R programming

March 22, 2017

Contents

1 Control flow and user-written functions

In the normal course of events, the statements in an R program are executed sequentially from the top of the program to the bottom. But there are times that you'll want to execute some statements repetitively, while only executing other statements if certain conditions are met.

For the syntax examples throughout this section, keep the following in mind:

- statement is a single R statement or a compound statement (a group of R statements enclosed in curly braces { } and separated by semicolons).
- cond is an expression that resolves to true or false.
- expr is a statement that evaluates to a number or character string.
- seq is a sequence of numbers or character strings.

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- if, else: testing a condition
- for: execute a loop a fixed number of times
- while: execute a loop while a condition is true
- repeat: execute an infinite loop
- break: break the execution of a loop
- next: skip an interation of a loop
- return: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

1.1 Repetition and looping

Looping constructs repetitively execute a statement or series of statements until a condition isn't true. These include the for and while structures.

FOR

The for loop executes a statement repetitively until a variable's value is no longer contained in the sequence seq. The syntax is

```
for (var in seq) statement
```

A.

```
x <- c("a", "b", "c", "d")
for(i in 1:4) {
print(x[i])
}

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(i in seq_along(x)) {
print(x[i])
}

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"

for(letter in x) {
print(letter)
}

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"

for(i in 1:4) print(x[i])

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

B.

```
iTotal <-  0
for(i in 1:100)
{
    iTotal <- iTotal + i
}
cat("Sum of 1-100:",iTotal,"\n",sep="")

## Sum of 1-100:5050
```

C.

```
szSymbols <- c("MSFT","GOOG","AAPL","INTL","ORCL","SYMC")
for(SymbolName in szSymbols)
{
    cat(SymbolName,"\n",sep="")
}

## MSFT
## GOOG
## AAPL
## INTL
## ORCL
## SYMC
```

D.

```
x <- matrix(1:6, 2, 3)
x

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}

## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

WHILE

A while loop executes a statement repetitively until the condition is no longer true. The syntax is

`while (cond) statement`

A.

```
count <- 0
while(count < 10) {
        print(count)
        count <- count + 1
}

## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

B.

```
i <- 1
iTotal <- 0
while(i <= 100)
{
      iTotal <- iTotal + i
      i <- i + 1
}
cat("Total:",iTotal,"\n",sep="")

## Total:5050
```

C.

```
z <- 5
while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)
        if(coin == 1) { ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}

## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

REPEAT

```
i <- 1
iTotal <- 0
repeat
{
iTotal <- iTotal + i
i <- i + 1
if(i <= 100) next else break
}
cat("Total:",iTotal,"\n",sep="")

## Total:5050
```

Looping in R can be inefficient and time consuming when you're processing the rows or columns of large datasets. Whenever possible, it's better to use R's builtin numerical and character functions in conjunction with the apply family of functions.

1.2 Conditional execution

In conditional execution, a statement or statements are only executed if a specified condition is met. These constructs include if-else, ifelse, and switch.

IF-ELSE

The if-else control structure executes a statement if a given condition is true. Optionally, a different statement is executed if the condition is false. The syntax is

```
#1
if (cond) statement
#2
if (cond) statement1 else statement2
#3
if(<condition1>) {
## do something
} else if(<condition2>) {
## do something different
}else {
## do something different
}
```

```r
a <- 3
if( a == 1)
{
    print("a == 1")
}else
{
    print("a != 1")
}

## [1] "a != 1"
```

```r
grade <- as.factor(c("grade1","grade2"))
if (!is.factor(grade))
{
    grade <- as.factor(grade)
}else
{
    print("Grade already is a factor")
}

## [1] "Grade already is a factor"
```

```r
a <- 4
if( a == 1)
{
print("a == 1")
}else if( a == 2)
{
print("a == 2")
}else
{
print("Not 1 & 2")
}
```

```
## [1] "Not 1 & 2"
```

IFELSE

The ifelse construct is a compact and vectorized version of the if-else construct . The syntax is

`ifelse(cond, statement1, statement2)`

```
x <- matrix(1:6, 2, 3)
ifelse(x >= 0, sqrt(x), NA)

##          [,1]     [,2]     [,3]
## [1,] 1.000000 1.732051 2.236068
## [2,] 1.414214 2.000000 2.449490
```

Use ifelse when you want to take a binary action or when you want to input and output vectors from the construct.

