MODULE 2

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

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-then else constructs, which can make the code confusing. Let's take a look at an example that uses next.

```
sim <- function(nreps) {
commdata <- list()
commdata$countabsamecomm <- 0
for (rep in 1:nreps) {
commdata$whosleft <- 1:20
commdata$numabchosen <- 0
commdata <- choosecomm(commdata,5)
if (commdata$numabchosen > 0) next
commdata <- choosecomm(commdata,4)
if (commdata$numabchosen > 0) next
commdata <- choosecomm(commdata,3)
}
print(commdata$countabsamecomm/nreps)
}
```

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.

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

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

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

Arithmetic and Boolean Operators and Values

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

Table 7-1: Basic R Operators

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

Default Values for Arguments

> 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. Note, though, that because R uses lazy evaluation—it does not evaluate an expression until/unless it needs to—the named argument may not actually be used.

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
}
```

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>
```

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)
```

An alternative is to edit it using the edit() function.

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 a language object, 2*x+3 is indeed a call—in fact, it is one function call nested within another.

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, 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.

Recursion

Once a mathematics PhD student whom I knew to be quite bright, but who had little programming background, sought my advice on how to write a certain function. I quickly said, "You don't even need to tell me what the function is supposed to do. The answer is to use recursion." Startled, he asked what recursion is. I advised him to read about the famous Towers of Hanoi problem. Sure enough, he returned the next day, reporting that he was able to solve his problem in just a few lines of code, using recursion. Obviously, recursion can be a powerful tool. Well then, what is it? 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.

```
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:

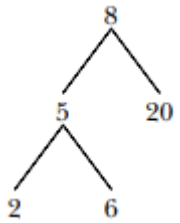
- It's fairly abstract. I knew that the graduate student, as a fine mathematician, would take to recursion like a fish to water, because recursion is really just the inverse of proof by mathematical induction. But many programmers find it tough.
- Recursion is very lavish in its use of memory, which may be an issue in R if applied to large problems.

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 partioning 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. However, there are real issues with tree structures in R, many of them related to the fact that R does not have pointer-style references, as discussed in Section 7.7. Indeed, for this reason and for performance purposes, a better option is often to write the core code in C with an R wrapper, as we'll discuss in Chapter 15. Yet trees can be implemented in R itself, and if performance is not an issue, using this approach may be more convenient. 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, what can we do? Our solution is to go back to the basics. In the old prepointer days in FORTRAN, linked data structures were implemented in long arrays. A pointer, which in C is a memory address, was an array index instead. Specifically, we'll represent each node by a row in a three-column matrix. The node's stored value will be in the third element of that row, while the first and second elements will be the left and right links. For instance, if the first element in a row is 29, it means that this node's left link points to the node stored in row 29 of the matrix. Remember that allocating space for a matrix in R is a time-consuming activity. In an effort to amortize the memory-allocation time, we allocate new space for a tree's matrix several rows at a time, instead of row by row. The number of rows allocated each time will be given in the variable inc. As is common with tree traversal, we implement our algorithm with recursion.

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,] 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: