An introduction to \texttt{R}

(By James Bernhard)

Getting started with \texttt{R} can be pretty intimidating, especially for those who are not used to entering commands at a command prompt. However, keeping in mind what \texttt{R} is a good tool for as well as what it is \textit{not} a good tool for can be very helpful.

The two main things that \texttt{R} is particularly good for are \textit{producing statistical graphics} and \textit{performing statistical computations}. While \texttt{R} can be and frequently is used for other tasks, there are often other much better tools for such other tasks.

Perhaps more important than what \texttt{R} is good for are two things that \texttt{R} is not:

1. \texttt{R} is decidedly not a \textit{spreadsheet}. While it has some basic spreadsheet functionality built into it, and while there have been several attempts to increase this functionality through packages, there are many programs (such as Microsoft Excel) whose spreadsheet functionality is far greater than that of \texttt{R}. Almost any attempt to treat \texttt{R} as a spreadsheet will come up woefully short. There are much better tools available for displaying data tables and editing individual or small numbers of entries by hand. Use those tools instead of \texttt{R} for such tasks.
2. **R** is not a *word processor*. The standard output from **R** is not pretty. It generally does not contain any fancy formatting and can almost invariably be laid out in a fashion much more conducive to communication of analysis and results. While it does have some tools to facilitate formatting of output, **R** was not built for such tasks and there are far better tools available. For those who use **LaTeX** as a word processor, the package **Sweave** represents a wonderful integration of **R** with a word processor. For those who use other word processing programs, cutting and pasting (or entering manually) output from **R** and then reformatting it in the word processor is probably the easiest route for formatting **R** output for analyses and write-ups.\(^1\)

Trying to use **R** for tasks to which it is not well-suited can often be inefficient and quite frustrating. Instead, if we use **R** for the purposes for which it was intended, we will not be disappointed.

In this chapter, we discuss the basics of getting around in **R**, including some simple things to do at the command prompt and some ways to load data into **R**. We should keep in mind, however, that there are usually many different ways to accomplish any given task in **R**. In this guide, we will usually give only one such way. Other sources may describe different ways to accomplish the same task, and you are encouraged to explore and find other such ways yourself as well.

### 0.1 The absolute necessities

When we first enter the **R** programming environment, we see some brief welcome messages and are given a command prompt “>” followed by a cursor. It may be difficult to know what to do from here, which is where we begin our discussion.

**Notational conventions**

In this guide, we use a typewriter font within a paragraph to indicate text to be entered at the **R** prompt and text that is output from **R**. (This includes names of **R** packages, objects, and variables, terms which we will

---

\(^1\)A version of **Sweave** for Open Document Format is also available for those wanting to explore it.
discuss later.) Also, within a paragraph, we use the convention that an *italicized typewriter* font indicates something that needs to be substituted into the expression, not typed literally. For example, the logarithm function in R is \( \log(number, \text{ base}=base) \). This means that to compute \( \log_{10} 32 \), we should enter \( \log(32, \text{ base}=10) \) into R at the command prompt.

For longer demonstrations of R code, we will use an italicized typewriter display set off from paragraphs to show R input and non-italicized typewriter display for resulting output, such as:

```R
> \log(32, \text{ base }= 10)

[1] 1.50515
```

In this example, what is being shown is that if we enter \( \log(32, \text{ base}=10) \) at the command prompt in R, the output will be \([1] \ 1.50515\). Note in particular that we do not type the “>” that appears here; it is merely shown to indicate that what follows is to be typed at the command prompt.

Also, even if a function wraps around to the next line in this guide, it is usually preferable to enter it on a single command line in R, simply ignoring the line breaks (which are there only for typesetting considerations). However, we can split any command that is not syntactically complete (perhaps it has a left parenthesis that hasn’t been matched with a right parenthesis yet, or some such thing) over multiple lines in R just by hitting Enter. We are then be given a “+” instead of a “>” at the beginning of the next line to indicate that what we are typing is a continuation of the previous line. We can do this as many times as we like, continuing a single command over multiple lines, as long as each new line comes at a point where the command couldn’t possibly be complete yet.

### Quitting and stopping R

The first thing to know about R is how to exit it once we have entered it. There are various ways to do this, but one that is common to all platforms of R is to enter `q()` at the command prompt, the `q` being short for “quit”. We do need to include the parentheses, since `q()` is (like almost everything else in R) a function, although it has no arguments. This function and its purpose are actually mentioned in the welcome messages as R starts up.
Section 0.1 The absolute necessities

After we have entered `q()`., we are asked whether or not we would like to save the current workspace. It doesn’t really matter if we answer “yes”, but at this point it would probably be preferable simply to answer “no” instead.

Another good thing to know about R is how to stop a computation or command if R has frozen or is taking too long. While this doesn’t happen very often, one way that it can happen is if `Enter` is pressed in the middle of a command being entered, in such a way that completing the command is difficult or impossible.

In any case, it is useful to know that to stop an R process or command during its execution, we press the `Esc` key. There are times when this doesn’t stop the process and more drastic and less desirable measures must be applied (halting R through the computer’s operating system, unplugging the computer, etc.), but in general pressing the `Esc` key is a good method of stopping a R process that is running.

Getting help in R

There are several sources for help built into R. Sometimes they require some getting used to in order to use them effectively, and sometimes they aren’t as helpful as one would like, but with a little practice, they can often be put to good use. We discuss them in the following order:

- `help("function name")`, which gives a help page for the function specified.
- `apropos("search string")`, which gives a listing of functions that contain the specified search string in their names.
- `help.search("search string")`, which gives a listing of functions that contain the specified search string in their names or descriptions.

If we happen to know the name of the function or command that we are interested in but would like to learn more about its exact syntax, its arguments or parameters, or how it is used, the `help("function name")` command can be used. For example, suppose we would like to know

^2This will merely cause R to create a file that we will probably want to delete later, to remove clutter.
which base the \texttt{log(number)} function in R uses if we don’t specify one explicitly. For this, we call up a help page on this function with

\begin{verbatim}
> help("log")
\end{verbatim}

Right at the top of the help page, we see that “\texttt{\textquote{log}} computes logarithms, by default natural logarithms”, so the default base is \textit{e}.

The help pages produced by the \texttt{help("function name")} command vary in how useful they are. Such help pages are written by the authors of the various functions, and some authors are more careful about their documentation than others. Most of the standard functions in R are fairly well documented though, even if it sometimes takes a little bit of practice to get used to the format of help pages in order to be able to extract the desired information from them.