SWITCH

switch chooses statements based on the value of an expression.

```
n <- 1
switch(n,
   print("option1"),
   print("option2"),
   print("option3")
)

## [1] "option1"
```

Next

Next is used to skip an iteration of a loop

```
for(i in 1:10) {
if(i == 3) {
## Skip the number 3
next
}
print(i)
}
```

Break

Break is used to stop a loop.

```
for(i in 1:10) {
if(i == 3) {
break
}
print(i)
}
```

1.3 User-written functions

One of R's greatest strengths is the user's ability to add functions. In fact, many of the functions in R are functions of existing functions. The structure of a function looks like this:

`myfunction <- function(arg1, arg2, ... ){ statements return(object) }`

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have named arguments which potentially have default values.

- The formal arguments are the arguments included in the function definition
- The formals function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be missing or might have default values

Example A:

```
"%g%" <- function(x,y)
{
print(x+y)
print(x-y)
print(x*y)
print(x/y)
}
3%g%5

## [1] 8
## [1] -2
## [1] 15
## [1] 0.6
```

Example B:

```
columnmean <- function(y){
        nc <- ncol(y)
        means <- numeric(nc)
        for(i in 1:nc){
                means[i] <- mean(y[,i])
        }
        means
}
columnmean(airquality)

## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922

columnmean <- function(y,removeNA=TRUE){
        nc <- ncol(y)
        means <- numeric(nc)
        for(i in 1:nc){
                means[i] <- mean(y[,i], na.rm = removeNA)
        }
        means
}
columnmean(airquality)

## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922

columnmean(airquality,FALSE)

## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

Argument Matching

Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is

1. Check for exact match for a named argument
2. Check for a partial match
3. Check for a positional match

```
mydata <- rnorm(100)
sd(mydata)

## [1] 0.9467659

sd(x = mydata)

## [1] 0.9467659

sd(x = mydata, na.rm = FALSE)

## [1] 0.9467659

sd(na.rm = FALSE, x = mydata)

## [1] 0.9467659

sd(na.rm = FALSE, mydata)

## [1] 0.9467659
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
args(lm)

## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list.

Return

Return signals that a function should exit and return a given value

```
x <- c(1,9,2,8,3,7)
y <- c(9,2,8,3,7,2)
pmax(x,y)

## [1] 9 9 8 8 7 7

parmax <- function (a,b){
c <- pmax(a,b)
median(c)}
parmax(x,y)

## [1] 8
```

If you want to return two or more variables from a function you should use return with a list containing the variables to be returned.

```
parboth <- function (a,b) {
c <- pmax(a,b)
d <- pmin(a,b)
answer <- list(median(c),median(d))
names(answer)[[1]] <- "median of the parallel maxima"
names(answer)[[2]] <- "median of the parallel minima"
return(answer) }
parboth(x,y)

## $`median of the parallel maxima`
## [1] 8
##
## $`median of the parallel minima`
## [1] 2
```

The “...” argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions. ... is often used when extending another function and you don’t want to copy the entire argument list of the original function.

```
myplot <- function(x, y, type = "l", ...) {
plot(x, y, type = type, ...)
}
```

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
args(paste)

## function (..., sep = " ", collapse = NULL)
## NULL

args(cat)

## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

One catch with ... is that any arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched.

```
paste("a", "b", sep = ":")

## [1] "a:b"

paste("a", "b", se = ":")

## [1] "a b :"
```

Programming tips

- Know exactly what you are trying to achieve.
- Keep it simple.
- Clever is good, but clear is better.
- Test each line as you go along, to make sure it does what you want it to do.
- Put plenty of comments in the code, using # for documentation.
- Use variable names and function names that are self-explanatory.
- Do not use attach in programs. Use with, or refer to variables within named dataframes.
- Try different ways of doing the same thing, and select the fastest method.
- Use indents (tabs) to improve clarity of loops and if statements.
- Build up the program from small, independently tested functions.
- Stop tinkering once it works effectively.

A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
lm
lm <- function(x) { x * x }
lm
```

How does R know what value to assign to the symbol lm? Why doesn't it give it the value of lm that is in the stats package? When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly The search list can be found by using the search function.

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list.

