UNIT-II

R Programming Structures, Control Statements, Loops, - Looping Over Nonvector Sets,- If-Else, Arithmetic and Boolean Operators and values, Default Values for Argument, Return Values,Deciding Whether to explicitly call return- Returning Complex Objects, Functions are Objective,No Pointers in R, Recursion, A Quicksort Implementation-Extended Extended Example: A Binary Search Tree.

R PROGRAMMING STRUCTURES

R is a block-structured language in the manner of the ALGOL-descendant family, such as C, C++, Python, Perl, and so on.

Blocks are delineated by braces, though braces are optional if the block consists of just a single statement. Statements are separated by newline characters or, optionally, by semicolons. Here, the basic structures of R as a programming language.

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

The oddcount() function.

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. The following code comes from an extended example

```
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$numabchosen == 0)
11 commdata <- choosecomm(commdata,3)
12 }
13 }
14 print(commdata$countabsamecomm/nreps)
15 }
```

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. We'll talk more about scan() in Chapter 10.

```
> 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*.

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.

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.

Arithmetic and Boolean Operators and Values

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.

Default Values for Arguments

Read in a data set from a file named exams:

examsquiz <- read.table("exams",header=TRUE)

The column names now appear, with periods replacing blanks:

> head(examsquiz)

Exam.1 Exam.2 Quiz

1 2.0 3.3 4.0

2 3.3 2.0 3.7

3 4.0 4.0 4.0

4 2.3 0.0 3.3

5 2.3 1.0 3.3

6 3.3 3.7 4.0

> 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. For instance, consider the

```
oddcount()> 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
}
}
```

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
```

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.

```
>>> 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.

Recursion

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 Quick sort 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))
}
```

if (length(x) <= 1) return(x)

Without this, the function would keep calling itself repeatedly on empty vectors, executing forever.

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.

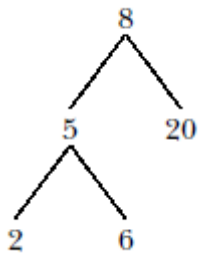
However, there are real issues with tree structures in R, many of them related to the fact that R does not have pointer-style references,

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.

. > 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.

```
1 # 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$mat) + 1) {
40 tr$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.