UNIT-I

INTRODUCTION TO R

How to Run R :

R operates in two modes: *interactive* and *batch*. The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on. On the other hand, batch mode does not require interaction with the user. It's useful for production jobs, such as when a program must be run periodically, say once per day, because you can automate the process.

Interactive Mode

On a Linux or Mac system, start an R session by typing R on the command line in a terminal window. On a Windows machine, start R by clicking the R icon.The result is a greeting and the R prompt, which is the > sign. The screen will look something like this:

```
R version 2.10.0 (2009-10-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
```

You can then execute R commands. The window in which all this appears is called the R *console*. As a quick example, consider a standard normal distribution—that is, with mean 0 and variance 1. If a random variable X has that distribution,
then its values are centered around 0, some negative, some positive, averaging in the end to 0. Now form a new random variable $Y = |X|$. Since we've taken the absolute value, the values of Y will *not* be centered around 0, and the mean of Y will be positive. Let's find the mean of Y. Our approach is based on a simulated example of $N(0,1)$ variates.

```
> mean(abs(rnorm(100)))
[1] 0.7194236
```

This code generates the 100 random variates, finds their absolute values,and then finds the mean of the absolute values.
The [1] you see means that the first item in this line of output is item 1.In this case, our output consists of only one line (and one item), so this is redundant. This notation becomes helpful when you need to read voluminous output that consists of a lot of items spread over many lines. For example, if there were two rows of output with six items per row, the second row would be labeled [7].

```
> rnorm(10)
[1] -0.6427784 -1.0416696 -1.4020476 -0.6718250 -0.9590894 -0.8684650
[7] -0.5974668 0.6877001 1.3577618 -2.2794378
```

Here, there are 10 values in the output, and the label [7] in the second row lets you quickly see that 0.6877001, for instance, is the eighth output item.You can also store R commands in a file. By convention, R code files have the suffix *.R* or *.r*. If you create a code file called *z.R*, you can execute the contents of that file by issuing the following command:

```
> source("z.R")
```

Batch Mode

Sometimes it's convenient to automate R sessions. For example, you may wish to run an R script that generates a graph without needing to bother with manually launching R and executing the script yourself. Here you would run R in batch mode
.As an example, let's put our graph-making code into a file named *z.R* with the following contents:

```
pdf("xh.pdf") # set graphical output file
hist(rnorm(100)) # generate 100 N(0,1) variates and plot their histogram
dev.off() # close the graphical output file
```

The items marked with # are *comments*. They're ignored by the R interpreter.Comments serve as notes to remind us and others what the code is doing, in a human-readable format.
Here's a step-by-step breakdown of what we're doing in the preceding code:
• We call the pdf() function to inform R that we want the graph we create to be saved in the PDF file *xh.pdf*.
• We call rnorm() (for *random normal*) to generate 100 $N(0,1)$ random variates.
• We call hist() on those variates to draw a histogram of these values.
• We call dev.off() to close the graphical "device" we are using, which is
the file *xh.pdf* in this case. This is the mechanism that actually causes the file to be written to disk.
We could run this code automatically, without entering R's interactive mode, by invoking R with an operating system shell command (such as at the $ prompt commonly used in Linux systems):

```
$ R CMD BATCH z.R
```

You can confirm that this worked by using your PDF viewer to display the saved histogram. (It will just be a plain-vanilla histogram, but R is capable of producing quite sophisticated variations.)

A First R Session:

Let's make a simple data set (in R parlance, a *vector*) consisting of the numbers 1, 2, and 4, and name it x:

```
> x <- c(1,2,4)
```

The standard assignment operator in R is <-. You can also use =, but this is discouraged, as it does not work in some special situations. Note that there are no fixed types associated with variables. Here, we've assigned a vector to x, but later we might assign something of a different type to it. We'll look at vectors and the other types in Section 1.4.
The c stands for *concatenate*. Here, we are concatenating the numbers 1, 2, and 4. More precisely, we are concatenating three one-element vectors that consist of those numbers. This is because any number is also considered to be a one-element vector.Now we can also do the following:

```
> q <- c(x,x,8)
```

which sets q to (1,2,4,1,2,4,8) (yes, including the duplicates).Now let's confirm that the data is really in x. To print the vector to the screen, simply type its name. If you type any variable name (or, more generally,any expression) while in interactive mode, R will print out the value of that variable (or expression). Programmers familiar with other languages such as Python will find this feature familiar. For our example, enter this:

```
> x
[1] 1 2 4
```

Yep, sure enough, x consists of the numbers 1, 2, and 4. Individual elements of a vector are accessed via [ ]. Here's how we can print out the third element of x:

```
> x[3]
[1] 4
```

As in other languages, the selector (here, 3) is called the *index* or *subscript*. Those familiar with ALGOL-family languages, such as C and C++, should note that elements of R vectors are indexed starting from 1, not 0. *Subsetting* is a very important operation on vectors. Here's an example:

```
> x <- c(1,2,4)
> x[2:3]
```

[1The expression x[2:3] refers to the subvector of x consisting of elements 2 through 3, which are 2 and 4 here. We can easily find the mean and standard deviation of our data set, as follows:

```
> mean(x)
[1] 2.333333
> sd(x)
[1] 1.527525
```

This again demonstrates typing an expression at the prompt in order to print it. In the first line, our expression is the function call mean(x). The return value from that call is printed automatically, without requiring a call to R's print() function.

If we want to save the computed mean in a variable instead of just printing it to the screen, we could execute this code:

```
> y <- mean(x)
```

Again, let's confirm that y really does contain the mean of x:

```
> y
[1] 2.333333
```

As noted earlier, we use # to write comments, like this:

```
> y # print out y
[1] 2.333333
```

Comments are especially valuable for documenting program code, but they are useful in interactive sessions, too, since R records the command history (as discussed in Section 1.6). If you save your session and resume it later, the comments can help you remember what you were doing.Finally, let's do something with one of R's internal data sets (these are
used for demos). You can get a list of these data sets by typing the following:

```
> data()
```

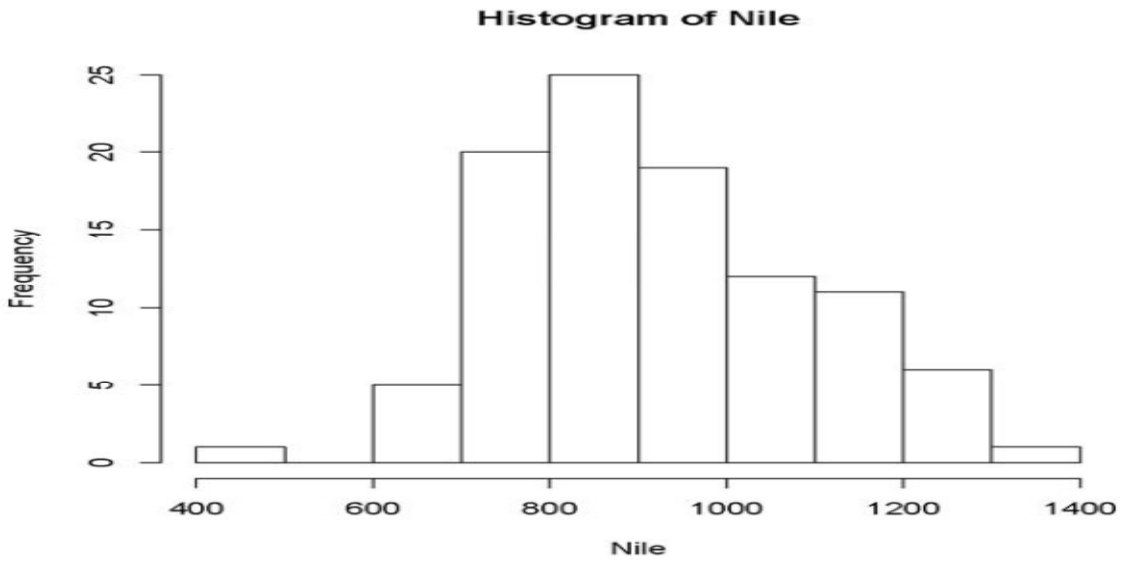
One of the data sets is called Nile and contains data on the flow of the Nile River. Let's find the mean and standard deviation of this data set:

```
> mean(Nile)
[1] 919.35
> sd(Nile)
[1] 169.2275] 2 4
```

We can also plot a histogram of the data:

```
> hist(Nile)
```

A window pops up with the histogram in it, as shown in Figure 1-1. This graph is bare-bones simple, but R has all kinds of optional bells and whistles for plotting. For instance, you can change the number of bins by specifying the breaks variable. The call hist(z,breaks=12) would draw a histogram of the data set z with 12 bins. You can also create nicer labels, make use of color, and make many other changes to create a more informative and eyeappealing graph. When you become more familiar with R, you'll be able to construct complex, rich color graphics of striking beauty.



Well, that's the end of our first, five-minute introduction to R. Quit R by calling the q() function (or alternatively by pressing CTRL-D in Linux or CMD-D on a Mac):

```
> q()
Save workspace image? [y/n/c]: n
```

That last prompt asks whether you want to save your variables so that you can resume work later. If you answer y, then all those objects will be loaded automatically the next time you run R. This is a very important feature, especially when working with large or numerous data sets. Answering y here also saves the session's command history. We'll talk more about saving your workspace.

Introduction to Functions :

As in most programming languages, the heart of R programming consists of writing *functions*. A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

As a simple introduction, let's define a function named oddcount(), whose purpose is to count the odd numbers in a vector of integers. Normally, we would compose the function code using a text editor and save it in a file, but in this quick-and-dirty example, we'll enter it line by line in R's interactive mode. We'll then call the function on a couple of test cases.

```
# counts the number of odd integers in x
> oddcount <- function(x) {
+ k <- 0 # assign 0 to k
+ for (n in x) {
+ if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
+ }
```

```
+ return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

First, we told R that we wanted to define a function named oddcount with one argument, x. The left brace demarcates the start of the body of the function. We wrote one R statement per line.

Until the body of the function is finished, R reminds you that you're still in the definition by using + as its prompt, instead of the usual >. (Actually,+ is a line-continuation character, not a prompt for a new input.) R resumes the > prompt after you finally enter a right brace to conclude the function body.

After defining the function, we evaluated two calls to oddcount(). Since there are three odd numbers in the vector (1,3,5), the call oddcount(c(1,3,5)) returns the value 3. There are four odd numbers in (1,2,3,7,9), so the second call returns 4.

Notice that the modulo operator for remainder arithmetic is %% in R, as indicated by the comment. For example, 38 divided by 7 leaves a remainder of 3:

```
> 38 %% 7
[1] 3
```

For instance, let's see what happens with the following code:

```
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

First, it sets n to x[1], and then it tests that value for being odd or even. If the value is odd, which is the case here, the count variable k is incremented.Then n is set to x[2], tested for being odd or even, and so on.
By the way, C/C++ programmers might be tempted to write the preceding loop like this:

```
for (i in 1:length(x)) {
if (x[i] %% 2 == 1) k <- k+1
}
```

Here, length(x) is the number of elements in x. Suppose there are 25 elements. Then 1:length(x) means 1:25, which in turn means 1,2,3,...,25. This code would also work (unless x were to have length 0), but one of the major themes of R programming is to avoid loops if possible; if not, keep loops simple. Look again at our original formulation:

```
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

It's simpler and cleaner, as we do not need to resort to using the length() function and array indexing.

At the end of the code, we use the return statement:

```
return(k)
```

This has the function return the computed value of k to the code that called it. However, simply writing the following also works:

```
K
```

R functions will return the last value computed if there is no explicit return() call. However, this approach must be used with care,
In programming language terminology, x is the *formal argument* (or *formal parameter*) of the function oddcount(). In the first function call in the preceding example, c(1,3,5) is referred to as the *actual argument*. These terms allude to the fact that x in the function definition is just a placeholder,whereas c(1,3,5) is the value actually used in the computation. Similarly, in
the second function call, c(1,2,3,7,9) is the actual argument.

Variable Scope

A variable that is visible only within a function body is said to be *local* to that function. In oddcount(), k and n are local variables. They disappear after the function returns:

```
> oddcount(c(1,2,3,7,9))
[1] 4
> n
Error: object 'n' not found
```

It's very important to note that the formal parameters in an R function are local variables. Suppose we make the following function call:

```
> z <- c(2,6,7)
> oddcount(z)
```

Now suppose that the code of oddcount() changes x. Then z would *not* change. After the call to oddcount(), z would have the same value as before. To evaluate a function call, R copies each actual argument to the corresponding local parameter variable, and changes to that variable are not visible outside the function.

Variables created outside functions are *global* and are available within functions as well. Here's an example
:

```
> f <- function(x) return(x+y)
> y <- 3
> f(5)
[1] 8
```

Here y is a global variable.
A global variable can be written to from within a function by using R's *superassignment operator*, <<-.

Default Arguments

R also makes frequent use of *default arguments*. Consider a function definition like this:

```
> g <- function(x,y=2,z=T) { ... }
```

Here y will be initialized to 2 if the programmer does not specify y in the call.Similarly, z will have the default value TRUE.
Now consider this call:

```
> g(12,z=FALSE)
```

Here, the value 12 is the actual argument for x, and we accept the default value of 2 for y, but we override the default for z, setting its value to FALSE.The preceding example also demonstrates that, like many programming languages, R has a *Boolean* type; that is, it has the logical values TRUE and FALSE.

Preview of Some Important R Data Structures

R has a variety of data structures. Here, we will sketch some of the most frequently used structures to give you an overview of R before we dive into the details. This way, you can at least get started with some meaningful examples,
even if the full story behind them must wait.

Vectors, the R Workhorse

The vector type is really the heart of R. It's hard to imagine R code, or even an interactive R session, that doesn't involve vectors. The elements of a vector must all have the same *mode*, or data type. You can have a vector consisting of three character strings (of mode character) or three integer elements (of mode integer), but not a vector with one integer
element and two character string elements..

Scalars

Scalars, or individual numbers, do not really exist in R. As mentioned earlier, what appear to be individual numbers are actually one-element vectors.

Consider the following:

```
> x <- 8
> x
[1] 8
```

Recall that the [1] here signifies that the following row of numbers begins with element 1 of a vector—in this case, x[1]. So you can see that R was indeed treating x as a vector, albeit a vector with just one element.

Character Strings

Character strings are actually single-element vectors of mode character,(rather than mode numeric):

```
> x <- c(5,12,13)
> x
[1] 5 12 13
> length(x)
[1] 3
> mode(x)
[1] "numeric"
> y <- "abc"
> y
[1] "abc"
> length(y)
[1] 1
> mode(y)
[1] "character"
> z <- c("abc","29 88")
> length(z)
[1] 2
> mode(z)
[1] "character"
```

In the first example, we create a vector x of numbers, thus of mode numeric. Then we create two vectors of mode character: y is a one-element (that is,one-string) vector, and z consists of two strings.R has various string-manipulation functions. Many deal with putting strings together or taking them apart, such as the two shown here:

```
> u <- paste("abc","de","f") # concatenate the strings
> u
[1] "abc de f"
> v <- strsplit(u," ") # split the string according to blanks
> v
[[1]]
[1] "abc" "de" "f"
```

Matrices

An R matrix corresponds to the mathematical concept of the same name: a rectangular array of numbers. Technically, a matrix is a vector, but with two additional attributes: the number of rows and the number of columns. Here is some sample matrix code:

```
> m <- rbind(c(1,4),c(2,2))
> m
[,1] [,2]
[1,] 1 4
[2,] 2 2
> m %*% c(1,1)
[,1]
[1,] 5
[2,] 4
```

First, we use the rbind() (for *row bind*) function to build a matrix from two vectors that will serve as its rows, storing the result in m. (A corresponding function, cbind(), combines several columns into a matrix.) Then entering the variable name alone, which we know will print the variable, confirms that the intended matrix was produced. Finally, we compute the matrix product of the vector (1,1) and m. The matrix-multiplication operator, which you may know from linear algebra courses, is %*% in R.

Matrices are indexed using double subscripting, much as in C/C++, although subscripts start at 1 instead of 0.

```
> m[1,2]
[1] 4
> m[2,2]
[1] 2
```

An extremely useful feature of R is that you can extract submatrices from a matrix, much as you extract subvectors from vectors. Here's an example:

```
> m[1,] # row 1
[1] 1 4
> m[,2] # column 2
[1] 4 2
```

Lists

Like an R vector, an R list is a container for values, but its contents can be items of different data types. (C/C++ programmers will note the analogy to a C struct.) List elements are accessed using two-part names, which are indicated with the dollar sign $ in R. Here's a quick example:

```
> x <- list(u=2, v="abc")
> x
```

```
$u
[1] 2
$v
[1] "abc"
> x$u
[1] 2
```

The expression x$u refers to the u component in the list x. The latter contains one other component, denoted by v.
A common use of lists is to combine multiple values into a single package that can be returned by a function. This is especially useful for statistical functions, which can have elaborate results. As an example, consider R's basic histogram function, hist(), introduced in Section 1.2. We called the function on R's built-in Nile River data set:

```
> hist(Nile)
```

This produced a graph, but hist() also returns a value, which we can save:

```
> hn <- hist(Nile)
```

What's in hn? Let's take a look:

```
> print(hn)
$breaks

[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

$counts

[1] 1 0 5 20 25 19 12 11 6 1

$intensities

[1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
[6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$density

[1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
[6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$mids

[1] 450 550 650 750 850 950 1050 1150 1250 1350

$xname

[1] "Nile"

$equidist
[1] TRUE
```

---

attr(,"class")

[1] "histogram"
Don't try to understand all of that right away. For now, the point is that, besides making a graph, hist() returns a list with a number of components. Here, these components describe the characteristics of the histogram. For instance, the breaks component tells us where the bins in the histogram start and end, and the counts component is the numbers of observations in each bin.

The designers of R decided to package all of the information returned by hist() into an R list, which can be accessed and manipulated by further R commands via the dollar sign. Remember that we could also print hn simply by typing its name:
> hn
But a more compact alternative for printing lists like this is str():
> str(hn)
List of 7
$ breaks : num [1:11] 400 500 600 700 800 900 1000 1100 1200 1300 ...
$ counts : int [1:10] 1 0 5 20 25 19 12 11 6 1
$ intensities: num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ density : num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ mids : num [1:10] 450 550 650 750 850 950 1050 1150 1250 1350
$ xname : chr "Nile"
$ equidist : logi TRUE
- attr(*, "class")= chr "histogram"
Here str stands for *structure*. This function shows the internal structure of any R object, not just lists.

Data Frames :
A typical data set contains data of different modes. In an employee data set, for example, we might have character string data, such as employee names, and numeric data, such as salaries. So, although a data set of (say) 50 employees with 4 variables per worker has the look and feel of a 50-by-4 matrix, it does not qualify as such in R, because it mixes types.
Instead of a matrix, we use a *data frame*. A data frame in R is a list, with each component of the list being a vector corresponding to a column in our "matrix" of data. Indeed, you can create data frames in just this way:

> d <- data.frame(list(kids=c("Jack","Jill"),ages=c(12,10)))
> d
kids ages
12 Jill 10
> d$ages
[1] 12 10
Typically, though, data frames are created by reading in a data set from
a file or database. Jack 12

Classes :
R is an object-oriented language. *Objects* are instances of *classes*. Classes are a bit more abstract than the data types you've met so far. Here, we'll look briefly at the concept using R's S3 classes. (The name stems from their use in the old S language, version 3, which was the inspiration for R.) Most of R is based on these classes, and they are exceedingly simple. Their instances are simply R lists but with an extra attribute: the class name.For example, we noted earlier that the (nongraphical) output of the hist() histogram function is a list with various components, such as break and count components. There was also an *attribute*, which specified the class of the list, namely histogram.

> print(hn)
$breaks
[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

```
$counts
[1] 1 0 5 20 25 19 12 11 6 1
...
...
attr(,"class")
[1] "histogram"
```

A generic function stands for a family of functions, all serving a similar purpose but each appropriate to a specific class. A commonly used generic function is summary(). An R user who wants to use a statistical function, like hist(), but is unsure of how to deal with its output (which can be voluminous), can simply call summary() on the output, which is not just a list but an instance of an S3 class.

The plot() function is another generic function. You can use plot() on just about any R object. R will find an appropriate plotting function based on the object's class. Classes are used to organize objects. Together with generic functions, they allow flexible code to be developed for handling a variety of different but related tasks. Chapter 9 covers classes in depth.

UNIT-II
R Programming Structures

Control Statements

Control statements in R look very similar to those of the ALGOL-descendant family languages mentioned above. Here, we'll look at loops and if-else statements

Loops

In Section 1.3, we defined the oddcount() function. In that function, the following line should have been instantly recognized by Python programmers:

for (n in x) {

It means that there will be one iteration of the loop for each component of the vector x, with n taking on the values of those components—in the first iteration, n = x[1]; in the second iteration, n = x[2]; and so on. For example, the following code uses this structure to output the square of every element in a vector:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

C-style looping with while and repeat is also available, complete with break, a statement that causes control to leave the loop. Here is an example that uses all three:

```
> i <- 1
> while (i <= 10) i <- i+4
> i
[1] 13
>
> i <- 1
> while(TRUE) { # similar loop to above
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
>
> i <- 1
> repeat { # again similar
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
```


In the first code snippet, the variable i took on the values 1, 5, 9, and 13 as the loop went through its iterations. In that last case, the condition i <= 10 failed, so the break took hold and we left the loop.

This code shows three different ways of accomplishing the same thing,with break playing a key role in the second and third ways. Note that repeat has no Boolean exit condition. You must use break (or something like return()). Of course, break can be used with for loops, too.Another useful statement is next, which instructs the

interpreter to skip the remainder of the current iteration of the loop and proceed directly to the next one. This provides a way to avoid using complexly nested if-thenelse constructs, which can make the code confusing. Let's take a look at an example that uses next.

