more complicated, and we will relegate the details to a footnote.[2] We create a combined income variable as follows:

```
income.approx[income.approx>=90] <- NA
income.approx[income.approx==1] <- 150
income <- ifelse (is.na(income.exact), 1000*income.approx, income.exact)
```

The new `income` variable still has 237 missing values (out of 2031 respondents in total) and is imperfect in various ways, but we have to make some choices when working with real data.

Looking at the data

If you stare at the table of responses to the weight question you can see more. People usually round their weight to the nearest 5 or 10 pounds, and so we see a lot of weights reported as 100, 105, 110, and so forth, but not so many in between. Beyond this, people appear to like round numbers: 57 people report weights of 200 pounds, compared to only 46 and 36 people reporting 190 and 210, respectively.

Similarly, if we go back to reported heights we see some evidence that the reported numbers do not correspond exactly to physical heights: 129 people report heights

[2] The variable `income.exact` contains exact responses for income (in *dollars per year*) for those who answered the question. Typing `table(is.na(income.exact))` reveals that 1380 people answered the question (that is, `is.na(income.exact)` is `FALSE` for these respondents), and 651 did not answer (`is.na` was `TRUE`). These nonrespondents were asked the second, discrete income question, `income.approx`, which gives incomes in round numbers (in *thousands of dollars per year*) for people who were willing to answer in this way. A careful look at the codebook reveals that an `income.approx` code of 1 corresponds to people who did not supply an exact income value but did say it was more than $100,000. We code these people as having incomes of 150,000. (This is approximately the average income of the over-100,000 group from the exact income data, as calculated by `mean(income.exact[income.exact>100000],na.rm=TRUE)`, which yields the value 155,000.) The `income.approx` codes also appears to have several values indicating ambiguity or missingness, which we code as NA.

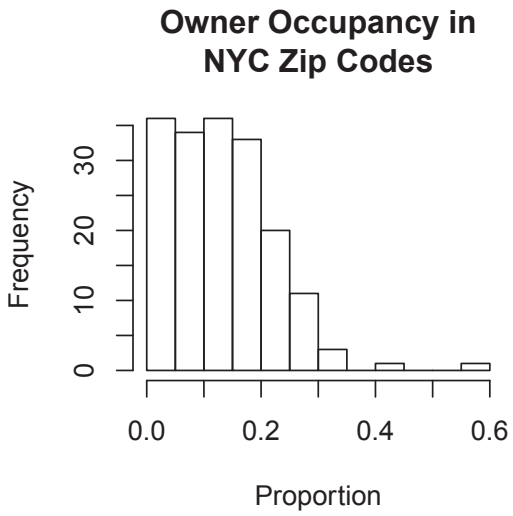

Figure 3.3 *Histogram of owner occupancy in New York City zip codes. Most of the zip codes have less than 20% owner occupancy.*

of exactly 6 feet, compared to 90 people at 5 feet 11 inches and 59 people at 6 feet 1 inch. Who are these people? Let's look at the breakdown of height and sex:

```
table (female, height)
```

Here's the result:

```
       height
female  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74
     0   0   0   0   3   2   4   3  14  13  47  43  81  77 119  82 125  56  44
     1   1   3  17  63  54 140 170 236 142 200  84  93  28  26   8   4   3   2
       height
female  75  76  77  78  82
     0  19   8   5   1   1
     1   1   0   0   0   0
```

The extra 6-footers (72 inches tall) are just about all men. But both men and women appear to have an excess of people who report being exactly 5 feet or exactly 5 feet 6 inches.

Graphing data

Histograms. A histogram shows the distribution of data. For example, Figure 3.3 shows the histogram of owner occupancy in New York zip codes. To generate this graph, type:

```
hist (nycData$owner.occupied / nycData$pop.total,
  main="Proportion of Owner Occupancy in NYC Zip Codes")
```

For continuous data such as proportions we use `hist()`. For discrete data we use `discrete.histogram()`, which has spaces between the bars indicating that the bar represents the probability of seeing of that result. For example, in Figure 3.3, the first bar represents the number of zip codes that have owner occupancy between 0% and 5%. The first bar in Figure 3.4 shows the frequency students that saw exactly 2 heads in 10 flips. To generate Figure 3.4, type:

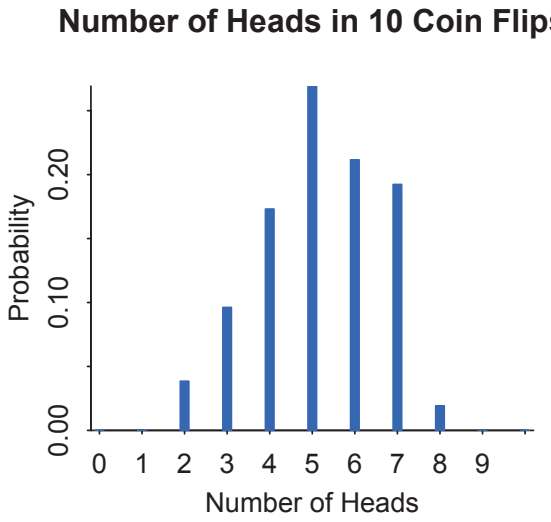

Figure 3.4 *This discrete histogram shows the distribution of number of heads seen in 10 flips by a class of 52 students.*

```
discrete.histogram (heads$x, heads$count,
  main="Number of Heads in 10 Coin Flips")
```