```
search()

## [1] ".GlobalEnv"        "package:knitr"     "package:stats"
## [4] "package:graphics"  "package:grDevices" "package:utils"
## [7] "package:datasets"  "Autoloads"         "package:base"
```

Exercise

Question: We have two sample:
A 79.98, 80.04, 80.02, 80.04, 80.03, 80.03, 80.04, 79.97, 80.05, 80.03, 80.02, 80.00, 80.02
B 80.02, 79.94, 79.98, 79.97, 79.97, 80.03, 79.95, 79.97
Find T statistics.
Hint: T statistics are

$$T = \frac{(\bar{X} - \bar{Y})}{S\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}},$$

where

$$S = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}$$

Answer:

```
## [1] 3.472245
```

1.4 Messages, Warnings, and Errors

For displaying diagnostic information about the state of the program, R has three functions. In increasing order of severity, they are message, warning and error.

- message: A generic notification/diagnostic message produced by the message function; execution of the function continues
- warning: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the warning function
- error: An indication that a fatal problem has occurred; execution stops; produced by the stop function

message

"message" concatenates its inputs without spaces and writes them to the console. Some common uses are providing status updates for long-running functions, notifying users of new behavior when you've changed a function, and providing information on default arguments:

```
f <- function(x) {
        message("'x' contains ", toString(x))
        x
}
f(letters[1:5])

## 'x' contains a, b, c, d, e

## [1] "a" "b" "c" "d" "e"
```

It may seem trivial, but when you are repeatedly running the same code, not seeing the same message 100 times can have a wonderful effect on morale:

```
suppressMessages(f(letters[1:5]))

## [1] "a" "b" "c" "d" "e"
```

warning

Warnings behave very similarly to messages, but have a few extra features to reflect their status as indicators of bad news. Warnings should be used when something has gone wrong, but not so wrong that your code should just give up.

```
log(-1)
```

Common use cases are bad user inputs, poor numerical accuracy, or unexpected side effects:

```
g <- function(x) {
        if(any(x < 0))
        {
                warning("'x' contains negative values: ", toString(x[x < 0]))
        }
        x
}
g(c(3, -7, 2, -9))

## Warning in g(c(3, -7, 2, -9)):  'x' contains negative values:  -7, -9

## [1]  3 -7  2 -9
```

As with messages, warnings can be suppressed:

```
suppressWarnings(g(c(3, -7, 2, -9)))

## [1]  3 -7  2 -9
```

There is a global option, warn, that determines how warnings are handled. By default warn takes the value 0, which means that warnings are displayed when your code has finished running. You can see the current level of the warn option using getOption:

```
getOption("warn")

## [1] 0
```

If you change this value to be less than zero, all warnings are ignored:

```
old_ops <- options(warn = -1)
g(c(3, -7, 2, -9))
```

It is usually dangerous to completely turn off warnings, though, so you should reset the options to their previous state using:

```
options(old_ops)
```

Setting warn to 1 means that warnings are displayed as they occur, and a warn value of 2 or more means that all warnings are turned into errors.

You can access the last warning by typing

```
last.warning
```

If 10 or fewer warnings were generated, then this is what happens. But if there were more than 10 warnings, you get a message stating how many warnings were generated, and you have to type warnings() to see them.

error

Errors are the most serious condition, and throwing them halts further execution. Errors should be used when a mistake has occurred or you know a mistake will occur. Common reasons include bad user input that can't be corrected (by using an as.* function, for example), the inability to read from or write to a file, or a severe numerical error.

Antibugging

You may adopt some "antibugging" strategies as well. Suppose you have a section of code in which a variable x should be positive. You could insert this line:

```
stopifnot(x > 0)
```

If there is a bug earlier in the code that renders x equal to, say, -12, the call to stopifnot() will bring things to a halt right there, with an error message.

```
h <- function(x, na.rm = FALSE) {
        if(!na.rm)
        {
                stopifnot(!any(is.na(x)))
        }
        x
}
h(c(1, NA))

## Error:  !any(is.na(x)) is not TRUE

h(c(1, NA),na.rm = T)

## [1]  1 NA
```

1.5 Debugging

For example, say you have the following code:

```r
x <- y^2 + 3*g(z,2)
w <- 28
if (w+q > 0) u <- 1 else v <- 10
```

In the old days, programmers would perform the debugging confirmation process by temporarily inserting print statements into their code and rerunning the program to see what printed out.