```
1 sim <- function(nreps) {
2 commdata <- list()
3 commdata$countabsamecomm <- 0
4 for (rep in 1:nreps) {
5 commdata$whosleft <- 1:20
6 commdata$numabchosen <- 0
7 commdata <- choosecomm(commdata,5)
8 if (commdata$numabchosen > 0) next
9 commdata <- choosecomm(commdata,4)
10 if (commdata$numabchosen > 0) next
11 commdata <- choosecomm(commdata,3)
12 }
13 print(commdata$countabsamecomm/nreps)
14 }
```

There are next statements in lines 8 and 10. Let's see how they work and how they improve on the alternatives. The two next statements occur within the loop that starts at line 4. Thus, when the if condition holds in line 8, lines 9 through 11 will be skipped, and control will transfer to line 4.

The situation in line 10 is similar. Without using next, we would need to resort to nested if statements,something like these:

```
1 sim <- function(nreps) {
2 commdata <- list()
3 commdata$countabsamecomm <- 0
4 for (rep in 1:nreps) {
5 commdata$whosleft <- 1:20
6 commdata$numabchosen <- 0
7 commdata <- choosecomm(commdata,5)
8 if (commdata$numabchosen == 0) {
9 commdata <- choosecomm(commdata,4)
10 if (commdata$numabchosen == 0)
11 commdata <- choosecomm(commdata,3)
12 }
13 }
print(commdata$countabsamecomm/nreps)
15 }
```

Because this simple example has just two levels, it's not too bad. However, nested if statements can become confusing when you have more levels. The for construct works on any vector, regardless of mode. You can loop over a vector of filenames, for instance. Say we have a file named *file1* with the following contents:

```
1
2
3
4
5
6
```

We also have a file named *file2* with these contents:

```
5
```

12
13
The following loop reads and prints each of these files. We use the scan() function here to read in a file of numbers and store those values in a vector.

```
> for (fn in c("file1","file2")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
[1] 5 12 13
```

So, fn is first set to *file1*, and the file of that name is read in and printed out. Then the same thing happens for *file2*.

7.1.2 Looping Over Nonvector Sets

R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

• Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.

• Use get(). As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

Let's look at an example of using get(). Say we have two matrices, u and v, containing statistical data, and we wish to apply R's linear regression function lm() to each of them.

```
> u
[,1] [,2]
[1,] 1 1
[2,] 2 2
[3,] 3 4
> v
[,1] [,2]
[1,] 8 15
[2,] 12 10
[3,] 20 2
> for (m in c("u","v")) {
+ z <- get(m)
+ print(lm(z[,2] ~ z[,1]))
+ }
Call:
lm(formula = z[, 2] ~ z[, 1])
Coefficients:
(Intercept) z[, 1]
-0.6667 1.5000
Call:
lm(formula = z[, 2] ~ z[, 1])
Coefficients:
(Intercept) z[, 1]
23.286 -1.071
```

Here, m was first set to u. Then these lines assign the matrix u to z, which allows the call to lm() on u:

```
z <- get(m)
print(lm(z[,2] ~ z[,1]))
```

The same then occurs with v.

if-else

The syntax for if-else looks like this:

```
if (r == 4) {
x <- 1
} else {
x <- 3
y <- 4
}
```

It looks simple, but there is an important subtlety here. The if section consists of just a single statement:

```
x <- 1
```

So, you might guess that the braces around that statement are not necessary. However, they are indeed needed. The right brace before the else is used by the R parser to deduce that this is an if-else rather than just an if. In interactive mode, without braces, the parser would mistakenly think the latter and act accordingly, which is not what we want.

An if-else statement works as a function call, and as such, it returns the last value assigned.

```
v <- if (cond) expression1 else expression2
```

This will set v to the result of expression1 or expression2, depending on whether cond is true. You can use this fact to compact your code. Here's a simple example:

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2
> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4
```

Without taking this tack, the code

```
y <- if(x == 2) x else x+1
```

would instead consist of the somewhat more cluttered

```
if(x == 2) y <- x else y <- x+1
```

In more complex examples, expression1 and/or expression2 could be function calls. On the other hand, you probably should not let compactness take priority over clarity. When working with vectors, use the ifelse() function

Arithmetic and Boolean Operators and Values

Table 7-1 lists the basic operators.

Table 7-1: Basic R Operators :**Operation Description**

x + y Addition
x - y Subtraction
x * y Multiplication
x / y Division
x ^ y Exponentiation
x %% y Modular arithmetic
x %/% y Integer division
x == y Test for equality

x <= y Test for less than or equal to
x >= y Test for greater than or equal to
x && y Boolean AND for scalars
x || y Boolean OR for scalars
x & y Boolean AND for vectors (vector x,y,result)
x | y Boolean OR for vectors (vector x,y,result)
!x Boolean negation
Though R ostensibly has no scalar types, with scalars being treated as one-element vectors, we see the exception in Table 7-1: There are different Boolean operators for the scalar and vector cases. This may seem odd, but a simple example will demonstrate the need for such a distinction.

```
> x
[1] TRUE FALSE TRUE
> y
[1] TRUE TRUE FALSE
> x & y
[1] TRUE FALSE FALSE
> x[1] && y[1]
[1] TRUE
> x && y # looks at just the first elements of each vector
[1] TRUE
> if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
> if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
```

the condition has length > 1 and only the first element will be used The central point is that in evaluating an if, we need a single Boolean, not a vector of Booleans, hence the warning seen in the preceding example, as well as the need for having both the & and && operators.

The Boolean values TRUE and FALSE can be abbreviated as T and F (both must be capitalized). These values change to 1 and 0 in arithmetic expressions:

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
> (1 < 2) == 1
[1] TRUE
```

In the second computation, for instance, the comparison 1 < 2 returns TRUE, and 3 < 4 yields TRUE as well. Both values are treated as 1 values, so the product is 1.

On the surface, R functions look similar to those of C, Java, and so on. However, they have much more of a functional programming flavor, which has direct implications for the R programmer.

Default Values for Arguments :

In Section 5.1.2, we read in a data set from a file named exams:

```
> testscores <- read.table("exams",header=TRUE)
```

The argument header=TRUE tells R that we have a header line, so R should not count that first line in the file as data.
This is an example of the use of *named arguments*. Here are the first few lines of the function:

```
> read.table
function (file, header = FALSE, sep = "", quote = "\"", dec = ".",
row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#", allowEscapes = FALSE, flush = FALSE,
stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")
{
if (is.character(file)) {
file <- file(file, "r")
on.exit(close(file))
...
...
```

The second formal argument is named header. The = FALSE field means that this argument is optional, and if we don't specify it, the default value will be FALSE. If we don't want the default value, we must name the argument in our call:

```
> testscores <- read.table("exams",header=TRUE)
```

Hence the terminology *named argument*.

Return Values
The return value of a function can be any R object. Although the return value is often a list, it could even be another function. You can transmit a value back to the caller by explicitly calling return(). Without this call, the value of the last executed statement will be returned by default.


```
> oddcount
function(x) {
k <- 0 # assign 0 to k
for (n in x) {
if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
}
return(k)
}
```

This function returns the count of odd numbers in the argument. We could slightly simplify the code by eliminating the call to return(). To do this, we evaluate the expression to be returned, k, as our last statement in the code:

```
oddcount <- function(x) {
k <- 0
pagebreak
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
```

```
k
}
```
On the other hand, consider this code:
```
oddcount <- function(x) {
k <- 0

for (n in x) {
if (n %% 2 == 1) k <- k+1
}
}
```
It wouldn't work, for a rather subtle reason: The last executed statement here is the call to for(), which returns the value NULL (and does so, in R parlance, *invisibly*, meaning that it is discarded if not stored by assignment). Thus, there would be no return value at all.

7.4.1 Deciding Whether to Explicitly Call return()

The prevailing R idiom is to avoid explicit calls to return(). One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible,
so this might not be the most compelling reason to refrain from using return(). But it usually isn't needed nonetheless.

Consider our second example from the preceding section:
```
oddcount <- function(x) {
k <- 0
for (n in x) {
if (n %% 2 == 1) k <- k+1
}
k
}
```
Here, we simply ended with a statement listing the expression to be returned—in this case, k. A call to return() wasn't necessary. Code in this book usually does include a call to return(), for clarity for beginners, but it is customary to omit it.
Good software design, however, should be mean that you can glance through a function's code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return. (You can still omit a return() call at the end of the function if
you wish.)

Returning Complex Objects

Since the return value can be any R object, you can return complex objects. Here is an example of a function being returned:
```
> g
function() {
t <- function(x) return(x^2)
return(t)
}
> g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```
If your function has multiple return values, place them in a list or other container.

Functions Are Objects

R functions are *first-class objects* (of the class "function", of course), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

```
> g <- function(x) {
+ return(x+1)
+ }
```

Here, function() is a built-in R function whose job is to create functions! On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function we're creating—here, just x—and the second is the body of that function—here, just the single statement return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g.

By the way, even the "{" is a function, as you can verify by typing this:

```
> ?"{"
```

Its job is the make a single unit of what could be several statements.

These two arguments to function() can later be accessed via the R functions formals() and body(), as follows:

```
> formals(g)
$x
> body(g)
{
return(x + 1)
}
```

Recall that when using R in interactive mode, simply typing the name of an object results in printing that object to the screen. Functions are no exception, since they are objects just like anything else.

```
> g
function(x) {
return(x+1)
```

This is handy if you're using a function that you wrote but which you've forgotten the details of. Printing out a function is also useful if you are not quite sure what an R library function does. By looking at the code, you may understand it better. For example, if you are not sure as to the exact behavior of the graphics function abline(), you could browse through its code to better understand how to use it.

```
> abline
function (a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
coef = NULL, untf = FALSE, ...)
{
int_abline <- function(a, b, h, v, untf, col = par("col"),
lty = par("lty"), lwd = par("lwd"), ...) .Internal(abline(a,
b, h, v, untf, col, lty, lwd, ...))
if (!is.null(reg)) {
if (!is.null(a))
warning("'a' is overridden by 'reg'")
a <- reg
}
if (is.object(a) || is.list(a)) {
p <- length(coefa <- as.vector(coef(a)))
```

...
...
If you wish to view a lengthy function in this way, run it through page():

```
> page(abline)
```

Note, though, that some of R's most fundamental built-in functions are written directly in C, and thus they are not viewable in this manner. Here's an example:

```
> sum
function (..., na.rm = FALSE) .Primitive("sum")
```

Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1
> f(3,2)
[1] 5
> f <- f2}

> f(3,2)
[1] 1
> g <- function(h,a,b) h(a,b)
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

And since functions are objects, you can loop through a list consisting of several functions. This would be useful, for instance, if you wished to write a loop to plot a number of functions on the same graph, as follows:

```
> g1 <- function(x) return(sin(x))
> g2 <- function(x) return(sqrt(x^2+1))
> g3 <- function(x) return(2*x-1)
> plot(c(0,1),c(-1,1.5)) # prepare the graph, specifying X and Y ranges
> for (f in c(g1,g2,g3)) plot(f,0,1,add=T) # add plot to existing graph
```

The functions formals() and body() can even be used as replacement functions. We'll discuss replacement functions in Section 7.10, but for now, consider how you could change the body of a function by assignment:

```
> g <- function(h,a,b) h(a,b)
> body(g) <- quote(2*x + 3)
> g
function (x)
2 * x + 3
> g(3)
[1] 9
```

The reason quote() was needed is that technically, the body of a function has the class "call", which is the class produced by quote(). Without the call to quote(), R would try to evaluate the quantity 2*x+3. So if x had been defined and equal to 3, for example, we would assign 9 to the body of g(), certainly not what we want. By the way, since * and + are functions (as discussed in Section 2.4.1), as a language object, 2*x+3 is indeed a call—in fact, it is one function call nested within another.