In we do not actually know the name of the function that we are looking for in R but have a good guess at what it might be, we can use the \texttt{apropos("search string")} function. This function will produce a list of functions that contain the specified search string. For example, if we aren’t sure of the exact name of the function that computes base 2 logarithms in R but suspect that it contains at least the string “\texttt{log}” in its name, we can use:

\begin{verbatim}
> apropos("log")
\end{verbatim}

\begin{verbatim}
[1] ".\_\_C\_logical" "\_\_C\_logLik"
[3] ".\_\_T\_Logic:base" "as.data.frame.logical"
[5] "as.logical" "as.logical.factor"
[7] "clogit" "dlogis"
[9] "is.logical" "log"
[11] "log10" "log1p"
[13] "log2" "logb"
[15] "Logic" "logical"
[17] "loginPath" "logLik"
[19] "loglin" "Lognorm2"
[21] "logrank" "plogis"
[23] "print.logLik" "qlogis"
[25] "rlogis" "SSlogis"
[27] "tkdialog" "winDialog"
[29] "winDialogString"
\end{verbatim}
Section 0.1 The absolute necessities

From this, we can probably guess (with a little guessing and checking of help pages using help("function name") as needed) that \( \log_2 \) is just such a function, as help("log2") verifies.

If we don’t even have a guess as to what part of the name of the function is but can specify something about its purpose or description, we may be able to locate the function we seek with help.search("search string"). This will produce a list of functions that contain the given search string in their name or description.

For example, if we would like a function that is used in fitting linear models, we might try something like

\[
\textbf{help.search("linear model")}
\]

This produces a long list of functions, given in the form

\[
\text{package name::function name.}
\]

For the present, we ignore the package name and just use the function name. A little searching through the list in this case reveals that there is a function (in the stats package) called \texttt{lm} whose brief description is “Fitting linear models”, which is probably what we want. As we see though, the lists produced by help.search("search string") are often quite long, so we have to have a pretty good idea of what we are looking for in order to be able to spot it in the list.

One other option for getting R help, particularly if we don’t understand an error message, is to try using an internet search engine (such as Google) to search for a cut and pasted version of the exact error, or at least a large part of it. This doesn’t work as well with really basic error messages, but with errors resulting from more complicated and advanced commands in R, this is a good method. Help can often be found this way, and many times it will even be found on the official R website itself. Failing that, there are R newsgroups that can be asked about a particular question, but those tend to be more useful for advanced R users, so be sure to read all of the posting guidelines before posting to any of them.

Navigational shortcuts in R

Now that we have begun entering commands into R, a couple of keyboard shortcuts are good to know. First of all, to scroll back (and forth) through
prior commands that we have entered, we can use the up and down arrows on the keyboard. This often saves a large number of keystrokes, since many times we may want to execute almost the same command that we just entered. In this case, using the up arrow at the prompt will bring that command to the screen, and then we can edit it as usual from the R prompt. Note that the cursor does not need to be at the end of the line when we hit Enter; the whole line is submitted no matter what position the cursor occupies when Enter is pressed.

Also, on many computers, pressing and holding any of the arrows (up, down, left, or right) is the same as hitting that key repeatedly. There will be a brief delay as we hold the key down before the first repeat of the keypress starts up, and after that the keypresses will repeat themselves fairly rapidly as long as we hold the key down.

Two other useful keyboard shortcuts are Control-a and Control-e, which position the cursor at the beginning and end (respectively) of the line being entered. So for example, if we press the up arrow to recall the previous command, the cursor will be positioned at the end of the line. If we want to edit something near the beginning of the line, we can use Control-a to put the cursor at the beginning of the line and then use the right and left arrows to position the cursor where we would like it. This is usually faster than pressing the left arrow repeatedly where the cursor starts at the end of the line.

**Extending R**

At some point, we will want to take advantage of the multitudinous packages that R has at its disposal. For example, suppose we would like to use a function that is part of the “Grammar of Graphics” package ggplot2. We can’t use this function until we have done two things: installed the package and loaded the package into R.

To install the package, we select Install package(s) from the Packages pull-down menu in R. This will probably prompt us to select a mirror site from which to download the packages we select. While it isn’t particularly crucial, it is generally nicer and faster to download a package from a mirror site near us, so we select one that is fairly close to our location. We are then asked to select a package (or packages). We scroll down until we reach ggplot2 and select it (by double-clicking on it, for example). The package then downloads and installs into our local copy of R.
There are many packages in R, and it would be a tremendous waste of time and computer memory to load every previously installed package into R every time R is started up. Consequently, only a small core set of packages is loaded into R automatically at startup. Any other previously installed packages that we would like to use need to be loaded in with the `library("package name")` function. For example,

```r
> library("ggplot2")
```

This command would load the previously installed package ggplot2 into memory for the current R session. (It would need to be reloaded — but not reinstalled — when R is restarted.) Then any functions, data sets, etc. that are part of the ggplot2 package can be used for the remainder of the R session, until we quit R next.

In summary, new packages need to be installed only once, which puts them on the local computer. They can then be loaded with the `library("package name")` function, and this needs to be done once during each R session in which they are used.

We might not have need for installing packages immediately, but if we use R much, we will soon want to take advantage of some of the many packages that authors around the world have written and made available. These packages greatly extend the power and the usefulness of R.

### 0.2 Vectors and vectorized operations

At its most basic level, R can function as a standard calculator. We should take advantage of this as need be. Often students can be seen using handheld calculators while sitting in front of the computer with R started up. There is no need for this, and R will usually offer much greater calculator functionality (certainly in the realms of statistics, but also often otherwise) as long as we know how to use its calculator features, which is not difficult to do. The main thing that R is not particularly good at is giving answers as exact real numbers (such as fractions or square roots), a feature that isn’t of much use in applied statistics. The output of R is generally a real number rounded to a certain number of places. We should keep in mind though that R stores many more decimal places than it ordinarily displays. We will see some important ramifications of this later.
Basic calculator functions

All of the basic arithmetic functions in R work as usual, with answers being stored and displayed in decimal form:

\[
\begin{align*}
> 2 + 3 & \quad \text{[1] 5} \\
> 1 - 4.7 & \quad \text{[1] -3.7} \\
> 2 * 3 & \quad \text{[1] 6} \\
> 2/3 & \quad \text{[1] 0.6666667}
\end{align*}
\]

R uses the standard order of operations. For those not familiar with this order, if we would like to perform several arithmetic operations in a single command line, it is probably easiest just to use parentheses to indicate the order of operations, such as:

\[
\begin{align*}
> 2 + (3 * 4) & \quad \text{[1] 14} \\
> (2 + 3) * 4 & \quad \text{[1] 20}
\end{align*}
\]

R also has all of the other standard functions that a scientific calculator has, including:

- \texttt{sqrt(number)} for square roots
- \texttt{log(number)} for natural logarithms (denoted by "\text{ln}" in some texts, but not in R)
• \( \log(\text{number}, \ \text{base=base}) \) for logarithms with a given base