Scatterplots. Scatterplots show relationships between two variables. The `mile` data contains the world record times in the mile run since 1900 as three columns, `year`, `min`, and `sec`. Figure 1.7 plots the world record time (in minutes) against time. We can create scatterplots in R using `plot`:

```
year <- mile$year
record <- mile$min + mile$sec/60
plot (year, record, main="World record times in the mile run")
```

Lines and curves. We can add the straight line shown in Figure 1.7 to the scatterplot by using `curve()`:

```
curve (4.33 - 0.00655*(x-1900), add=TRUE)
```

The first argument is the equation of the line as a function of x. The second argument, `add=TRUE`, tells R to draw the line onto the existing scatterplot. We can do this many times with different equations of a line to put multiple lines on a single graph. `curve()` can draw more than straight lines. It can draw anything that is a function of x.

If you are only interested in drawing a straight line, we can achieve the same result with `abline(a=16.77,b=0.00655)`. We can draw a horizontal line by typing `abline(h=4.0)` (a horizontal line can not be drawn using `curve()`).

Making pretty graphs

When making graphs in R, we often want to provide additional settings to make the graphs prettier. Some of the more common adjustments we make are listed below.

`plot`. These are additional arguments that can be provided to `plot()`. (A full description can be found by typing `?plot`.) Let's start with an example and make it prettier one step at a time. Type:

```
income <- nycData$median.household.income / 1000
ownership <- nycData$owner.occupied /
   (nycData$owner.occupied + nycData$renter.occupied)
plot (income, ownership)
```

- `xlim` and `ylim`. These optional parameters control the x and y limits on the graph, respectively. Both `xlim` and `ylim` expect a vector of two elements, the lower limit and the higher limit. Type this:

  ```
  plot (income, ownership, xlim=c(0,120), ylim=c(0,1))
  ```

- `xaxs` and tt yaxs. These parameters control how wide R graphs the x and y beyond the data. We often use `xaxs="i"` and `yaxs="i"` ("i" for internal); this makes the graphing area the limit of the data. Type:

  ```
  plot (income, ownership, xaxs="i", yaxs="i")
  ```

- `main`, `xlab`, and `ylab`. These label the graph. `main` is used for the title of the graph. Type:

  ```
  plot (income, ownership,
    main="Owner Occupancy by Zip Code in NYC",
    xlab="Median Income Level ($1000)", ylab="Fraction Owner Occupied")
  ```

- `bty`. This parameter controls the box drawn around the graph. We can change the box to have an open top and open right side by using `bty="l"`. Type:

  ```
  plot (income, ownership, bty="l")
  ```

  We can pull all of this together by putting all the parameters into the `plot()` call at once:

  ```
  plot (income, ownership,
    xlim=c(0,120), ylim=c(0,1),
    xaxs="i", yaxs="i",
    main="Owner Occupancy by Zip Code in NYC",
    xlab="Median Income Level ($1000)", ylab="Fraction Owner Occupied",
    bty="l")
  ```

Additional functions. There are other functions we use to help make pretty graphs. Here are some of them:

- Plotting multiple graphs on a page. We can do this by making a call to `par()`:

  ```
  par(mfrow=c(rows,cols))
  ```

  Here, `rows` is the number of rows and `cols` is the number of columns to graph on a single page. For example, to plot two graphs side by side, you would use `par(mfrow=c(1,2))`. Then you would call `plot()` with the appropriate parameters to create your first graph, then you would call `plot()` again to create your second graph.

- Labeling the title and axes with more control. If you want finer control of the labeling of the graph, you can use the `title()` function.

- Axis tick marks and labels. To change the tick marks along the axis and their labels, we use `axis()`. The first argument to `axis()` is 1 for the bottom axis and 2 for the left axis. The argument `at`, supplied as a vector, indicates where the tick marks should be drawn. The argument `labels`, also supplied as a vector, tells R how to label the tick marks.

We will provide more examples of usage through the book.

Saving graphs. R provides many methods for saving graphs. The simplest is to go the R graphics window (the one with your graph), right-click (or, on a Mac, ...), and choose the menu item to copy. The graph is now stored in the operating system's clipboard and we can paste it into a Word document, for example.

We can also save a graph directly into a `pdf` or `png` file. (There are many other output formats that are supported. The steps are essentially the same.) For a `pdf` file, follow these steps:

1. Open a `pdf` file. Use the function `pdf()` to create the file we will save the graph as. We can provide the filename as the first argument. (If no filename is given, R will put the graph in `Rplots.pdf`.) No additional graphics window will appear at this point. Every graph command will be put into the `pdf` file.
2. Create a graph. You can try this out with any of the graphs we have created starting on page 27.
3. Close the `pdf` file. You must close the file by typing `dev.off()`.

For a `png` file, use `png()` instead of `pdf()`.