Environment and Scope Issues

A function—formally referred to as a *closure* in the R documentation— consists not only of its arguments and body but also of its *environment*. The latter is made up of the collection of objects present at the time the function is created. An understanding of how environments work in R is essential for writing effective R functions.

The Top-Level Environment

Consider this example:

```
> w <- 12
> f <- function(y) {
+ d <- 8
+ h <- function() {
+ return(d*(w+y))
+ }
+ return(h())
+ }
> environment(f)
<environment: R_GlobalEnv>
```

Here, the function f() is created at the *top level*—that is, at the interpreter command prompt—and thus has the top-level environment, which in R output is referred to as R_GlobalEnv but which confusingly you refer to in R code as .GlobalEnv. If you run an R program as a batch file, that is considered top level, too.

The function ls() lists the objects of an environment. If you call it at the top level, you get the top-level environment. Let's try it with our example

code:

```
> ls()
[1] "f" "w"
```

As you can see, the top-level environment here includes the variable w, which is actually used within f(). Note that f() is here too, as functions are indeed objects and we did create it at the top level. At levels other than the top, ls() works a little differently,

You get a bit more information from ls.str():

```
> ls.str()
f : function (y)
w : num 12
```

Next, we'll look at how w and other variables come into play within f().

The Scope Hierarchy

Let's first get an intuitive overview of how scope works in R and then relate it to environments. If we were working with the C language (as usual, background in C is not assumed), we would say that the variable w in the previous section is *global* to f(), while d is *local* to f(). Things are similar in R, but R is more hierarchical.

In C, we would not have functions defined within functions, as we have with h() inside f() in our example. Yet, since functions are objects, it is possible—and sometimes desirable from the point of view of the encapsulation goal of object-oriented programming—to define a function within a function; we are simply creating an object, which we can do anywhere.

Here, we have h() being local to f(), just like d. In such a situation, it makes sense for scope to be hierarchical. Thus, R is set up so that d, which is local to f(), is in turn global to h(). The same is true for y, as arguments are considered locals in R.

Similarly, the hierarchical nature of scope implies that since w is global to f(), it is global to h() as well. Indeed, we do use w within h(). In terms of environments then, h()'s environment consists of whatever objects are defined at the time h() comes into existence; that is, at the time

that this assignment is executed:

```
h <- function() {
return(d*(w+y))
}
```

(If f() is called multiple times, h() will come into existence multiple times, going out of existence each time f() returns.) What, then, will be in h()'s environment? Well, at the time h() is created, there are the objects d and y created within f(), *plus* f()'s environment (w). In other words, if one function is defined within another, then that inner function's environment consists of the environment of the outer one, plus whatever locals have been created so far within the outer one. With multiple nesting of functions, you have a nested sequence of larger and larger environments,with the "root" consisting of the top-level objects.

Let's try out the code:

```
> f(2)
[1] 112
```

What happened? The call f(2) resulted in setting the local variable d to 8, followed by the call h(). The latter evaluated d*(w+y)—that is, 8*(12+2)—giving us 112.

Note carefully the role of w. The R interpreter found that there was no local variable of that name, so it ascended to the next higher level—in this case, the top level—where it found a variable w with value 12.

Keep in mind that h() is local to f() and invisible at the top level.

```
> h
Error: object 'h' not found
```

It's possible (though not desirable) to deliberately allow name conflicts in this hierarchy. In our example, for instance, we could have a local variable d within h(), conflicting with the one in f(). In such a situation, the innermost environment is used first. In this case, a reference to d within h() would refer to h()'s d, not f()'s.

Environments created by inheritance in this manner are generally referred to by their memory locations. Here is what happened after adding a print statement to f() (using edit(), not shown here) and then running the code:

```
> f
function(y) {
d <- 8
h <- function() {
return(d*(w+y))
}
print(environment(h))
return(h())
}
> f(2)
<environment: 0x875753c>
[1] 112
```

Compare all this to the situation in which the functions are not nested:

```
> f
function(y) {
d <- 8
return(h())
}
> h
function() {
return(d*(w+y))
}
```

The result is as follows:

```
> f(5)
Error in h() : object 'd' not found
```

This does not work, as d is no longer in the environment of h(), because h() is defined at the top level. Thus, an error is generated. Worse, if by happenstance there had been some unrelated variable d in the top-level environment, we would not get an error message but instead would have incorrect results.

You might wonder why R didn't complain about the lack of y in the alternate definition of h() in the preceding example. As mentioned earlier, R doesn't evaluate a variable until it needs it under a policy called lazy evaluation. In this case, R had already encountered an error with d and thus never got to the point where it would try to evaluate y.

The fix is to pass d and y as arguments:

```
> f
function(y) {
d <- 8
return(h(d,y))
}
> h
function(dee,yyy) {
return(dee*(w+yyy))
}
> f(2)
[1] 88
```

Okay, let's look at one last variation:

```
> f
function(y,ftn) {
d <- 8
print(environment(ftn))
return(ftn(d,y))
}
> h
function(dee,yyy) {
return(dee*(w+yyy))
}
> w <- 12
> f(3,h)
<environment: R_GlobalEnv>
[1] 120
```


When f() executed, the formal argument ftn was matched by the actual argument h. Since arguments are treated as locals, you might guess that ftn could have a different environment than top level. But as discussed, a closure includes environment, and thus ftn has h's environment.

More on ls() :

Without arguments, a call to ls() from within a function returns the names of the current local variables (including arguments). With the envir argument, it will print the names of the locals of any frame in the call chain

Here's an example:

```
> f
function(y) {
d <- 8
return(h(d,y))
```

```
}
> h
function(dee,yyy) {
print(ls())
print(ls(envir=parent.frame(n=1)))
return(dee*(w+yyy))
}
> f(2)
[1] "dee" "yyy"
[1] "d" "y"
[1] 112
```

With parent.frame(), the argument n specifies how many frames to go up in the call chain. Here, we were in the midst of executing h(), which had been called from f(), so specifying n = 1 gives us f()'s frame, and thus we get its locals.

7.6.4 Functions Have (Almost) No Side Effects

Yet another influence of the functional programming philosophy is that functions do not change nonlocal variables; that is, generally, there are no *side effects*. Roughly speaking, the code in a function has read access to its nonlocal variables, but it does not have write access to them. Our code can appear to reassign those variables, but the action will affect only copies, not the variables themselves. Let's demonstrate this by adding some more code to our previous example.

```
> w <- 12
> f
function(y) {
d <- 8
w <- w + 1
y <- y - 2
print(w)
h <- function() {
return(d*(w+y))
}
return(h())
}

> t <- 4
> f(t)
[1] 13
[1] 120
> w
[1] 12
> t
[1] 4
```

So, w at the top level did not change, even though it appeared to change within f(). Only a local *copy* of w, within f(), changed. Similarly, the top-level variable t didn't change, even though its associated formal argument y did change.

Extended Example: A Function to Display the Contents of a Call Frame

In single-stepping through your code in a debugging setting, you often want to know the values of the local variables in your current function. You may also want to know the values of the locals in the parent function—that is, the one from which the current function was called. Here, we will develop code to display these values, thereby further demonstrating access to the environment hierarchy. (The code is adapted from my edtdbg debugging tool in R's CRAN code repository.)

For example, consider the following code:

```
f <- function() {
a <- 1
return(g(a)+a)
}
g <- function(aa) {
b <- 2
aab <- h(aa+b)
return(aab)
}
h <- function(aaa) {
c <- 3
return(aaa+c)
}
```

When we call f(), it in turn calls g(), which then calls h(). In the debugging setting, say we are currently about to execute the return() within g(). We want to know the values of the local variables of the current function, say the variables aa, b, and aab. And while we're in g(), we also wish to know the values of the locals in f() at the time of the call to g(), as well as the values of the global variables. Our function showframe() will do all this. The showframe() function has one argument, upn, which is the number of frames to go up the call stack. A negative value of the argument signals that we want to view the globals—the top-level variables. Here's the code:

```
# shows the values of the local variables (including arguments) of the
# frame upn frames above the one from which showframe() is called; if
# upn < 0, the globals are shown; function objects are not shown
showframe <- function(upn) {
# determine the proper environment
if (upn < 0) {
env <- .GlobalEnv
} else {
env <- parent.frame(n=upn+1)
}
# get the list of variable names
vars <- ls(envir=env)
# for each variable name, print its value
for (vr in vars) {
vrg <- get(vr,envir=env)
if (!is.function(vrg)) {
cat(vr,":\n",sep="")
print(vrg)
}
}
}
```

Let's try it out. Insert some calls into g():

```
> g
function(aa) {
b <- 2
showframe(0)
showframe(1)
aab <- h(aa+b)
return(aab)
```

```
}
```
Now run it:
```
> f()
aa:
[1] 1
b:
[1] 2
a:
[1] 1
```
To see how this works, we'll first look at the get() function, one of the most useful utilities in R. Its job is quite simple: Given the name of an object, it fetches the object itself. Here's an example:

```
> m <- rbind(1:3,20:22)
> m
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 20 21 22
> get("m")
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 20 21 22
```
This example with m involves the current call frame, but in our showframe() function, we deal with various levels in the environment hierarchy. So, we need to specify the level via the envir argument of get():

```
vrg <- get(vr,envir=env)
```
The level itself is determined largely by calling parent.frame():
```
if (upn < 0) {
env <- .GlobalEnv
} else {
env <- parent.frame(n=upn+1)
}
```
Note that ls() can also be called in the context of a particular level, thus enabling you to determine which variables exist at the level of interest and then inspect them. Here's an example:

```
vars <- ls(envir=env)
for (vr in vars) {
```
This code picks up the names of all the local variables in the given frame and then loops through them, setting things up for get() to do its work.

No Pointers in R

R does not have variables corresponding to *pointers* or *references* like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature
called *reference classes*, which may reduce the difficulty.)

For example, you cannot write a function that directly changes its arguments.
In Python, for instance, you can do this:
```
>>> x = [13,5,12]
>>> x.sort()
>>> x
[5, 12, 13]
```

Here, the value of x, the argument to sort(), changed. By contrast, here's how it works in R:

```
> x <- c(13,5,12)
> sort(x)
[1] 5 12 13
> x
[1] 13 5 12
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
> x
[1] 5 12 13
```

What if our function has several variables of output? A solution is to gather them together into a list, call the function with this list as an argument, have the function return the list, and then reassign to the original list. An example is the following function, which determines the indices of odd and even numbers in a vector of integers:

```
> oddsevens
function(v){
odds <- which(v %% 2 == 1)
evens <- which(v %% 2 == 1)
list(o=odds,e=evens)
}
```

In general, our function f() changes variables x and y. We might store them in a list lxy, which would then be our argument to f(). The code, both called and calling, might have a pattern like this:

```
f <- function(lxxyy) {
...
lxxyy$x <- ...
lxxyy$y <- ...
return(lxxyy)
}

# set x and y
lxy$x <- ...
lxy$y <- ...
lxy <- f(lxy)
# use new x and y
... <- lxy$x
... <- lxy$y
```


However, this may become unwieldy if your function will change many variables. It can be especially awkward if your variables, say x and y in the example, are themselves lists, resulting in a return value consisting of lists within a list. This can still be handled, but it makes the code more syntactically complex and harder to read. Alternatives include the use of global variables, which we will look at in Section 7.8.4, and the new R reference classes mentioned earlier.