• \( \exp(\text{power}) \) for raising the number \( e = 2.71828 \ldots \) to the specified power, giving \( e^{\text{power}} \); this is the inverse function of the natural logarithm

• \( \pi \) to denote the number \( \pi = 3.14159 \ldots \)

For example, we have:

\[
\begin{align*}
> \sqrt{10} \\
&= 3.162278 \\
> \log(10) \\
&= 2.302585 \\
> \log(100, \ \text{base = 10}) \\
&= 2 \\
> \log(8, \ \text{base = 2}) \\
&= 3 \\
> \exp(1) \\
&= 2.718282 \\
> \exp(3) \\
&= 20.08554 \\
> \pi \\
&= 3.141593
\end{align*}
\]
We now begin to discuss how to store data in R. The types of data most frequently encountered in R are numbers, strings, and logical values. Numbers are fairly self-explanatory, and strings in R are sequences of characters enclosed in quotation marks,\(^3\) such as "fish", "a2", and "147". Note that even though 147 is a number, "147" is a string and is treated differently by R. Logical values are either TRUE or FALSE.\(^4\) Note that TRUE and FALSE are not the same as "TRUE" and "FALSE", and will not be interpreted the same way by R. The first two are logical values, while the next two are strings.

Data is typically entered into R in the form of vectors, which are typically ordered sets of one of the three main types of data: numbers, strings, or logical values.\(^5\) Vectors can also store other types of data, as we will discuss later. Note though that a vector cannot contain more than one type of data. A vector consisting of one of these three types is either all numbers, all strings, or all logical values. Vectors can be entered at the command prompt with the \texttt{c(element1, element2, \ldots)} function (short for “combine”), as in:

\begin{verbatim}
> c(3, 1, 4.2)
[1] 3.0 1.0 4.2

> c("fish", "sky", "tree", "squirrel")
[1] "fish"  "sky"  "tree"  "squirrel"

> c(FALSE, TRUE, TRUE, TRUE)
[1] FALSE TRUE TRUE TRUE
\end{verbatim}

The output here shows how vectors are typically displayed in R. The mysterious \([1]\) that appears at the beginning of each line is actually an indication that the first entry on that line is the first element of the vector. The first entry on a line beginning with \([7]\) would be the seventh element of

\(^3\)If we want to work with a string that actually contains " as one of its characters, we can delimit the string with ’ instead, as in the string ’The sign says, "Hello"’.

\(^4\)Or \texttt{NULL} or \texttt{NA}, but we will ignore such complications for the present.

\(^5\)Ordered means that the order matters: \((2, 1)\) is not to be the same as \((1, 2)\).
the vector. As soon as we begin working with vectors having many elements, we will see other things besides just [1] at the beginnings of lines.

Variables

Very often we will want to use and re-use vectors of data. For this, it is important to be able to assign vectors to variables, which are named entities that we create in R to store such data. For example,

```r
> a <- c(2, 1, 3)
```

The <- operator (which is typed as the less than symbol < followed by a hyphen −) is called the assignment operator in R, and the code above is read, “the vector (2, 1, 3) is assigned to the variable a”, or more succinctly “a gets (2, 1, 3)”. While the second of these statements reads from left to right the same way that the code reads, both statements indicate clearly that the direction of assignment is the direction in which the arrow points.

In any case, the result of the assignment is that the variable a, which was created by this statement if it didn’t exist before, now stores the vector (2, 1, 3). To verify this, we type:

```r
> a
```

```
[1] 2 1 3
```

Very often it is nice to verify what was actually stored in an assignment without having to type the variable name at the next prompt. Enclosing an entire assignment statement in parentheses does exactly this:

```r
> (b <- c("fish", "sky", "tree", "squirrel"))
```

```
[1] "fish" "sky" "tree" "squirrel"
```

We will often use such enclosing parentheses in this guide, in order to display what is getting assigned while we assign it. It should be understood that the parentheses are not necessary for the variable assignment,

---

6 For those familiar with other computer programming languages, it is worth noting that in R, the first element of a vector is in fact numbered 1, not 0 as in some programming languages.
but that we are including them here solely for pedagogical reasons, to help someone who is reading the code and its output.

We should keep in mind that whenever we make a variable assignment, if the variable is not already in existence, then it will be created. If it is already in existence, then it will be written over with the new value, and the old value that it stored will be lost. This is usually a good feature, but in any case it is something that the R user should be aware of.

Also, without going into full detail about what characters are and are not allowed in variable names, some words about naming variables are in order. First, we should restrict the characters in our variable names to small and capital letters, numbers, underscores, and periods. In addition, we should use variable names that are meaningful but not overly long. For example, if a sample consists of measurements of heights of elm trees, a good variable name would be \texttt{height}. A variable name that is too short for most purposes would be \texttt{h} or \texttt{ht}, or even worse something like \texttt{x} that gives no indication at all of the nature of the data. A variable name that is too long for most purposes would be \texttt{elm.tree.height}. In general, we should aim for a balance between readability and typability, usually erring on the side of readability if need be.

Hyphens and spaces are not allowed in variable names in R. This complicates matters when we want to have variables with more than one word in their names. While there are many ways to get around this difficulty, we should be consistent in our variable-naming convention. Two commonly used conventions for naming variables with more than one word are:

1. Begin the variable name with a small letter, and capitalize all the later words in the variable, as in \texttt{elmTreeHeight}.

2. Use periods (or possibly underscores instead) to separate words in variable names, as in \texttt{elm.tree.height}.

We will use the first of these conventions in this guide, but it is fine to use another variable naming scheme consistently instead.

We should keep in mind though that R is case-sensitive, meaning that \texttt{elmtreeheight} and \texttt{elmTreeHeight} are different variables in R, as are \texttt{height}, \texttt{Height}, and \texttt{HEIGHT}. This can cause a great deal of confusion if we are not aware of it. It can also be used to write very confusing R code: if we have two variables named \texttt{height} and \texttt{Height}, the computer will know the difference but anyone reading the code will have a very difficult time keeping
track of the different variables. Because of this, we should generally avoid having such similarly named variables.

*Vectorized operations and useful vector functions*

Many operations in R, including the four basic arithmetic ones, are *vectorized*, meaning that when applied to vectors, the operations are performed on each element individually. For example, we have:

```r
> x <- c(1, 2, 3)
> y <- c(4, 6, 5)
> x + y
[1] 5 8 8
> x - y
[1] -3 -4 -2
> x * y
[1] 4 12 15
> x/y
[1] 0.2500000 0.3333333 0.6000000
> x^2
[1] 1 4 9
> sqrt(x)
[1] 1.000000 1.414214 1.732051
```

If we are wondering how a function in R that is usually applied to numbers behaves on vectors in R, we can simply try it out. The function is very likely to be vectorized in a sensible way.