```r
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
u <- 1
print("the 'if' was done")
} else {
v <- 10
print("the 'else' was done")
}
```

We would rerun the program and inspect the feedback printed out. We would then remove the print statements and put in new ones to track down the next bug.

An example

```r
printmessage <- function(x){
if (x > 0)
print("x is greater than zero")
else print("x is less than or equal to zero")
invisible(x)
}
printmessage(1)
printmessage(NA)
```

```r
printmessage2 <- function(x) {
if(is.na(x))
print("x is a missing value!")
else if(x > 0) print("x is greater than zero")
else
print("x is less than or equal to zero")
invisible(x)
}
x <- log(-1)
printmessage2(x)
```

This manual process is fine for one or two cycles, but it gets really tedious during a long debugging session. And worse, all that editing work distracts your attention, making it harder to concentrate on finding the bug.

Debugging Tools

The primary tools for debugging functions in R are

- traceback: prints out the function call stack after an error occurs; does nothing if there's no error
- debug: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- browser: suspends the execution of a function wherever it is called and puts the function in debug mode
- trace: allows you to insert debugging code into a function a specific places
- recover: allows you to modify the error behavior so that you can browse the function call stack

Single-Stepping with the debug() and browser() Functions

The core of R's debugging facility consists of the browser. It allows you to single-step through your code, line by line, taking a look around as you go. You can invoke the browser through a call to either the debug() or browser() function. R's debugging facility is specific to individual functions. If you believe there is a bug in your function f(), you can make the call debug(f) to set the debug status for the function f().

This means that from that point onward, each time you call the function, you will automatically enter the browser at the beginning of the function. Calling undebug(f) will unset the debug status of the function so that entry to the function will no longer invoke the browser.

On the other hand, if you place a call to browser() at some line within f(), the browser will be invoked only when execution reaches that line. You then can single-step through your code until you exit the function. If you believe the bug's location is not near the beginning of the function, you probably don't want to be single-stepping from the beginning, so this approach is more direct.

It can become tedious to call debug(f) and then undebug(f) when you just want to go through one debugging session for f(). Starting with R 2.10, one can now call debugonce() instead; calling debugonce(f) puts f() into debugging status the first time you execute it, but that status is reversed immediately upon exit from the function.

Using Browser Commands

While you are in the browser, the prompt changes from > to Browse[d]>. (Here, d is the depth of the call chain.) You may submit any of the following commands at that prompt:

- n (for next): Tells R to execute the next line and then pause again. Hitting ENTER causes this action, too.
- c (for continue): This is like n, except that several lines of code may be executed before the next pause. If you are currently in a loop, this command will result in the remainder of the loop being executed and then pausing upon exit from the loop. If you are in a function but not in a loop, the remainder of the function will be executed before the next pause.
- Any R command: While in the browser, you are still in R's interactive mode and thus can query the value of, say, x by simply typing x. Of course, if you have a variable with the same name as a browser command, you must explicitly call something like print(), as in print(n).
- where: This prints a stack trace. It displays what sequence of function calls led execution to the current location.
- Q: This quits the browser, bringing you back to R's main interactive mode.

Debugging with RStudio

https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio

2 *apply functions

2.1 apply

For sums and means of matrix dimensions, we have some shortcuts. apply is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices

```
str(apply)

## function (X, MARGIN, FUN, ...)
```

- X is an array
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

```
x <- matrix(1:24,nrow=4)
x

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    5    9   13   17   21
## [2,]    2    6   10   14   18   22
## [3,]    3    7   11   15   19   23
## [4,]    4    8   12   16   20   24

rowSums = apply(x, 1, sum)
rowSums

## [1] 66 72 78 84

rowMeans = apply(x, 1, mean)
rowMeans

## [1] 11 12 13 14

colSums = apply(x, 2, sum)
colSums

## [1] 10 26 42 58 74 90

colMeans = apply(x, 2, mean)
colMeans

## [1]  2.5  6.5 10.5 14.5 18.5 22.5
```

The apply function is used for applying functions to the rows or columns of matrices or dataframes. Quantiles of the rows of a matrix.