Another class of applications in which lack of pointers causes difficulties is that of treelike data structures. C code normally makes heavy use of pointers for these kinds of structures. One solution for R is to revert to what was done in the "good old days" before C, when programmers formed their own "pointers" as vector indices.

Writing Upstairs

As mentioned earlier, code that exists at a certain level of the environment hierarchy has at least read access to all the variables at the levels above it. On the other hand, direct write access to variables at higher levels via the standard <- operator is not possible.

If you wish to write to a global variable—or more generally, to any variable higher in the environment hierarchy than the level at which your write statement exists—you can use the superassignment operator, <<-, or the assign() function. Let's discuss the superassignment operator first.

7.8.1 Writing to Nonlocals with the Superassignment Operator

Consider the following code:

```
> two <- function(u) {
+ u <<- 2*u
+ z <- 2*z
+ }
> x <- 1
> z <- 3
> u
Error: object "u" not found
> two(x)
> x
[1] 1
> z
[1] 3
> u
[1] 2
```

Let's look at the impact (or not) on the three top-level variables x, z, and u:

• x: Even though x was the actual argument to two() in the example, it retained the value 1 after the call. This is because its value 1 was copied to the formal argument u, which is treated as a local variable within the function. Thus, when u changed, x did not change with it.

• z: The two z values are entirely unrelated to each other—one is top level, and the other is local to two(). The change in the local variable has no effect on the global variable. Of course, having two variables with the same name is probably not good programming practice.

• u: The u value did not even exist as a top-level variable prior to our calling two(), hence the "not found" error message. However, it was created as a top-level variable by the superassignment operator within two(), as confirmed after the call.

Though <<- is typically used to write to top-level variables, as in our example, technically, it does something a bit different. Use of this operator to write to a variable w will result in a search up the environment hierarchy, stopping at the first level at which a variable of that name is encountered. If none is found, then the level selected will be global. Look what happens in this little example:

```
> f
function() {
inc <- function() {x <<- x + 1}
x <- 3
inc()
return(x)
}
> f()
[1] 4
> x
Error: object 'x' not found
```

Here, inc() is defined within f(). When inc() is executing, and the R interpreter sees a superassignment to x, it starts going up the hierarchy. At the first level up—the environment within f()—it does find an x, and so that x is the one that is written to, not x at the top level.

Writing to Nonlocals with assign()

You can also use the assign() function to write to upper-level variables. Here's an altered version of the previous example:

```
> two
function(u) {
assign("u",2*u,pos=.GlobalEnv)
z <- 2*z
}
> two(x)
> x
[1] 1
> u
[1] 2
```

Here, we replaced the superassignment operator with a call to assign(). That call instructs R to assign the value 2*u (this is the local u) to a variable u further up the call stack, specifically in the top-level environment. In this case, that environment is only one call level higher, but if we had a chain of calls, it could be much further up.

The fact that you reference variables using character strings in assign() can come in handy. Recall the example in Chapter 5 concerning analysis of hiring patterns of various large corporations. We wanted to form a subdata frame for each firm, extracted from the overall data frame, all2006. For instance, consider this call:

```
makecorpdfs(c("MICROSOFT CORPORATION","ms","INTEL CORPORATION","intel","
SUN MICROSYSTEMS, INC.","sun","GOOGLE INC.","google")
```

This would first extract all Microsoft records from the overall data frame, naming the resulting subdata frame ms2006. It would then create intel2006 for Intel, and so on. Here is the code (changed to function form, for clarity):

```
makecorpdfs <- function(corplist) {
for (i in 1:(length(corplist)/2)) {
corp <- corplist[2*i-1]
newdtf <- paste(corplist[2*i],"2006",sep="")
assign(newdtf,makecorp(corp),pos=.GlobalEnv)
}
}
```

In the iteration i = 1, the code uses paste() to splice together the strings "ms" and "2006", resulting in "ms2006", the desired name.

When Should You Use Global Variables?

Use of global variables is a subject of controversy in the programming community. Here, the term *global variable*, or just *global*, will be used to include any variable located higher in the environment hierarchy than the level of the given code of interest. The use of global variables in R is more common than you may have guessed. You might be surprised to learn that R itself makes very substantial use of globals internally, both in its C code and in its R routines. The superassignment operator <<-, for instance, is used in many of the R library functions (albeit typically in writing to a variable just one level up in the environment hierarchy). *Threaded* code and *GPU* code, which are used for writing fast programs (as described in Chapter 16), tend to make heavy use of global variables, which provide the main avenue of communication between parallel actors.

Now, to make our discussion concrete, let's return to the earlier example from Section 7.7:

```
f <- function(lxxyy) { # lxxyy is a list containing x and y
...
lxxyy$x <- ...
lxxyy$y <- ...
return(lxxyy)
}
R
# set x and y
lxy$x <- ...
lxy$y <- ...
lxy <- f(lxy)
# use new x and y
... <- lxy$x
... <- lxy$y
```

As noted earlier, this code might be a bit unwieldy, especially if x and y are themselves lists.

By contrast, here is an alternate pattern that uses globals:

```
f <- function() {
...
x <<- ...
y <<- ...
}
# set x and y
x <- ...
y <- ...
f() # x and y are changed in here
# use new x and y
... <- x
... <- y
```

We chose to use globals, rather than to return lists, in the DES code earlier in this chapter. Let's explore that example further. We had two global variables (both lists, encapsulating various information): sim, associated with the library code, and mm1glbls, associated with our M/M/1 application code. Let's consider sim first. Even many programmers who have reservations about using globals agree that such variables may be justified if they are truly global, in the sense that they are used broadly in the program. This is the case for sim in our DES example. It is used both in the library code (in schedevnt(), getnextevnt(), and dosim()) and in in our M/M/1 application code (in mm1reactevnt()).

So, using sim as a global seems justified. Nevertheless, if we were bound and determined to avoid using globals, we could have placed sim as a local within dosim(). This function would then pass sim as an argument to all of the functions mentioned in the previous paragraph (schedevnt(), getnextevnt(), and so on), and each of these functions would return the modified sim.

for example, would change from this: reactevnt(head) to this: sim <- reactevnt(head)

We would then need to add a line like the following to our applicationspecific function mm1reactevnt():

```
return(sim)
```

We could do something similar with mm1glbls, placing a variable called, say, appvars as a local within dosim(). However, if we did this with sim as well, we would need to place them together in a list so that both would be returned, as in our earlier example function f().

On the other hand, critics of the use of global variables counter that the simplicity of the code comes at a price. They worry that it may be difficult during debugging to track down locations at which a global variable changes value, since such a change could occur anywhere in the program.
Another concern raised by critics involves situations in which a function is called in several unrelated parts of the overall program using different values. For example, consider using our example f() function in different parts of our program, each call with its own values of x and y, rather than just a single value of each, as assumed earlier. This could be solved by setting up vectors of x and y values, with one element for each instance of f() in your program.
Within dosim(), the line
sim <<- list()
would be replaced by
assign("simenv",new.env(),envir=.GlobalEnv)
This would create a new environment, pointed to by simenv at the top level. It would serve as a package in which to encapsulate our globals. We would access them via get() and assign(). For instance, the lines

if (is.null(sim$evnts)) {
sim$evnts <<- newevnt
in schedevnt() would become
if (is.null(get("evnts",envir=simenv))) {
assign("evnts",newevnt,envir=simenv)

Yes, this is cluttered too, but at least it is not complex like lists of lists of lists. And it does protect against unwittingly writing to an unrelated variable the user has at the top level. Using the superassignment operator still yields the least cluttered code, but this compromise is worth considering

Recursion
A *recursive* function calls itself. If you have not encountered this concept before, it may sound odd, but the idea is actually simple. In rough terms, the idea is this:
To solve a problem of type X by writing a recursive function f():
1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.

A Quicksort Implementation
A classic example is Quicksort, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88).
We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5,
yielding (3,4,5,8,12,13,88), as desired.
R's vector-filtering capability and its c() function make implementation of Quicksort quite easy.
NOTE *This example is for the purpose of demonstrating recursion. R's own sort function, sort(), is much faster, as it is written in C.*

```
qs <- function(x) {
if (length(x) <= 1) return(x)
pivot <- x[1]
therest <- x[-1]
sv1 <- therest[therest < pivot]
sv2 <- therest[therest >= pivot]
sv1 <- qs(sv1)
sv2 <- qs(sv2)
```

```
return(c(sv1,pivot,sv2))
}
```

Note carefully the *termination condition*:
if (length(x) <= 1) return(x)
Without this, the function would keep calling itself repeatedly on empty vectors, executing forever. (Actually, the R interpreter would eventually refuse to go any further, but you get the idea.)
Sounds like magic? Recursion certainly is an elegant way to solve many problems. But recursion has two potential drawbacks:
• It's fairly abstract. I knew that the graduate student, as a fine mathematician, would take to recursion like a fish to water, because recursion is really just the inverse of proof by mathematical induction. But many programmers find it tough.
• Recursion is very lavish in its use of memory, which may be an issue in R if applied to large problems.

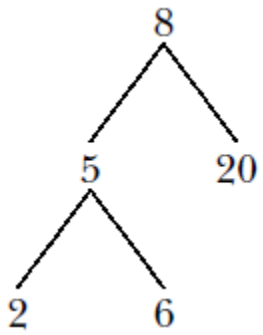
Quick sort another example

```
quicksort <- function(x) {
  if(length(x) <= 1) return(x)

  pivot      <- x[1]
  remainder  <- x[-1]
  remainder_1 <- remainder[remainder < pivot]
  remainder_2 <- remainder[remainder >= pivot]

  remainder_1 <- quicksort(remainder_1)
  remainder_2 <- quicksort(remainder_2)

  return(c(remainder_1, pivot, remainder_2))
}

if (require("testthat")) {
  test_that("quicksort function sorts properly", {
    x <- c(1, 6, 2, 10, -9, 12, 4, -12)
    y <- c(-12, -9, 1, 2, 4, 6, 10, 12)
    expect_that(quicksort(x), equals(y))
    expect_that(quicksort(x), equals(sort(x)))
  })
}
```

Extended Example: A Binary Search Tree
Treelike data structures are common in both computer science and statistics. In R, for example, the rpart library for a recursive portioning approach to regression and classification is very popular. Trees obviously have applications
in genealogy, and more generally, graphs form the basis of analysis of social networks.

For the sake of simplicity, our example here will be a binary search tree, a classic computer science data structure that has the following property: In each node of the tree, the value at the left link, if any, is less than or equal to that of the parent, while the value at the right link, if any, is greater than that of the parent.

Here is an example:



We've stored 8 in the *root*—that is, the head—of the tree. Its two child nodes contain 5 and 20, and the former itself has two child nodes, which store 2 and 6.
Note that the nature of binary search trees implies that at any node, all of the elements in the node's left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree—5, 2 and 6—are less
than 8, while 20 is greater than 8.