A function that is useful when dealing with vectors is the `length(vector)` function, which gives the number of elements that the specified vector has. For example:
Two other useful functions are the `sum(vector)` and `prod(vector)` functions, which compute the sum and the product of all the elements in the specified numerical vector. For example, using the vector given immediately above, we have:

```r
> sum(a)
[1] 16

> prod(a)
[1] 105
```

## 0.3 Getting data into R

When using R for statistical computations, the first step is usually to get the data into R somehow. There are many possible ways to do this, but we will focus on some of the basic ones here.

### Small and simple samples

Very small samples, perhaps of size 20 or less, might be entered into R manually by simply assigning a data vector to a variable, as in:

```r
> (height = c(10.7, 12, 13.1, 11.4, 18.2, 18.4, +
  17.3))
```

```
[1] 10.7 12.0 13.1 11.4 18.2 18.4 17.3
```

For very small samples, this might be the most efficient way to put the data into a usable format. However, this method is highly error-prone and should not be used at all for larger data sets. Using this method in such cases is asking for trouble, and amounts to using R as something that it is not. There are much better tools than R available for manual
manipulation of larger data sets (if such manipulation is even necessary), such as spreadsheet programs.

If the data happens be an arithmetic sequence (i.e., differing by the same amount between any two consecutive terms), then R has some easy ways to enter it. While this is usually less practical as a data entry method, we will see that it has many other uses, such as in subsetting data.

To generate a vector of consecutive integers from $a$ to $b$ in R, we can simply enter $a:b$. For example:

```r
> 3:7
[1] 3 4 5 6 7
> -2:5
[1] -2 -1 0 1 2 3 4 5
> 8:2
[1] 8 7 6 5 4 3 2
> 2:10.3
[1] 2 3 4 5 6 7 8 9 10
> 2.4:7
[1] 2.4 3.4 4.4 5.4 6.4
```

If the right endpoint is not an integer more than the left endpoint, the sequence merely continues until the next value would be beyond the right endpoint. Also, note that the sequence begins at the left endpoint, whether or not that endpoint is an integer.

For sequences that skip by the same amount each time, but not necessarily by 1, the `seq(start, end, by=skip amount)` function can be used. For example,

```r
> seq(1, 10, by=2)
[1] 1 3 5 7 9
```
> seq(18, 2, by = -4)
[1] 18 14 10 6 2
> seq(3, 28, by = 3.3)
[1] 3.0 6.3 9.6 12.9 16.2 19.5 22.8 26.1

If it is easier to say how many elements we would like in the sequential vector rather than how much should be skipped between terms, the `seq(start, end, length=number of terms)` function can be used instead, as in:

> seq(1, 10, length = 3)
[1] 1.0 5.5 10.0

Data frames

Usually we will have more than just one variable that we are working with, so it is convenient to have a way to store observations of several different variables on individual subjects. In R, this can be done with a data frame, which is a matrix whose columns are vectors. While different columns can be vectors of different types (such as numbers, strings, or logical values), elements within the same column must be of the same type. This is consistent with the usual usage of data frames, in which rows correspond to observations and columns correspond to variables. That is, the elements of each row are observations of several variables for a single subject; the elements of each column are observations of a single variable for several subjects.

To illustrate this concretely, we use one the ToothGrowth data frame, which is one of the built-in data frames in R included with the installation.

> ToothGrowth

This is actually much more data that we would like to display, especially in a printed book such as this, so we have suppressed the output. From now on in such situations, we will simply use the `head(data frame)` function to display only the first 6 observations (rows) in the data frame:
Section 0.3 Getting data into R

> head(ToothGrowth)

       len supp dose
1     4.2  VC  0.5
2    11.5  VC  0.5
3     7.3  VC  0.5
4     5.8  VC  0.5
5     6.4  VC  0.5
6    10.0  VC  0.5

This is a good time to remind the reader that *data without context is useless*. These 60 rows and 3 columns mean nothing unless we know where they came from. For this, we enter

> help("ToothGrowth")

This gives us the help page for the built-in ToothGrowth data frame (data frames that aren't built-in but that we create don't have help pages), which explains that the data frame is a collection of observations of the teeth of guinea pigs. In the Format: section of the help page, we learn that the first column len (indicated by [,1], a notation that we will discuss later) gives the length of the tooth, the second column supp indicates the supplement given to the guinea pig, and the third column dose gives the dose of the supplement in milligrams. Notice how important all of this information is in trying to understand and analyze this data set.

This ToothGrowth data frame is a typical example of a data frame. The first and third columns are both numerical vectors. The second column is a particular type of vector called a *factor*, which we will discuss later. Suffice it to say for the present that a factor is different from the three types of vectors that we have encountered so far.

Any column can be accessed and used individually using the notation `data frame$column name`, as in:

> ToothGrowth$len

      [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5
     [12] 16.5 15.2 17.3 22.5 17.3 13.6 14.5 18.8 15.5 23.6 18.5
     [23] 33.9 25.5 26.4 32.5 26.7 21.5 23.3 29.5 15.2 21.5 17.6
     [34]  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3 23.6 26.4
     [45] 20.0 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8
     [56] 30.9 26.4 27.3 29.4 23.0
> head(ToothGrowth$supp)
[1] VC VC VC VC VC VC
Levels: OJ VC

> 10 * ToothGrowth$dose

[1]  5  5  5  5  5  5  5  5 10 10 10 10 10 10 10 10
[19] 10 10 20 20 20 20 20 20  5  5  5  5  5  5  5
[37]  5  5  5 10 10 10 10 10 10 10 10 10 20 20 20 20
[55] 20 20 20 20 20 20

We should keep in mind the case sensitivity of R though. An easy mistake to make is to forget to capitalize a data frame name or a column name when it is in fact capitalized.

Although it is not practical for even moderately large data sets, we can enter data frames into R by hand, first by assembling the column vectors and then by using the `data.frame()` function, as in:

> a <- c("red", "blue", "green", "purple")
> mass <- c(10, 14, 11, 20)
> (pens <- data.frame(color = a, mass, c(-1, 1, + 4, 4)))

color mass c..1..1..4..4.
1  red 10 -1
2  blue 14 1
3 green 11 4
4 purple 20 4

Notice that if the column name is not included, it will default to the name of the vector placed there, as in the second column above. If the column is created in the `data.frame()` function, as in the third column above, R will make up a name for it, usually a rather unappetizing one, as above.