```
x <- matrix(rnorm(200), 20, 10)
#?quantile
apply(x, 1, quantile, probs = c(0.25, 0.75))

##           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
## 25% -0.0497953 -0.8022563 -0.9165523 -0.9827469 -0.5212556 -0.5021466
## 75%  0.7826299  0.3231492  0.6085441  0.2463501  0.1655246  0.5439360
##           [,7]        [,8]      [,9]      [,10]      [,11]      [,12]
## 25% -1.0290199 -0.97685650 -0.618691 -0.3365819 -0.6432999 -0.7652058
## 75%  0.2339169  0.05407604  1.053643  0.2937521  0.5984581  0.3417121
##          [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
## 25% -0.1956111 -0.7860212 -1.0213136 -0.2800948 -0.7231161 -0.7557842
## 75%  1.1500474  0.6629060  0.3117813  0.6599279  0.9879331 -0.5087939
##          [,19]      [,20]
## 25% -0.7533123 -0.1003522
## 75%  0.5754819  0.7471505
```

Average matrix in an array

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)

##            [,1]         [,2]
## [1,] -0.3606650 -0.049059415
## [2,] -0.2131403 -0.002567112

rowMeans(a, dims = 2)

##            [,1]         [,2]
## [1,] -0.3606650 -0.049059415
## [2,] -0.2131403 -0.002567112
```

2.2 mapply

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
str(mapply)

## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))

## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```r
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

- FUN is a function to apply
- ... contains arguments to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified.

```r
noise <- function(n, mean, sd) rnorm(n, mean, sd)
noise(5,1,2)
```

```
## [1] -2.413031  2.775439  5.719322  3.968982  5.350133
```

```r
noise(1:5,1:5,2)
```

```
## [1] 2.973930 5.866200 4.458258 3.386506 3.949374
```

```r
mapply(noise, 1:5, 1:5, 2)
```

```
## [[1]]
## [1] -3.10921
##
## [[2]]
## [1] 1.155555 1.049458
##
## [[3]]
## [1] -1.128978  1.569946  5.601192
##
## [[4]]
## [1] 5.600113 4.867628 5.636744 2.620906
##
## [[5]]
## [1] 4.973062 5.285809 4.213198 6.402439 3.736101
```

```r
list(noise(1, 1, 2), noise(2, 2, 2), noise(3, 3, 2), noise(4, 4, 2), noise(5, 5, 2))

## [[1]]
## [1] -2.495152
##
## [[2]]
## [1]  1.88575619 -0.08583168
##
## [[3]]
## [1] 2.0571313 2.8158007 0.9964285
##
## [[4]]
## [1] 2.516064 4.168613 5.295537 4.692418
##
## [[5]]
## [1] 6.666417 5.040478 6.835452 3.194508 3.342273
```

2.3 lapply() and sapply()

Each of the *apply functions will SPLIT up some data into smaller pieces, APPLY a function to each piece, then COMBINE the results. lapply takes three arguments: (1) a list X; (2) a function (or the name of a function) FUN; (3) other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list. lapply always returns a list, regardless of the class of the input.

```r
#1
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)

## $a
## [1] 3
##
## $b
## [1] 0.2325889

x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean)

## $a
## [1] 2.5
##
## $b
## [1] 0.7871178
##
## $c
## [1] 0.7734139
##
## $d
## [1] 4.961637

#2
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.5654771
##
## [[2]]
## [1] 0.6108811 0.8745072
##
## [[3]]
## [1] 0.4228842 0.9224485 0.6929324
##
## [[4]]
## [1] 0.6767389 0.3495941 0.2747887 0.4976514
```

```
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 4.890143
##
## [[2]]
## [1] 5.951393 8.216313
##
## [[3]]
## [1] 6.273454 3.727089 2.918946
##
## [[4]]
## [1] 4.590575 9.170664 8.662025 8.199267
```

```
#3
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
lapply(x, function(elt) elt[,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

This dataset contains details of various nations and their flags. More information may be found here: http://archive.ics.uci.edu/ml/datasets/Flags.