If implemented in C, a tree node would be represented by a C struct, similar to an R list, whose contents are the stored value, a pointer to the left child, and a pointer to the right child. But since R lacks pointer variables,

Before discussing the code, let's run through a quick session of tree building using its routines.

```
> x <- newtree(8,3)
> x
$mat
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NA NA
[3,] NA NA NA
$nxt
[1] 2
$inc
[1] 3
> x <- ins(1,x,5)
> x
$mat

[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NA NA
$nxt
[1] 3
$inc
[1] 3
> x <- ins(1,x,6)
> x
$mat
```

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA 3 5
[3,] NA NA 6
$nxt
[1] 4
$inc
[1] 3
> x <- ins(1,x,2)
> x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA NA
[6,] NA NA NA
$nxt
[1] 5
$inc
[1] 3
> x <- ins(1,x,20)
> x
$mat
[,1] [,2] [,3]

[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NA NA
$nxt
[1] 6
$inc
[1] 3
```


What happened here? First, the command containing our call newtree(8,3) creates a new tree, assigned to x, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time.

The result is that the matrix component of the list x is now as follows:

```
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NA NA
[3,] NA NA NA
```

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children.

We then make the call ins(1,x,5) to insert a second value, 5, into the tree x. The argument 1 specifies the root. In other words, the call says,

"Insert 5 in the subtree of x whose root is in row 1." Note that we need to reassign the return value of this call back to x. Again, this is due to the lack of pointer variables in R. The matrix now looks like this:

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NA NA
```

The element 2 means that the left link out of the node containing 8 is meant to point to row 2, where our new element 5 is stored.

The session continues in this manner. Note that when our initial allotment of three rows is full, ins() allocates three new rows, for a total of six. In the end, the matrix is as follows:

```
[,1] [,2] [,3]
[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6

[4,] NA NA 2
[5,] NA NA 20
[6,] NA NA NA
```

This represents the tree we graphed for this example.

The code follows. Note that it includes only routines to insert new items and to traverse the tree. The code for deleting a node is somewhat more complex, but it follows a similar pattern

```
# routines to create trees and insert items into them are included
2 # below; a deletion routine is left to the reader as an exercise
3
4 # storage is in a matrix, say m, one row per node of the tree; if row
5 # i contains (u,v,w), then node i stores the value w, and has left and
6 # right links to rows u and v; null links have the value NA
7
8 # the tree is represented as a list (mat,nxt,inc), where mat is the
9 # matrix, nxt is the next empty row to be used, and inc is the number of
10 # rows of expansion to be allocated whenever the matrix becomes full
11
12 # print sorted tree via in-order traversal
13 printtree <- function(hdidx,tr) {
14 left <- tr$mat[hdidx,1]
15 if (!is.na(left)) printtree(left,tr)
16 print(tr$mat[hdidx,3]) # print root
17 right <- tr$mat[hdidx,2]
18 if (!is.na(right)) printtree(right,tr)
19 }
20
21 # initializes a storage matrix, with initial stored value firstval
22 newtree <- function(firstval,inc) {
23 m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
24 m[1,3] <- firstval
25 return(list(mat=m,nxt=2,inc=inc))
26 }
27
28 # inserts newval into the subtree of tr, with the subtree's root being
29 # at index hdidx; note that return value must be reassigned to tr by the
```

```
30 # caller (including ins() itself, due to recursion)
31 ins <- function(hdidx,tr,newval) {
32 # which direction will this new node go, left or right?
33 dir <- if (newval <= tr$mat[hdidx,3]) 1 else 2
34 # if null link in that direction, place the new node here, otherwise
35 # recurse
36 if (is.na(tr$mat[hdidx,dir])) {
37 newidx <- tr$nxt # where new node goes
38 # check for room to add a new element
39 if (tr$nxt == nrow(tr$mat) + 1) {
tr$mat <-
41 rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
42 }
43 # insert new tree node
44 tr$mat[newidx,3] <- newval
45 # link to the new node
46 tr$mat[hdidx,dir] <- newidx
47 tr$nxt <- tr$nxt + 1 # ready for next insert
48 return(tr)
49 } else tr <- ins(tr$mat[hdidx,dir],tr,newval)
50 }
```

There is recursion in both printtree() and ins(). The former is definitely the easier of the two, so let's look at that first. It prints out the tree, in sorted order. Recall our description of a recursive function f() that solves a problem of category X: We have f() split the original X problem into one or more smaller X problems, call f() on them, and combine the results. In this case, our problem's category X is to print a tree, which could be a subtree of a larger one. The role of the function on line 13 is to print the given tree, which it does by calling itself in lines 15 and 18. This mechanism is seen in the if() statements in lines 15 and 18. When we come to a null link, we do not continue to recurse. The recursion in ins() follows the same principles but is considerably more delicate. Here, our "category X" is an insertion of a value into a subtree.

UNIT-III
Doing Math and Simulation in R

Math Functions

R includes an extensive set of built-in math functions. Here is a partial list:

- exp(): Exponential function, base e
- log(): Natural logarithm
- log10(): Logarithm base 10
- sqrt(): Square root
- abs(): Absolute value
- sin(), cos(), and so on: Trig functions
- min() and max(): Minimum value and maximum value within a vector
- which.min() and which.max(): Index of the minimal element and maximal element of a vector
- pmin() and pmax(): Element-wise minima and maxima of several vectors
- sum() and prod(): Sum and product of the elements of a vector
- cumsum() and cumprod(): Cumulative sum and product of the elements of a vector
- round(), floor(), and ceiling(): Round to the closest integer, to the closest integer below, and to the closest integer above
- factorial(): Factorial function

Extended Example: Calculating a Probability

As our first example, we'll work through calculating a probability using the prod() function. Suppose we have *n* independent events, and the *i*th event has the probability *pi* of occurring. What is the probability of exactly one of these events occurring?

Suppose first that $n = 3$ and our events are named A, B, and C. Then we break down the computation as follows:

P(exactly one event occurs) =
P(A and not B and not C) +
P(not A and B and not C) +
P(not A and not B and C)

P(A and not B and not C) would be $pA(1 - pB)(1 - pC)$, and so on.

For general *n*, that is calculated as follows:

$$\sum_{i=1}^{n} p_i(1-p_1)...(1-p_{i-1})(1-p_{i+1})...(1-p_n)$$

(The *i*th term inside the sum is the probability that event *i* occurs and all the others do *not* occur.)

Here's code to compute this, with our probabilities *pi* contained in the vector p:

```
exactlyone <- function(p) {
notp <- 1 - p
tot <- 0.0
for (i in 1:length(p))
tot <- tot + p[i] * prod(notp[-i])
return(tot)
}
```

How does it work? Well, the assignment

```
notp <- 1 - p
```

creates a vector of all the "not occur" probabilities $1 - pj$, using recycling. The expression notp[-i] computes the product of all the elements of notp, except the *i*th—exactly what we need.

8.1.2 Cumulative Sums and Products :

As mentioned, the functions cumsum() and cumprod() return cumulative sums and products.

```
> x <- c(12,5,13)
> cumsum(x)
[1] 12 17 30
> cumprod(x)
[1] 12 60 780
```

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30.

The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

8.1.3 Minima and Maxima :

There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if pmin() is applied to two or more vectors,

it returns a vector of the pair-wise minima, hence the name pmin.Here's an example:

```
> z
[,1] [,2]
[1,] 1 2
[2,] 5 3
[3,] 6 2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

You can use more than two arguments in pmin(), like this:

```
> pmin(z[1,],z[2,],z[3,])
[1] 1 2
```

The 1 in the output is the minimum of 1, 5, and 6, with a similar computation leading to the 2.

The max() and pmax() functions act analogously to min() and pmin(). Function minimization/maximization can be done via nlm() and optim().

For example, let's find the smallest value of $f(x) = x2 - \sin(x)$.

```
> nlm(function(x) return(x^2-sin(x)),8)
$minimum
[1] -0.2324656
$estimate
[1] 0.4501831
$gradient
[1] 4.024558e-09
$code
[1] 1
$iterations
[1] 5
```

Here, the minimum value was found to be approximately −0.23, occurring at $x = 0.45$. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case.
The second argument specifies the initial guess, which we set to be 8. (This second argument was picked pretty arbitrarily here, but in some problems, you may need to experiment to find a value that will lead to convergence.)

Calculus :
R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

```
> D(expression(exp(x^2)),"x") # derivative
exp(x^2) * (2 * x)
> integrate(function(x) x^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

Here, R reported

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2}$$

and

$$\int_0^1 x^2\,dx \approx 0.3333333$$

You can find R packages for differential equations (odesolve), for interfacing R with the Yacas symbolic math system (ryacas), and for other calculus operations. These packages, and thousands of others, are available from the
Comprehensive R Archive Network (CRAN);

Functions for Statistical Distributions :
R has functions available for most of the famous statistical distributions.
Prefix the name as follows:
• With d for the density or probability mass function (pmf)
• With p for the cumulative distribution function (cdf)
• With q for quantiles
• With r for random number generation
The rest of the name indicates the distribution. Table 8-1 lists some common statistical distribution functions.

Table 8-1: Common R Statistical Distribution Functions

Distribution	Density/pmf	cdf	Quantiles	Random Numbers
Normal	dnorm()	pnorm()	qnorm() r	norm()
Chi square	dchisq()	pchisq()	qchisq()	rchisq()
Binomia	l dbinom()	pbinom()	qbinom()	rbinom()

As an example, let's simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

```
> mean(rchisq(1000,df=2))
[1] 1.938179
```

The r in rchisq specifies that we wish to generate random numbers— in this case, from the chi-square distribution. As seen in this example, the first argument in the r-series functions is the number of random variates to generate.

These functions also have arguments specific to the given distribution families. In our example, we use the df argument for the chi-square family, indicating the number of degrees of freedom.

Let's also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

```
> qchisq(0.95,2)
[1] 5.991465
```

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile.
The first argument in the d, p, and q series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points.

Let's find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

```
qchisq(c(0.5,0.95),df=2)
[1] 1.386294 5.991465
```

Sorting :
Ordinary numerical sorting of a vector can be done with the sort() function, as in this example:

```
> x <- c(13,5,12,5)
> sort(x)
[1] 5 5 12 13
> x
[1] 13 5 12 5
```

Note that x itself did not change, in keeping with R's functional language philosophy.

If you want the indices of the sorted values in the original vector, use the order() function. Here's an example:

```
> order(x)
[1] 2 4 3 1
```

This means that x[2] is the smallest value in x, x[4] is the second smallest,x[3] is the third smallest, and so on.

You can use order(), together with indexing, to sort data frames, like this:

```
> y
V1 V2
1 def 2
2 ab 5
3 zzzz 1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,]
> z
V1 V2
```

3 zzzz 1
1 def 2
2 ab 5
What happened here? We called order() on the second column of y, yielding a vector r, telling us where numbers should go if we want to sort them. The 3 in this vector tells us that x[3,2] is the smallest number in x[,2]; the 1 tells us that x[1,2] is the second smallest; and the 2 tells us that x[2,2] is the third smallest. We then use indexing to produce the frame sorted by
column 2, storing it in z.