Two useful functions relating to data frames are `nrow(data frame)` and `ncol(data frame)`, which give the number of rows (observations) and columns (variables) in the data frame. For example, using the data frame `pens` constructed above:
> nrow(pens)
[1] 4

> ncol(pens)
[1] 3

This tells us that the pens data frame has 4 observations and 3 variables.

If we want to see the names of the columns, we can use the `names(data frame)` function. For example,

> names(pens)

[1] "color"    "mass"    "c..1..1..4..4."

What’s more, if we want to change the column names, we can use the `names(data frame)` function in an assignment, as in:

> names(pens) <- c("color", "mass", "rating")
> pens

    color  mass rating
   1 red   10    -1
   2 blue  14     1
   3 green 11     4
   4 purple 20     4

The `row.names(data frame)` function works similarly for row names. Row names are by default simply the numbers from 1 to the number of rows. However, we can make them something else if we like, as in:

> row.names(pens)

[1] "1" "2" "3" "4"

> row.names(pens) <- c("Bic", "Pilot", "Zebra", +    "Unknown")
> pens
Section 0.3 Getting data into R

```r
> row.names(pens) <- 2:5
> pens
  color mass rating
  2 red 10 -1
  3 blue 14 1
  4 green 11 4
  5 purple 20 4
```

Row names, being labels for individual observations, must be unique, and any attempt to set non-unique row names will result in an error. (Column names should be unique too, but this is not enforced by R.)

**Reading data files into R**

It is quite rare that one would actually enter data directly into R for analysis. A lot of data nowadays is generated electronically, and when it is not, data entry personnel are often hired. Even if this is not the case, data is usually entered in the form of one or more spreadsheets, very often into Microsoft Excel. So while R has the ability to have data entered directly, it is usual neither practical nor advisable to do so.

Since data is most commonly entered into Microsoft Excel before processing and analysis, we will focus here on how to import data from Microsoft Excel. The techniques described here carry over with little modification to importing data in other formats, and we will mention briefly how to do so after discussing Microsoft Excel data.

Microsoft Excel stores enormous amounts of additional information (formatting and otherwise) besides simply data, and consequently there is no really nice way to read an Excel file directly into R. Various authors have developed functions to do so, but none of them seem as straightforward or widely applicable as the method we now describe.

To import data into R from Excel, first understand that the following method works only for a single Excel worksheet at a time. If we have
multiple worksheets in a given Excel file, we will want to import each one of them separately. In addition, we should know that preferably our data should be in table format, laid out exactly as we want the data frame in R to be laid out. That is, we will want rows to correspond to observations and columns to correspond to variables. Also, column names should be given in the first row, and the data should be stored in the rest of the rows. It is worth noting that the only things that will actually be imported are column names and observations. All formatting (colors, boldface, font sizes, etc.) will be ignored and lost in the import to R. If we are entering the data into Excel ourselves (which should not ordinarily be the case, except possibly with very small data sets), we can lay it out in this fashion, ready to be imported into R. If someone else has entered the data and given it to us, we will probably need to do some cutting and pasting in Excel before importing the data into R.

Once we have a worksheet in a suitable form in Excel, we should then save the worksheet of interest in Excel as a .csv (comma separated value) file. This can be done through the “Save as” menu option in Excel. Among the list of choices for file formats, there may be more than one variety of .csv file. While it probably doesn’t make much difference which one we choose (and if one doesn’t work, we can try another), usually the simplest one that doesn’t specify any particular operating system is preferable.

When we have saved the file (and noted its location on the computer), we can read it into a data frame with the read.csv(file.choose()) function. However, this function doesn’t actually store the imported data (it just imports it), so we will want to assign the imported data to a data frame of our choice. For example, to read a file called birdData.csv into a data frame called birds, we use:

```r
> birds <- read.csv(file.choose())
```

When we enter this at the command prompt, we will be given a dialog box that allows us to select the desired file birdData.csv, and as soon as we give R the okay, the data from this .csv file will be stored in the data frame birds.

In short, once the data is laid out suitably in an Excel worksheet and saved as a .csv file, then it is easily imported into a data frame in R with a single command. However, it often does take a bit of doing to put the data in a suitable format in Excel.
Of course, .csv files don’t have to have anything to do with Excel (although they can always be read into Excel if so desired). In fact, data sometimes comes from a .csv file on a website. While it is quite possible to save the file on our computer and then load it into R with the method just described, saving the file on our computer separately is not necessarily. Instead of the file.choose() argument to the read.csv() function, we can simply enter the data website’s url (e.g., http://www.someWebsite.com/someFile.csv) in quotation marks, as in:

```r
> birds <- read.csv("http://www.someWebsite.com/someFile.csv")
```

This can be quite convenient, especially when using pre-prepared data sets (in a statistics class, for example).

In order to accept a slightly wider range of layouts, some of the additional optional parameters in the read.csv(file.choose()) function can be used. These include whether or not the data file contains column names, how many lines to ignore at the top of the file, which character strings should be interpreted as missing values, etc. To learn more about these options, see the help page for the read.csv() function.

For an even wider range of possible layouts (not necessarily emanating from Excel) including tab-delimited and whitespace-delimited data, the read.table() function is provided. Its use is quite similar to read.csv(), except that you need to tell it to include the column names (with the header=TRUE parameter) and you need to specify the delimiter (for example, sep="t" for a tab-delimited file).

```r
> birds <- read.table("myFile.txt", header = TRUE, + sep = "\t")
```

For further information on this function, see its help page. It has many other options, giving it considerable flexibility.

For importing data from a number of other standard statistical packages (such as SPSS, SAS, Minitab, etc.), the foreign package is quite useful. Consult this package’s help page with library(help="foreign") for further information.
0.4 Working with data

Now that we have discussed some ways to get data into R, we turn our attention to working with that data.

Subsetting vectors

R is equipped with powerful and very useful subsetting tools. Although their versatility might not be apparent when we are just beginning to use R, these are tools that will come in very handy as we continue to work with data in R.

Often we don’t want to work with an entire vector. Sometimes we’re interested in just a single element of that vector. For example, suppose that we have a data vector defined as follows:

```r
> (temperatures <- c(10, 20, 30, 40, 50, 60, 70))
[1] 10 20 30 40 50 60 70
```

This might be a sample of temperature readings (in degrees centigrade, say). If we are only interested in the temperature reading from the third day, we can extract it and work with it using the subsetting brackets `[`], as in:

```r
> temperatures[3]
[1] 30
```

```r
> temperatures[3] + 10
[1] 40
```

Along these lines, sometimes the output from an R function is in the form of a vector, and we are only interested in a single element of that vector. This kind of subsetting allows us to extract the desired single piece of information.