```
flags<-read.csv(file="C:\\Users\\XXXHHF\\Documents\\R\\workfile\\flag.csv",header=FALSE)
names(flags)<-c("name","landmass","zone","area","population","language",
"religion","bars","stripes","colours","red","green","blue","gold","white",
"black","orange","mainhue","circles","crosses","saltires","quarter",
"sunstars","crescent","triangle","icon","animate","text","topleft","botright")
```

```
load("C:/Users/XXXHHF/Documents/LYX/8. R programming/Data/flags.RData")
head(flags)

##              name landmass zone area population language religion bars
## 1    Afghanistan        5    1  648         16       10        2    0
## 2        Albania        3    1   29          3        6        6    0
## 3        Algeria        4    1 2388         20        8        2    2
## 4 American-Samoa        6    3    0          0        1        1    0
## 5        Andorra        3    1    0          0        6        0    3
## 6         Angola        4    2 1247          7       10        5    0
##   stripes colours red green blue gold white black orange mainhue circles
## 1       3       5   1     1    0    1     1     1      0   green       0
## 2       0       3   1     0    0    1     0     1      0     red       0
## 3       0       3   1     1    0    0     1     0      0   green       0
## 4       0       5   1     0    1    1     1     0      1    blue       0
## 5       0       3   1     0    1    1     0     0      0    gold       0
## 6       2       3   1     0    0    1     0     1      0     red       0
##   crosses saltires quarter sunstars crescent triangle icon animate text
## 1       0        0       0        1        0        0    1       0    0
## 2       0        0       0        1        0        0    0       1    0
## 3       0        0       0        1        1        0    0       0    0
## 4       0        0       0        0        0        1    1       1    0
## 5       0        0       0        0        0        0    0       0    0
## 6       0        0       0        1        0        0    1       0    0
##   topleft botright
## 1   black    green
## 2     red      red
## 3   green    white
## 4    blue      red
## 5    blue      red
## 6     red    black

dim(flags)

## [1] 194  30

class(flags)

## [1] "data.frame"
```

This tells us that there are 194 rows, or observations, and 30 columns, or variables. Each observation is a country and each variable describes some characteristic of that country or its flag. The lapply() function takes a list as input, applies a function to each element of the list, then returns a list of the same length as the original one.

```
cls_list <- lapply(flags, class)
#cls_list
class(cls_list)

## [1] "list"

as.character(cls_list)

##  [1] "factor"  "integer" "integer" "integer" "integer" "integer" "integer"
##  [8] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [15] "integer" "integer" "integer" "factor"  "integer" "integer" "integer"
## [22] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [29] "factor"  "factor"
```

As expected, we got a list of length 30 – one element for each variable/column. Sapply() allows you to automate this process by calling lapply() behind the scenes, but then attempting to simplify (hence the 's' in 'sapply') the result for you. Use sapply() the same way you used lapply() to get the class of each column of the flags dataset and store the result in cls_vect. If you need help, type ?sapply to bring up the documentation.

```
cls_vect <- sapply(flags, class)
cls_vect

##       name   landmass       zone       area population   language
##   "factor"  "integer"  "integer"  "integer"  "integer"  "integer"
##   religion       bars    stripes    colours        red      green
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##       blue       gold      white      black     orange    mainhue
##  "integer"  "integer"  "integer"  "integer"  "integer"   "factor"
##    circles    crosses   saltires    quarter   sunstars   crescent
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##   triangle       icon    animate       text    topleft   botright
##  "integer"  "integer"  "integer"  "integer"   "factor"   "factor"

class(cls_vect)

## [1] "character"
```

Therefore, if we want to know the total number of countries (in our dataset) with, for example, the color orange on their flag, we can just add up all of the 1s and 0s in the 'orange' column.