You can use order() to sort according to character variables as well as numeric ones, as follows:

```
> d
kids ages
1 Jack 12
2 Jill 10
3 Billy 13

> d[order(d$kids),]
kids ages
3 Billy 13
1 Jack 12
2 Jill 10

> d[order(d$ages),]
kids ages
2 Jill 10
1 Jack 12
3 Billy 13
```

A related function is rank(), which reports the rank of each element of a vector.

```
> x <- c(13,5,12,5)
> rank(x)
[1] 4.0 1.5 3.0 1.5
```

This says that 13 had rank 4 in x; that is, it is the fourth smallest. The value 5 appears twice in x, with those two being the first and second smallest, so the rank 1.5 is assigned to both. Optionally, other methods of handling ties can be specified.

8.4 Linear Algebra Operations on Vectors and Matrices

Multiplying a vector by a scalar works directly, as you saw earlier. Here's another example:

```
> y
[1] 1 3 4 10
> 2*y
[1] 2 6 8 20
```

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

```
> crossprod(1:3,c(5,12,13))
[,1]
[1,] 68
```

The function computed 1 · 5 + 2 · 12 + 3 · 13 = 68.

For matrix multiplication in the mathematical sense, the operator to use is %*%, not *. For instance, here we compute the matrix product:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix}$$

Here's the code:

```
> a
[,1] [,2]
[1,] 1 2
[2,] 3 4
> b
[,1] [,2]
[1,] 1 -1
[2,] 0 1
> a %*% b
[,1] [,2]
[1,] 1 1
[2,] 3 1
```

The function solve() will solve systems of linear equations and even find matrix inverses. For example, let's solve this system:

$x1 + x2 = 2$

$-x1 + x2 = 4$

Its matrix form is as follows:

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

Here's the code:

```
> a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)
> b <- c(2,4)
> solve(a,b)
[1] 3 1
> solve(a)
[,1] [,2]
[1,] 0.5 0.5
[2,] -0.5 0.5
```

In that second call to solve(), the lack of a second argument signifies that we simply wish to compute the inverse of the matrix.

Here are a few other linear algebra functions:

- t(): Matrix transpose
- qr(): QR decomposition
- chol(): Cholesky decomposition
- det(): Determinant
- eigen(): Eigenvalues/eigenvectors

• diag(): Extracts the diagonal of a square matrix (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix).
• sweep(): Numerical analysis sweep operations Note the versatile nature of diag(): If its argument is a matrix, it returns
a vector, and vice versa. Also, if the argument is a scalar, the function returns the identity matrix of the specified size.

```
> m
[,1] [,2]
[1,] 1 2
[2,] 7 8
> dm <- diag(m)
> dm
[1] 1 8
> diag(dm)
[,1] [,2]
[1,] 1 0
[2,] 0 8
> diag(3)
[,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1
```

The sweep() function is capable of fairly complex operations. As a simple example, let's take a 3-by-3 matrix and add 1 to row 1, 4 to row 2, and 7 to row 3.

```
> m
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
> sweep(m,1,c(1,4,7),"+")
[,1] [,2] [,3]
[1,] 2 3 4
[2,] 8 9 10
[3,] 14 15 16
```

The first two arguments to sweep() are like those of apply(): the array and the margin, which is 1 for rows in this case. The fourth argument is a function to be applied, and the third is an argument to that function (to the "+" function).

8.4.1 Extended Example: Vector Cross Product:

Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors ($x1$, $x2$, $x3$) and ($y1$, $y2$, $y3$) in threedimensional space is a new three-dimensional vector,

This can be expressed compactly as the expansion along the top row of the determinant, as shown in Equation 8.2.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}$$

Here, the elements in the top row are merely placeholders. Don't worry about this bit of pseudomath. The point is that the cross product vector can be computed as a sum of subdeterminants. For instance, the first component in Equation 8.1, $x2y3 - x3y2$, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column in
Equation

$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix}$$

Our need to calculate subdeterminants—that is determinants of submatrices—fits perfectly with R, which excels at specifying submatrices. This suggests calling det() on the proper submatrices, as follows:

```
xprod <- function(x,y) {
m <- rbind(rep(NA,3),x,y)
xp <- vector(length=3)
for (i in 1:3)
xp[i] <- -(-1)^i * det(m[2:3,-i])
return(xp)
}
```

Note that even R's ability to specify values as NA came into play here to deal with the "placeholders" mentioned above.
All this may seem like overkill. After all, it wouldn't have been hard to code Equation 8.1 directly, without resorting to use of submatrices and determinants. But while that may be true in the three-dimensional case, the approach shown here is quite fruitful in the *n*-ary case, in *n*-dimensional space. The cross product there is defined as an *n*-by-*n* determinant of the
form shown in Equation 8.1, and thus the preceding code generalizes perfectly.

Extended Example: Finding Stationary Distributions of Markov Chains:

A Markov chain is a random process in which we move among various *states*,in a "memoryless" fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite.

As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads.Our state at any time *i* will the number of consecutive heads we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

The central interest in Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we'll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions\

57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar

if we are in state 2 and toss a head, so 0.143 × 0.5 = 0.071 of our tosses will result in wins.

Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads.

Let *pij* denote the *transition probability* of moving from state *i* to state *j*

during a time step. In the game example, for instance, $p23 = 0.5$, reflecting the fact that with probability 1/2, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus

$p21 = 0.5$.

We are interested in calculating the vector $\pi = (\pi 1, ..., \pi s)$, where πi is the long-run proportion of time spent at state i, over all states i. Let P denote the transition probability matrix whose ith row, jth column element is pij . Then it can be shown that π must satisfy Equation
$\pi = \pi P$
which is equivalent to Equation 8.5:
$(I - P^T)\pi = 0$
Here I is the identity matrix and PT denotes the transpose of P. Any single one of the equations in the system of Equation 8.5 is redundant.
We thus eliminate one of them, by removing the last row of $I-P$ in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation

But note that there is also the constraint shown in Equation

$$\sum_i \pi_i = 1$$

In matrix terms, this is as follows:

$$1_n^T \pi = 1$$

where $1n$ is a vector of n 1s.
So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s and, on the right-hand side, replace the removed 0 with a 1. We can then solve the system.

All this can be computed with R's solve() function, as follows:

```
1 findpi1 <- function(p) {
2 n <- nrow(p)
3 imp <- diag(n) - t(p)
imp[n,] <- rep(1,n)
5 rhs <- c(rep(0,n-1),1)
6 pivec <- solve(imp,rhs)
7 return(pivec)
8 }
```

Here are the main steps:
1. Calculate $I - PT$ in line 3. Note again that diag(), when called with a scalar argument, returns the identity matrix of the size given by that argument.
2. Replace the last row of P with 1 values in line 4.
3. Set up the right-hand side vector in line 5.
4. Solve for π in line 6.

Another approach, using more advanced knowledge, is based on eigenvalues. Note from Equation 8.4 that π is a left eigenvector of P with eigenvalue
1. This suggests using R's eigen() function, selecting the eigenvector corresponding to that eigenvalue. (A result from mathematics, the Perron- Frobenius theorem, can be used to carefully justify this.)

Since π is a left eigenvector, the argument in the call to eigen() must be P transpose rather than P. In addition, since an eigenvector is unique
only up to scalar multiplication, we must deal with two issues regarding the eigenvector returned to us by eigen():
• It may have negative components. If so, we multiply by -1.
• It may not satisfy Equation 8.6. We remedy this by dividing by the length
of the returned vector.
Here is the code:

```
1 findpi2 <- function(p) {
2 n <- nrow(p)
3 # find first eigenvector of P transpose
4 pivec <- eigen(t(p))$vectors[,1]
5 # guaranteed to be real, but could be negative
6 if (pivec[1] < 0) pivec <- -pivec
7 # normalize to sum to 1
8 pivec <- pivec / sum(pivec)
9 return(pivec)
10 }
```

The return value of eigen() is a list. One of the list's components is a matrix named vectors. These are the eigenvectors, with the ith column being the eigenvector corresponding to the ith eigenvalue. Thus, we take column 1 here.

Set Operations
R includes some handy set operations, including these:
• union(x,y): Union of the sets x and y
• intersect(x,y): Intersection of the sets x and y
• setdiff(x,y): Set difference between x and y, consisting of all elements of
x that are not in y
• setequal(x,y): Test for equality between x and y
• c %in% y: Membership, testing whether c is an element of the set y
• choose(n,k): Number of possible subsets of size k chosen from a set of
size n
Here are some simple examples of using these functions:

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
```

```
> 2 %in% x
[1] TRUE
> 2 %in% y
[1] FALSE
> choose(5,2)
[1] 10
```

Recall from Section 7.12 that you can write your own binary operations. For instance, consider coding the symmetric difference between two sets—that is, all the elements belonging to exactly one of the two operand sets. Because the symmetric difference between sets x and y consists exactly of those elements in x but not y and vice versa, the code consists of easy calls to setdiff() and union(), as follows:

```
> symdiff
function(a,b) {
sdfxy <- setdiff(x,y)
sdfyx <- setdiff(y,x)

return(union(sdfxy,sdfyx))
}
```

Let's try it.

```
> x
[1] 1 2 5
> y
[1] 5 1 8 9
> symdiff(x,y)
[1] 2 8 9
```

Here's another example: a binary operand for determining whether one set u is a subset of another set v. A bit of thought shows that this property is equivalent to the intersection of u and v being equal to u. Hence we have another easily coded function:

```
> "%subsetof%" <- function(u,v) {
+ return(setequal(intersect(u,v),u))
+ }
> c(3,8) %subsetof% 1:10
[1] TRUE
> c(3,8) %subsetof% 5:10
[1] FALSE
```

The function combn() generates combinations. Let's find the subsets of {1,2,3} of size 2.

```
> c32 <- combn(1:3,2)
> c32
[,1] [,2] [,3]
[1,] 1 1 2
[2,] 2 3 3
> class(c32)
[1] "matrix"
```

The results are in the columns of the output. We see that the subsets of {1,2,3} of size 2 are (1,2), (1,3), and (2,3).

The function also allows you to specify a function to be called by combn() on each combination. For example, we can find the sum of the numbers in each subset, like this:

```
> combn(1:3,2,sum)
[1] 3 4 5
```

The first subset, {1,2}, has a sum of 2, and so on.

Simulation Programming in R

One of the most common uses of R is simulation. Let's see what kinds of tools R has available for this application.

8.6.1 Built-In Random Variate Generators

As mentioned, R has functions to generate variates from a number of different distributions. For example, rbinom() generates binomial or Bernoulli random variates.1

Let's say we want to find the probability of getting at least four heads out of five tosses of a coin (easy to find analytically, but a handy example).

Here's how we can do this:

```
> x <- rbinom(100000,5,0.5)
> mean(x >= 4)
[1] 0.18829
```

First, we generate 100,000 variates from a binomial distribution with five trials and a success probability of 0.5. We then determine which of them has a value 4 or 5, resulting in a Boolean vector of the same length as x. The TRUE
and FALSE values in that vector are treated as 1s and 0s by mean(), giving us our estimated probability (since the average of a bunch of 1s and 0s is the proportion of 1s).

Other functions include rnorm() for the normal distribution, rexp() for the exponential, runif() for the uniform, rgamma() for the gamma, rpois() for the Poisson, and so on.

Here is another simple example, which finds $E[\max(X, Y)]$, the
expected value of the maximum of independent N(0,1) random variables
X and Y:

```
sum <- 0
nreps <- 100000
for (i in 1:nreps) {
xy <- rnorm(2) # generate 2 N(0,1)s
sum <- sum + max(xy)
}
print(sum/nreps)
```

We generated 100,000 pairs, found the maximum for each, and averaged those maxima to obtain our estimated expected value. The preceding code, with an explicit loop, may be clearer, but as before, if we are willing to use some more memory, we can do this more compactly.