If we are interested in more than one value, we can extract those values by enclosing the index numbers of the desired values as a vector in the subsetting brackets:
> temperatures[c(1, 3, 6)]
[1] 10 30 60

> temperatures[2:5]
[1] 20 30 40 50

> temperatures[seq(1, 7, by = 2)]
[1] 10 30 50 70

Notice how this can be combined with the methods of creating sequential vectors to create sequential subsets (e.g., all odd-numbered indices of a vector, as above).

In order to exclude certain values from the vector, we use negative indices:

> temperatures[-7]
[1] 10 20 30 40 50 60

> temperatures[c(-1, -3)]
[1] 20 40 50 60 70

This first method of subsetting by using the subsetting brackets enclosed by a numerical vector of indices (or negative indices) is particularly useful when the desired index numbers are actually known.

The second method of subsetting is more useful when the desired index numbers are not known explicitly but instead satisfy some condition. In this method, the subsetting brackets [ ] are still used, but this time they enclose a logical vector of the same length as the vector being subsetted. The elements that are included in the subset are precisely those for which the corresponding element in the logical vector is TRUE. For example:

> temperatures

[1] 10 20 30 40 50 60 70
> temperatures[c(TRUE, FALSE, TRUE, FALSE, FALSE, + FALSE, FALSE)]
[1] 10 30

Here since the only elements for which the subsetting vector is TRUE are the first and third elements, the extracted subset consists of the first and third elements of the temperatures vector.

This wouldn’t be such a useful tool if creating logical vectors weren’t so easy in R. Logical vectors are rarely entered by hand; instead, they are usually defined by conditions, as with:

> temperatures > 20
[1] FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE

> highTemperatures <- (temperatures > 20)

Here we see that the logical vector temperatures > 20 is the same length as the vector temperatures, and it has a TRUE value precisely when the specified condition is satisfied. It might seem strange at first to think of a statement such as temperatures > 20 as a logical vector, but it is a very useful point of view. Notice in the second statement above that we were able to store this logical vector as a vector called highTemperatures for future use. By the way, the parentheses in the variable assignment above are not necessary but help increase the readability of the code, which is why we included them.

Now that we have defined the highTemperatures logical vector, we can use it to subset the temperatures vector:

> temperatures[highTemperatures]
[1] 30 40 50 60 70

Notice that, by design, exactly the temperatures that satisfied the condition of being greater than 20 degrees were extracted. And of course, storing the condition as a vector named highTemperatures is not necessary at all. Instead we could have listed the condition in the subsetting brackets explicitly, as in:
Section 0.4  Working with data

> temperatures[temperatures > 20]

[1] 30 40 50 60 70

In summary, the two main ways to subset a vector both involved the subsetting brackets. The first way uses a numerical vector of desired indices inside the subsetting brackets, while the second way uses a logical vector for which `TRUE` indicates inclusion in the subset.

Subsetting data frames

Subsetting a data frame works just like subsetting a vector, except that there are two indices in data frames, the first for rows and the second for columns. So while subsetting a vector is accomplished by appending `[subset]` to the vector’s name, subsetting a dataframe works by appending `[row subset, column subset]` to the data frame name. As with vectors, the row and column subsets can each be specified either by giving a vector of numerical indices or by giving a logical vector for which `TRUE` indicates inclusion in the subset. For example, consider the `pens` data frame that we constructed earlier:

> pens

    color mass rating
   1 red    10   -1
   2 blue   14    1
   3 green  11    4
   4 purple 20    4

> pens[2, 1]

[1] blue
Levels: blue green purple red

> pens[c(FALSE, TRUE, FALSE, FALSE), 1]

[1] blue
Levels: blue green purple red

> pens[1, -2]

> pens[1, -2]
color rating
1 red -1

> pens[c(1, 4), 2:3]

mass rating
1 10 -1
4 20 4

Note that to include all rows or all columns, we leave the corresponding subset blank in the subsetting brackets. For example:

> pens[4, ]

color mass rating
4 purple 20 4

> pens[2:4, ]

color mass rating
2 blue 14 1
3 green 11 4
4 purple 20 4

> pens[-4, ]

color mass rating
1 red 10 -1
2 blue 14 1
3 green 11 4

> pens[, 3]

[1] -1 1 4 4

> pens[, c(1, 3)]

color rating
1 red -1
2 blue 1
Also, subsetting the row with a logical vector constructed via a condition is particularly useful in selecting a subset of observations to use, as in:

```r
> pens[pens$rating >= 0, ]

   color mass rating
2   blue  14     1
3  green  11     4
4 purple  20     4
```

Notice that even though the condition is being used inside subsetting brackets, the prefix “pens$” still has to be included before the `rating` column name. Also note the use of “>=” for “greater than or equal to” (and “<=” is for “less than or equal to”, and “!” is for “not equal to”). This type of subsetting of observations is quite common and is a very handy feature in R, so it is important to become comfortable with it.

To combine two or more conditions with “and”, use `&`; to combine two or more conditions with “or”, use `|`, which is Shift-on many computers, just above the Enter key.

Now we have already learned that individual columns can be accessed by name, so the `pens[,3]` command above is exactly the same as the command `pens$ratings`. While there isn’t an analogous way to access multiple columns with the “$” notation, columns (multiple columns if desired) can be accessed by name with subsetting brackets, as in:

```r
> pens[, "rating"]

[1] -1 1 4 4

> pens[, c("color", "rating")]

    color rating
1    red    -1
2  blue     1
3  green     4
4 purple     4
```

```r
> pens[2, "color"]
```
If you don’t need to specify the name or number of the rows and columns that you’d like to subset, but would instead prefer to subset based on a condition, the `subset(data frame, row condition, select=vector of columns)` function is highly useful because you don’t have to enter the name of the data frame repeatedly. Also, the column names don’t have to have quotation marks. For example:

```r
> subset(pens, rating >= 0)

   color mass rating
 2   blue   14     1
 3  green   11     4
 4 purple  20     4

> subset(pens, select = c(color, rating))

   color rating
 1   red     -1
 2  blue      1
 3 green      4
 4 purple     4

> subset(pens, (rating >= 0) & (rating < 4), select = c(color, +

   mass))

   color mass
 2  blue   14

> subset(pens, (rating == -1) | (rating == 4))

   color mass rating
 1   red    10    -1
 3  green    11      4
 4 purple   20      4
```

These basic subsetting methods provide useful tools in working with data in R. There is only one more main item to discuss before we actually begin to analyze data with R, namely R scripts.
A script is a type of computer program, one that is interpreted during its execution rather than compiled ahead of time. Scripts are typically fairly short, but they do not have to be.