```
sum(flags$orange)

## [1] 26
```

Now we want to repeat this operation for each of the colors recorded in the dataset.

```
flag_colors <- flags[, 11:17]
head(flag_colors)

##   red green blue gold white black orange
## 1   1     1    0    1     1     1      0
## 2   1     0    0    1     0     1      0
## 3   1     1    0    0     1     0      0
## 4   1     0    1    1     1     0      1
## 5   1     0    1    1     0     0      0
## 6   1     0    0    1     0     1      0
```

```
lapply(flag_colors, sum)
```

```
## $red
## [1] 153
##
## $green
## [1] 91
##
## $blue
## [1] 99
##
## $gold
## [1] 91
##
## $white
## [1] 146
##
## $black
## [1] 52
##
## $orange
## [1] 26
```

```
sapply(flag_colors, sum)
```

```
##    red  green   blue   gold  white  black orange
##    153     91     99     91    146     52     26
```

```
sapply(flag_colors, mean)
```

```
##       red     green      blue      gold     white     black    orange
## 0.7886598 0.4690722 0.5103093 0.4690722 0.7525773 0.2680412 0.1340206
```

This tells us that of the 194 flags in our dataset, 153 contain the color red, 91 contain green, 99 contain blue, and so on. In the examples we've looked at so far, sapply() has been able to simplify the result to vector. That's because each element of the list returned by lapply() was a vector of length one. Recall that sapply() instead returns a matrix when each element of the list returned by lapply() is a vector of the same length (> 1).

```
flag_shapes <- flags[, 19:23]
lshape <- lapply(flag_shapes, range)
mshape <- sapply(flag_shapes, range)
class(lshape)
```

```
## [1] "list"
```

```
class(mshape)
```

```
## [1] "matrix"
```

```
dim(mshape)

## [1] 2 5
```

When given a vector, the unique() function returns a vector with all duplicate elements removed. We want to know the unique values for each variable in the flags dataset.

```
unique(c(3, 4, 5, 5, 5, 6, 6))

## [1] 3 4 5 6

unique_vals <- lapply(flags, unique)
```

Since unique_vals is a list, you can use what you've learned to determine the length of each element of unique_vals (i.e. the number of unique values for each variable).

```
sapply(unique_vals, length)

##       name   landmass       zone       area population   language
##        194          6          4        136         48         10
##   religion       bars    stripes    colours        red      green
##          8          5         12          8          2          2
##       blue       gold      white      black     orange    mainhue
##          2          2          2          2          2          8
##    circles    crosses   saltires    quarter   sunstars   crescent
##          4          3          2          3         14          2
##   triangle       icon    animate       text    topleft   botright
##          2          2          2          2          7          8
```

2.4 vapply() and tapply()

Whereas sapply() tries to 'guess' the correct format of the result, vapply() allows you to specify it explicitly. If the result doesn't match the format you specify, vapply() will throw an error, causing the operation to stop.

```
vapply(flags, unique, numeric(1))
```

which says that you expect each element of the result to be a numeric vector of length 1.

```
cflags<-vapply(flags, class, character(1))
class(cflags)

## [1] "character"
```

The 'character(1)' argument tells R that we expect the class function to return a character vector of length 1 when applied to EACH column of the flags dataset.

You might think of vapply() as being 'safer' than sapply(), since it requires you to specify the format of the output in advance, instead of just allowing R to 'guess' what you wanted. In addition, vapply() may perform faster than sapply() for large datasets. However, when doing data analysis interactively (at the prompt), sapply() saves you some typing and will often be good enough.

As a data analyst, you'll often wish to split your data up into groups based on the value of some variable, then apply a function to the members of each group. The next function we'll look at, tapply(), does exactly that.

```
str(tapply)

## function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- X is a vector
- INDEX is a factor or a list of factors (or else they are coerced to factors)
- FUN is a function to be applied
- ... contains other arguments to be passed FUN
- simplify, should we simplify the result?

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
x

##  [1] -1.35786813 -0.14913200 -0.61935278  0.02564552 -0.94273179
##  [6]  0.29520945  0.03335189 -0.44569929 -0.19884647 -0.07787020
## [11]  0.07375128  0.06218643  0.65970996  0.76737767  0.59409123
## [16]  0.25572701  0.79466919  0.47844719  0.53884598  0.60606506
## [21]  0.68949798  1.27428513  1.44591688  1.62678667  1.58256284
## [26]  2.25619603  1.37249891  1.17428430  1.35961172 -0.30548013

f <- gl(3, 10)
f

##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3

tapply(x, f, mean)

##          1          2          3
## -0.3437294  0.4830871  1.2476160

tapply(x, f, range)

## $`1`
## [1] -1.3578681  0.2952095
##
## $`2`
## [1] 0.06218643 0.79466919
##
## $`3`
## [1] -0.3054801  2.2561960

tapply(x, f, mean, simplify = FALSE)

## $`1`
## [1] -0.3437294
##
## $`2`
## [1] 0.4830871
##
## $`3`
## [1] 1.247616
```

The 'landmass' variable in our dataset takes on integer values between 1 and 6, each of which represents a different part of the world. The 'animate' variable in our dataset takes the value 1 if a country's flag contains an animate image (e.g. an eagle, a tree, a human hand) and 0 otherwise. If you take the arithmetic mean of a bunch of 0s and 1s, you get the proportion of 1s.