```
> emax
function(nreps) {
x <- rnorm(2*nreps)
maxxy <- pmax(x[1:nreps],x[(nreps+1):(2*nreps)])
return(mean(maxxy))
}
```

Here, we generated double nreps values. The first nreps value simulates X, and the remaining nreps value represents Y. The pmax() call then computes
the pair-wise maxima that we need. Again, note the contrast here between max() and pmax(), the latter producing pair-wise maxima.

Accessing the Keyboard and Monitor

R provides several functions for accesssing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

10.1.1 Using the scan() Function

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list.

Suppose we have files named *z1.txt*, *z2.txt*, *z3.txt*, and *z4.txt*. The *z1.txt* file contains the following:

```
123
4 5
6
```

The *z2.txt* file contents are as follows:

```
123
4.2 5
6
```

The *z3.txt* file contains this:

```
abc
de f
g
```

And finally, the *z4.txt* file has these contents:

```
abc
123 6
y
```

Let's see what we can do with these files using the scan() function.

```
> scan("z1.txt")
Read 4 items
[1] 123 4 5 6
> scan("z2.txt")
Read 4 items
[1] 123.0 4.2 5.0 6.0
> scan("z3.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
scan() expected 'a real', got 'abc'
> scan("z3.txt",what="")

Read 4 items
[1] "abc" "de" "f" "g"
> scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
```

In the first call, we got a vector of four integers (though the mode is numeric). The second time, since one number was nonintegral, the others were shown as floating-point numbers, too.

In the third case, we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So,the nonnumeric contents of the file *z3* produced an error

The last call worked the same way. The first item was a character string, so it treated all the items that followed as strings too.

Of course, in typical usage, we would assign the return value of scan() to a variable. Here's an example:

```
> v <- scan("z1.txt")
```

By default, scan() assumes that the items of the vector are separated by *whitespace*, which includes blanks, carriage return/line feeds, and horizontal tabs. You can use the optional sep argument for other situations. As example,

we can set sep to the newline character to read in each line as a string, as follows:

```
> x1 <- scan("z3.txt",what="")
Read 4 items
> x2 <- scan("z3.txt",what="",sep="\n")
Read 3 items
> x1
[1] "abc" "de" "f" "g"
> x2
[1] "abc" "de f" "g"
> x1[2]
[1] "de"
> x2[2]
[1] "de f"
```

In the first case, the strings "de" and "f" were assigned to separate elements of x1. But in the second case, we specified that elements of x2 were to be delineated by end-of-line characters, not spaces. Since "de" and "f" are on the same line, they are assigned together to x[2].

More sophisticated methods for reading files will be presented later in this chapter, such as methods to read in a file one line at a time. But if you want to read the entire file at once, scan() provides a quick solution.

You can use scan() to read from the keyboard by specifying an empty string for the filename:

```
> v <- scan("")
1: 12 5 13
4: 3 4 5
7: 8
8:
Read 7 items
> v
[1] 12 5 13 3 4 5 8
```

Note that we are prompted with the index of the next item to be input, and we signal the end of input with an empty line.

If you do not wish scan() to announce the number of items it has read, include the argument quiet=TRUE.

Using the readline() Function

If you want to read in a single line from the keyboard, readline() is very handy.

```
> w <- readline()
abc de f
> w
[1] "abc de f"
```

Typically, readline() is called with its optional prompt, as follows:

```
> inits <- readline("type your initials: ")
type your initials: NM
> inits
[1] "NM"
```

Printing to the Screen

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won't work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

```
> x <- 1:3
> print(x^2)
[1] 1 4 9
```

Recall that print() is a *generic* function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called..

It's a little better to use cat() instead of print(), as the latter can print only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

```
> print("abc")
[1] "abc"
> cat("abc\n")
abc
```

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line. The arguments to cat() will be printed out with intervening spaces:

```
> x
[1] 1 2 3
> cat(x,"abc","de\n")
1 2 3 abc de
```

If you don't want the spaces, set sep to the empty string "", as follows:

```
> cat(x,"abc","de\n",sep="")
123abcde
```

Any string can be used for sep. Here, we use the newline character:

```
> cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```

You can even set sep to be a vector of strings, like this:

```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

10.2 Reading and Writing Files:

Now that we've covered the basics of I/O, let's get to some more practical applications of reading and writing files. The following sections discuss reading data frames or matrices from files, working with text files, accessing files on remote machines, and getting file and directory information.

Reading a Data Frame or Matrix from a File :

we discussed the use of the function read.table() to read in a data frame. As a quick review, suppose the file *z* looks like this:

```
name age
John 25
Mary 28
Jim 19
```

The first line contains an optional header, specifying column names. We could read the file this way:

```
> z <- read.table("z",header=TRUE)
> z
name age
1 John 25
2 Mary 28
3 Jim 19
```

Note that scan() would not work here, because our file has a mixture of numeric and character data (and a header).

There appears to be no direct way of reading in a matrix from a file, but it can be done easily with other tools. A simple, quick way is to use scan() to read in the matrix row by row. You use the byrow option in the function matrix() to indicate that you are defining the elements of the matrix in a row-wise, rather than column-wise, manner.

For instance, say the file *x* contains a 5-by-3 matrix, stored row-wise:

```
1 0 1
1 1 1
1 1 0
1 1 0
0 0 1
```

We can read it into a matrix this way:

```
> x <- matrix(scan("x"),nrow=5,byrow=TRUE)
```

This is fine for quick, one-time operations, but for generality, you can use read.table(), which returns a data frame, and then convert via as.matrix().

Here is a general method:

```
read.matrix <- function(filename) {
as.matrix(read.table(filename))
}
```

Reading Text Files:

In computer literature, there is often a distinction made between *text files* and *binary files*. That distinction is somewhat misleading—every file is binary in the sense that it consists of 0s and 1s. Let's take the term *text file* to mean a file that consists mainly of ASCII characters or coding for some other human language (such as GB for Chinese) and that uses newline characters to give humans the perception of lines. The latter aspect will turn out to be central

here. Nontext files, such as JPEG images or executable program files, are
generally called *binary files*.
You can use readLines() to read in a text file, either one line at a time or in a single operation. For example, suppose we have a file *z1* with the following
contents:

```
John 25
Mary 28
Jim 19
```

We can read the file all at once, like this:

```
> z1 <- readLines("z1")
> z1
[1] "John 25" "Mary 28" "Jim 19"
```

Since each line is treated as a string, the return value here is a vector of strings—that is, a vector of character mode. There is one vector element for each line read, thus three elements here. Alternatively, we can read it in one line at a time. For this, we first need to create a connection, as described next.

Introduction to Connections

Connection is R's term for a fundamental mechanism used in various kinds of I/O operations. Here, it will be used for file access. The connection is created by calling file(), url(), or one of several other R functions. To see a list of those functions, type this:

```
> ?connection
```

So, we can now read in the *z1* file (introduced in the previous section) line by line, as follows:

```
> c <- file("z1","r")
> readLines(c,n=1)
[1] "John 25"
> readLines(c,n=1)
[1] "Mary 28"
> readLines(c,n=1)
[1] "Jim 19"
```



```
> readLines(c,n=1)
character(0)
```

We opened the connection, assigned the result to c, and then read the file one line at a time, as specified by the argument n=1. When R encountered the end of file (EOF), it returned an empty result. We needed to set up a connection so that R could keep track of our position in the file as we read through it.

We can detect EOF in our code:

```
> c <- file("z","r")
> while(TRUE) {
+ rl <- readLines(c,n=1)
+ if (length(rl) == 0) {
+ print("reached the end")
+ break
+ } else print(rl)
+ }
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "reached the end"
```

If we wish to "rewind"—to start again at the beginning of the file—we can use seek():

```
> c <- file("z1","r")
> readLines(c,n=2)
[1] "John 25" "Mary 28"
> seek(con=c,where=0)
[1] 16
> readLines(c,n=1)
[1] "John 25"
```

The argument where=0 in our call to seek() means that we wish to position the file pointer zero characters from the start of the file—in other words, directly at the beginning.

The call returns 16, meaning that the file pointer was at position 16 before we made the call. That makes sense. The first line consists of "John 25" *plus* the end-of-line character, for a total of eight characters, and the same is true for the second line. So, after reading the first two lines, we were at position
16.

Writing to a File :

Given the statistical basis of R, file reads are probably much more common than writes. But writes are sometimes necessary, and this section will present methods for writing to files.
The function write.table() works very much like read.table(), except that it writes a data frame instead of reading one.

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d
kids ages
1 Jack 12
2 Jill 10
> write.table(d,"kds")
```

The file *kds* will now have these contents:

```
"kids" "ages"
"1" "Jack" 12
"2" "Jill" 10
```

In the case of writing a matrix to a file, just state that you do not want row or column names, as follows:

```
> write.table(xc,"xcnew",row.names=FALSE,col.names=FALSE)
```

The function cat() can also be used to write to a file, one part at a time.
Here's an example:

```
> cat("abc\n",file="u")
> cat("de\n",file="u",append=TRUE)
```

The first call to cat() creates the file *u*, consisting of one line with contents "abc". The second call appends a second line. Unlike the case of using the writeLines() function (which we'll discuss next), the file is automatically saved after each operation. For instance, after the previous calls, the file will look like this:

```
abc
de
```

You can write multiple fields as well. So:

```
> cat(file="v",1,2,"xyz\n")
```

would produce a file *v* consisting of a single line:

```
1 2 xyz
```

You can also use writeLines(), the counterpart of readLines(). If you use a connection, you must specify "w" to indicate you are writing to the file, not reading from it:

```
> c <- file("www","w")
> writeLines(c("abc","de","f"),c)
> close(c)
```

The file *www* will be created with these contents:

abc
de
f

Getting File and Directory Information :

R has a variety of functions for getting information about directories and files, setting file access permissions, and the like. The following are a few examples:

• file.info(): Gives file size, creation time, directory-versus-ordinary file status, and so on for each file whose name is in the argument, a character vector.

• dir(): Returns a character vector listing the names of all the files in the directory specified in its first argument. If the optional argument recursive=TRUE is specified, the result will show the entire directory tree rooted at the first argument.

• file.exists(): Returns a Boolean vector indicating whether the given file exists for each name in the first argument, a character vector.

• getwd() and setwd(): Used to determine or change the current workingdirectory.

To see all the file- and directory-related functions, type the following: > ?files

FREQUENTLY ASKED QUESTIONS

UNIT-I

1. Different Types of Data Structures in R
2. Variables ,Constants, Data Types in R
3. R Installation Procedure
4. Vectors in R

UNIT-II

1. Different Operators in R
2. Control Structures in R
3. Default Values for Arguments
4. Returning Complex Objects
5. Quiick Sort Implementation
6. Binary Search Tree

UNIT-III

1. Calculating Cumulative sums, and products Minima Maxima and Calculus
2. Different Functions for statistical Distribution
3. Finding Stationary Distribution of Markanov Chains
4. Functions for accessing keyboard and mouse, Reading and writing Files
5. Linear Algebra Operations on Vectors and Matrices