Scripts in R consist of lists of commands that could be typed at the command prompt. A single command then instructs R to execute all these commands in the given order. Those who have some familiarity with computer programming can probably already see why we might want to prepare a script in R and execute it instead of entering the commands ourselves at the command prompt. For those without a computer programming background, however, we should mention that preparing scripts is a very important part of conducting statistical analyses. In fact, whether we use R or any other statistical software in the future, we should always write a script for any kind of serious statistical analysis. It is quite difficult to communicate with others about the analysis, or even with yourself a week or a month or more later, without a well-documented, readable script. So while this may be one of the most unfamiliar aspects of R at first, it is in fact not an aspect of R but of conducting a statistical analysis. It happens to be prominently displayed in R because R was originally designed by statisticians for statisticians, but it is an important part of a statistical analysis whatever statistical software we use (and regardless of whether or not it is a featured part of that software).

Another reason for placing such emphasis on writing scripts is that they can easily be modified and reused, in whole or in part. This means that when new aspects of a statistical analysis come to light or when a journal referee suggests modifications to an analysis, the script can be changed and adapted appropriately with little effort. Without a script, any such required or desired modifications would demand manually entering the entire analysis from the command prompt again (if we can even remember exactly what it was that we entered before). Also, for future similar analyses of different data sets or experiments, the appropriate parts of the original analyses can be reused and modified, cutting down on the amount of work necessary and increasing the consistency of the similar analyses.
Making an R script

To prepare an R script, we should use a good text editor. A text editor is like a word processor except that it does not allow editing or saving of a document’s formatting, such as font sizes, boldfacing, italics, centering, etc. A text editor edits only the text. The editor may be configured to display computer code in a nice way (e.g., with colors for computer commands, etc.), but such formatting is not actually part of the script and is not saved with the script.

Many excellent text editors are available for free. Two that are designed to integrate specifically with R are Tinn-R (Windows only) and JGR (cross-platform). Searching for these on the internet will quickly lead us to sites where they can be downloaded for free. However, both of these programs are best suited to at least slightly advanced computer users, so those who are new to computer programming might prefer either Notepad++ (Windows) or TextWrangler (Mac). Both of these are very good, simple-to-use text editors that are more than adequate for any of the scripting tasks that we will discuss in this guide.

Once we have chosen and installed a text editor on our computer, to begin writing an R script, we use the text editor to open a new, empty file. Then in the text editor, we place a sequence of commands, one per line, as in:

\[
\begin{align*}
2+2 \\
a &\leftarrow 3:9 \\
a^2
\end{align*}
\]

Notice that we do not type the “>” that we normally see at the R command prompt, but rather only the commands themselves.

Be aware that we should always use Enter to finish a line in an R script; we should not allow any sort of line wrapping to occur. When an R script is executed, R will determine how lines are separated based on where Enter occurs in the file, not on how the script looks on the screen. Even if a line has wrapped around to the next line on screen, it will not be treated as a separate line by R unless we have pressed Enter.

We will look at how to use this script in R in the next section, but for the moment, we simply use the text editor to save the file that we just created. We can name it anything we like, but the file suffix (which tells the computer the file type, so that it knows what to open the file with when
it is clicked on) should generally be either .R or .txt. Ordinarily, .R is the preferred suffix for an R script, but a script file is simply a text file, so it wouldn’t be too misleading to use .txt as a file suffix either.

One important thing to know about saving a script is that it should be saved in plain text format. This is usually the default in a text editor, but for some reason there is a built-in Mac text editor that defaults to rich text format (with a .rtf extension), which cannot be read by R. In any case, whatever text editor we use, do not save the script in rich text format.

For more complicated R scripts, we may find it easier to cut and paste commands from the R command prompt to the text editor. This way, if we try a command and it works and does what we want it to at the command prompt, we can transfer it to the script without introducing any typographical errors.

### Executing an R script

Once we have created an R script, we can execute (or run) it in R with the `source(file.choose())` function:

```r
> source(file.choose())
```

This will open up a file selection dialog box, and when we select the script file, all the functions in it will be executed, almost as if we had entered them one at a time at the R command prompt. Sometimes the output from the script isn’t quite the same as what we would get from the command prompt, but usually it is about the same, so we won’t worry about this difficulty here.

The `source(file.choose())` function isn’t always the most useful way to run a script because it doesn’t display the commands that were entered to generate the output. For this, we use the optional `echo` argument to the `source()` function. When the `echo` argument is set to true, both the commands themselves and the output will be displayed:

```r
> source(file.choose(), echo = TRUE)
```

If we run into a problem with long commands (or perhaps comments, to be discussed soon) getting truncated when the script is run, we can set the `max.deparse.length` to something longer than its default 150 characters per command line:

```r
> source(file.choose(), echo = TRUE, max.deparse.length = 2000)
```
Navigating directories

One thing to take into account when writing a script is where any files that it uses (such as data files) are to be located. It is not a good idea to include full path names of files (such as `C:/Users/JamesB/Desktop/R Files`) in a script, since this generally makes it impossible to run the script on any computer other than the one on which it was written.

Instead we usually assume that files to be used will be contained in what is called the current **working directory** of R whenever the script is run. Roughly speaking, the working directory is the folder in which R will look for any files when instructed to do so from the command line. To view the current working directory, we use the `getwd()` function, which is short for “get working directory”:

```
> getwd()
```

Although we have suppressed the output here (since the directories on my computer are of no relevance to this book), it would be something like [1] "C:/Users/JamesB/Desktop/R Files", which would indicate that R will look for files in the folder entitled “R Files” on my computer’s desktop (which is actually itself a folder in the JamesB folder, which is in the Users folder in the C: drive of the computer).

We can change working directories with the `setwd("directory name")` function. However, most beginning R users find it easier to change working directories using menus instead of this function, so we merely refer the interested user to the help page of `setwd`. To change the working directory using menus, simply select the **Change dir...** option from the **File** pull-down menu at the top of the R interface. (This may be located in the **Misc.** pull-down menu on Mac computers.)

When running an R script that accesses files, we should first make sure that we have set the working directory appropriately. This will usually mean that the working directory contains the files that the script accesses.

Rather than changing the working directory with menus or with the `setwd("directory name")` function for the purpose of then running a script, if the script is in the folder that is desired for the working directory, we can set the optional `chdir` argument for the `source(file.choose())` function to `TRUE`. This will cause the working directory to be set to the the folder in which the R script is contained:

```
> source(file.choose(), chdir = TRUE)
```
Now the we have discussed the working directory, we can also mention that anywhere that we have used `file.choose()`, we can actually substitute `file="file name"` instead whenever the file is in the current working directory. For example, to run a script entitled `myScript.R` (notice that the file extension, which is `.R` here, must be included), we can use

```r
> source(file = "myScript.R")
```

For this, it is often useful to be able to see all of the files that are contained in the working directory, which can be accomplished with the `dir()` function:

```r
> dir()
```

This will output (a vector of) all the names of the files in the current working directory, although the output has been suppressed here. Individual files can then be referred to by name, or can be referred to by subsetting this vector. For example, if `myScript.R` is the third file listed in the `dir()` output, then `file=dir()[3]` (without quotation marks) could be used instead of `file="myScript.R"`.