```
table(flags$landmass)

##
##  1  2  3  4  5  6
## 31 17 35 52 39 20

table(flags$animate)

##
##   0   1
## 155  39

tapply(flags$animate, flags$landmass, mean)

##         1         2         3         4         5         6
## 0.4193548 0.1764706 0.1142857 0.1346154 0.1538462 0.3000000
```

The above command apply the mean function to the 'animate' variable separately for each of the six landmass groups, thus giving us the proportion of flags containing an animate image WITHIN each landmass group.

Similarly, we can look at a summary of population values (in round millions) for countries with and without the color red on their flag.

```
tapply(flags$population, flags$red, summary)

## $`0`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    3.00   27.63    9.00  684.00
##
## $`1`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     0.0     4.0    22.1    15.0  1008.0
```

Question: What is the median population (in millions) for countries *without* the color red on their flag?

2.5 split

split takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
str(split)

## function (x, f, drop = FALSE, ...)
```

- x is a vector (or list) or data frame
- f is a factor (or coerced to one) or a list of factors
- drop indicates whether empty factors levels should be dropped

```r
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
split(x, f)
```

```
## $`1`
##  [1] -0.642074795 -1.080305314  0.427325494 -1.873209657 -1.741598699
##  [6]  0.708129654 -1.110219225  0.007001781 -1.098660795  1.914250311
##
## $`2`
##  [1] 0.80514072 0.87066157 0.13616392 0.00710275 0.74310475 0.89354998
##  [7] 0.78449901 0.25709270 0.02149347 0.80960750
##
## $`3`
##  [1]  1.0741530  0.9109033  0.4381801  1.0105033 -0.3620661  1.9046239
##  [7]  1.9507956  2.6108858  1.1765143  0.4832202
```

```r
lapply(split(x, f), mean)
```

```
## $`1`
## [1] -0.4489361
##
## $`2`
## [1] 0.5328416
##
## $`3`
## [1] 1.119771
```

An Example

```r
library(datasets)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```r
s <- split(airquality, airquality$Month)
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
## $`5`
##    Ozone  Solar.R     Wind
##       NA       NA 11.62258
##
## $`6`
##     Ozone   Solar.R      Wind
##        NA 190.16667  10.26667
##
```

```
## $`7`
##      Ozone    Solar.R       Wind
##         NA 216.483871   8.941935
##
## $`8`
##    Ozone  Solar.R     Wind
##       NA       NA 8.793548
##
## $`9`
##    Ozone  Solar.R     Wind
##       NA 167.4333  10.1800
```

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Splitting on More than One Level

```
x <- rnorm(10)
f1 <- gl(2, 5)
f2 <- gl(5, 2)
interaction(f1, f2)
```

```
##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

```
str(split(x, list(f1, f2)))
```

```
## List of 10
##  $ 1.1: num [1:2] 0.899 -1.215
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] -0.243 1.566
##  $ 2.2: num(0)
##  $ 1.3: num 0.0322
##  $ 2.3: num 1.56
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] -0.67711 -0.00528
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] -0.229 0.432
```

```
str(split(x, list(f1, f2), drop = TRUE))
```

```
## List of 6
##  $ 1.1: num [1:2] 0.899 -1.215
##  $ 1.2: num [1:2] -0.243 1.566
##  $ 1.3: num 0.0322
##  $ 2.3: num 1.56
##  $ 2.4: num [1:2] -0.67711 -0.00528
##  $ 2.5: num [1:2] -0.229 0.432
```

2.6 Summary

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- lapply: Loop over a list and evaluate a function on each element
- sapply: Same as lapply but try to simplify the result
- apply: Apply a function over the margins of an array
- tapply: Apply a function over subsets of a vector
- mapply: Multivariate version of lapply