Comments and the readability of scripts

Aside from functionality, perhaps the most important aspect of a script is its readability. Scripts are meant to be executed by computers, but they are meant to be read and understood by human beings. If they cannot be read by human beings, we cannot check that the script really does what it is supposed to, and we can’t modify, reuse, or adapt the script at a later date. Worse yet, debugging a script that is not readable can be difficult to impossible. For those without a background in computer programming, debugging is the process of finding and removing errors, or bugs, from a computer program. As we begin to write R scripts, we will become familiar with the process of debugging.

How then do we make an R script readable? First and foremost, the script should include tons of comments, which are statements that are not executed by R but are included for clarity. To make a comment in R, simply enter “#”, either at the command line or in a script. All characters for the rest of the line will then be a comment and will not be executed, as in:
Students just beginning to use R are strongly recommended to have at least one comment per line in an R script. Generally that comment should be placed before the code, and it should help to clarify what the code will do. Remember that if we want to edit, modify, improve, or reuse this script at a later time, we will need to be able to read the script. The comments should be written with this in mind, as in:

```r
> # use height to store the plant's height in cm
> height <- 27
```

Notice that the comment contains information that is not obvious from the code itself, but this helps the reader understand why this particular piece of code is being used and what its function is. It wouldn’t be necessary or helpful to include a comment such as “height is 27” before this code instead. The reader can already tell that from the code itself, and important pieces of information about what the variable represents are omitted. As we continue to use R and write R scripts, we will become familiar with which comments are helpful and which are not; for now just try to keep readability (by others and by yourself at a later date) in mind.

Another thing that can aid readability is the use of whitespace and symbols to delineate sections of the script. Since other formatting such as boldfacing and font resizing is not stored as part of an R script, the placement of whitespace and symbols should be used for layout instead. Extra whitespace is completely ignored by R, so it can be inserted generously to the reader’s advantage. A typical example of this is:

```
############################
# Fitting the linear model #
############################
# make the actual fit and store it in fitModel
fitModel <- lm(y~x, data=myData)

# extract the constant term beta_0
beta0 <- coef(fitModel)[1]

# is this bigger than the previous model's constant term?
beta0 < oldBeta0
```
We can make even bigger section subdivisions using extra #’s, or other characters following a single # if we prefer. The point is that it should be easy for the reader scanning through the file to see where the major sections of the R script are.

Perhaps the most important comments that we will include in a script though are at the very beginning of the script. We should get in the habit of putting an extensive preamble into our scripts, indicating what the script is, by whom it was written, when it was written and/or revised (since scripts do get revised, it is particularly important to be aware of which versions are the newest), what other files or packages are needed, etc. In short, include anything you or anyone else would need to know in order to run the script on any computer any time. If these things aren’t carefully documented, a perfectly functioning script may be rendered useless because no one knows how to use it.\footnote{I seem to recall a Twilight Zone episode to this effect, where a mob storms a place and overthrows the overlords, only to discover that they just destroyed the only people who knew how to use a particular computer that was crucial to their society.}

An example of such a script header is:

```
# Analysis of Sarah Jackson's sea whale data #
# by James Bernhard #
# revision date: June 23, 2009 #
#
# input: whale.csv file in current working directory #
# output: fitted linear model coefficients #
# diagnostic scatterplots #
```

This might look like a lot of comments to enter before we have entered any executable code, but the habit of doing so is important for anyone planning to do serious statistical analyses in R (and in fact for anyone planning to write scripts or programs of any sort).

And now, with these preliminaries in place, we are indeed ready to begin learning how to conduct statistical analyses in R.
0.6 Problems

1. There is a function in R that computes the cumulative sum vector of a numeric vector. Use the help system in R to find it.

(a) Which R command(s) did you use to locate this function?
(b) What function is it?
(c) Execute the example at the bottom of this function’s help page to illustrate its use, and explain in your own words (possibly based on the understanding you get from the example) what is meant by the cumulative sum vector of a numeric vector.

2. Use R to compute

\[(1 \cdot 4) + (-3 \cdot 5) + (2.3^2 \cdot 2) + (4 \cdot .67)\]

in two different ways:

(a) First, just by entering the above expression into R at the command line.
(b) Second, vectorially, by computing the sum of the product of two vectors. (You should figure out what the relevant two vectors are.)

3. The following is a very realistic data cleaning exercise.

The original file from which the anthrokids data set is extracted is located at: http://ovrt.nist.gov/projects/anthrokids/orig77/individuals.csv. Download this file and edit it as needed (without modifying the data of course) in order to read it into R.

The documentation for this data is in the (30 MB) pdf file at: http://ovrt.nist.gov/projects/anthrokids/child771nk.pdf. Use R (not Excel, except to eyeball the data) and this documentation to help you in the following tasks.

(a) How many variables are there in this data set?
(b) How many observations are there in this data set? (Each row in the data set counts as a single observation.)
(c) A glance at the data in Excel reveals many 0 values, so many that it is apparent that missing values have been encoded as 0. How many children have 0 listed as their “ERECT SITTING HEIGHT”?

(d) Within R, change all the erect sitting heights that are listed as 0 to equal NA, which (when used without quotes) denotes a missing value in R. Do not show the output here, but only the command(s) that you used.

(e) If the column title is to be believed, the “AGE IN YEARS” column seems to indicate that these children are thousands of years old. This seems unlikely. What are the correct units for this column?

(f) There is also an “AGE IN MONTHS” column. This should be derivable from the “age in years” column, and vice versa. In this particular data set, which one is actually derived from the other one, and what is the exact relationship between the two columns?

(g) The column labeled “WEIGHT” is rather meaningless if we don’t know the units. What are the correct units? (For those with some background in the natural sciences, you might also think about why “WEIGHT” is not a good name for this column, and what a better name would be.)

(h) In the “SEX” column, which sex is 1, and which is 2? How do you know?

(i) In the “HANDEDNESS” column, which is 1 and which is 2? How do you know?

(j) How many children have “HANDEDNESS” listed as a number other than 1 or 2?

(k) Which values besides 1 and 2 occur for “HANDEDNESS”, and what do they mean?

(l) What does the “STATURE” column record, and in what units?

(m) What does the “BIRTH ORDER” column record? Are there any values that occur that might require additional explanation? If so, what are they, and how might they be explained?