Stat 405/705
Class 6
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Last time
- Using lists to return many features from a function
- The ellipsis argument "..."
- More examples
- Functions used in today's class
- Next time

Last time

- Writing functions.
- Argument checking.
- Environment and scope.

The parts of a function

A skeleton for writing a function

```
f1 <- function( arguments ){
  body
  body
  body
  return( something )
}
```

- Functions are associated with their own environments where they can find variables and other functions.
- If a function can't find a variable in its current environment it will then look for it in the parent environment.

Returning many objects from a single function

```
mostleast <- function(x){
#The first element in the sorted table
least.freq <- sort(table(x))[1]
#The last element in the sorted table
most.freq <- sort(table(x))[length(table(x))]
#Now make the list and return it
return(list(LEAST = least.freq, MOST = most.freq))
}
set.seed(20160328) #Create some data again
a <- as.factor(sample(x=LETTERS,size=1000,replace=TRUE))
mostleast(a)

## $LEAST
##  O
## 27
##
## $MOST
##  T
## 51
```

The three dots (ellipses), ..., special argument

Three dots. From the Wikipedia

> *Ellipsis (plural ellipses; from the Ancient Greek: "omission" or "falling short") is a series of dots (typically three, such as "...") that usually indicates an intentional omission of a word, sentence, or whole section from a text without altering its original meaning.*

One useful feature of R is that we can have have functions use other functions as arguments. We saw the `apply` function in HW 1:

```
my.matrix <- matrix(rnorm(100),ncol=10)
apply(X = my.matrix, MARGIN = 1, FUN = mean)
```

```
##  [1] -0.33910110  0.02502231  0.17363863  0.20993464
##  [5] -0.60397164 -0.26479405  0.10010886 -0.30265925
##  [9] -0.38024362 -0.01646073
```

This `applies` the function `mean` to the rows of `my.matrix`.

The three dots (ellipses), ..., special argument

The help page for the `apply` function:

```
Usage
apply(X, MARGIN, FUN, ...)
```

Note the "..." . These are the ellipses.

The three dots (ellipses), ..., special argument

We'll add in an NA to the matrix and repeat the `apply` function call:

```
my.matrix[1,1] <- NA # make the first element an NA
apply(X = my.matrix, MARGIN = 1, FUN = mean)

##  [1]          NA  0.02502231  0.17363863  0.20993464
##  [5] -0.60397164 -0.26479405  0.10010886 -0.30265925
##  [9] -0.38024362 -0.01646073
```

Notice that there is now an NA in the first element of the returned vector because the `mean` function fails on an NA.
However, there is an argument to the `mean` function: "na.rm" which determines how it should treat NAs. If we say "na.rm = TRUE" it will simply ignore the NAs and return the mean of the other values.
But how do we get the na.rm =TRUE into the mean function when using apply?
That's where the "..." comes in.

The three dots (ellipses), ..., special argument

Using the ellipses argument:

```
#Using the ... argument.
apply(X = my.matrix, MARGIN = 1, FUN = mean, na.rm = TRUE)

##  [1] -0.25296617  0.02502231  0.17363863  0.20993464
##  [5] -0.60397164 -0.26479405  0.10010886 -0.30265925
##  [9] -0.38024362 -0.01646073
```

There is an additional argument to `mean` called `trim` which sets the fraction of data to be trimmed from either end before calculating the mean. This can be passed in through the ... too:

```
#Passing multiple arguments to the "mean" function
apply(X = my.matrix, MARGIN = 1, FUN = mean, na.rm = TRUE, trim = 0.1)

##  [1] -0.25296617  0.04620525  0.22868084  0.23078294
##  [5] -0.61912342 -0.21812883  0.04875700 -0.15690288
##  [9] -0.40481106  0.03649638
```

Ellipses

In summary, the most likely use of the ... construct is in passing arguments down to "interior functions", that is, functions that were called by the function that had the ... argument.

Keeping track of time

- It can be very useful to know how long parts of a program are taking to run.
- There are sophisticated ways to do this called `profiling` and then the more straight forward approach, by peppering your code with timers.
- The `system.time()` function will time a specific bit of code:

```
#How long does it take to invert a 2000 x 2000 random matrix?
system.time(
solve(matrix(rnorm(4000000),ncol=2000))
)

##    user  system elapsed
##    6.65    0.03    6.67
```

Keeping track of time

The R command Sys.time() reports the current time. 

```
Sys.time() #What's the time?

## [1] "2019-02-11 10:51:47 EST"
```

Calling this function twice and returning the difference will give you elapsed time.

```
a <- Sys.time()
print ("I am reading a book for 5 seconds")

## [1] "I am reading a book for 5 seconds"

Sys.sleep(5) #This command suspends R for 5 seconds.

b <- Sys.time()
b - a # The elapsed time

## Time difference of 5.148033 secs
```

Creating informative messages

- If you want to provide informative messages in your programs, the `print` and `paste` commands are very helpful. 
- `paste` takes an unlimited number of comma separated arguments, and outputs a single character string, with all the inputs "pasted" together.
- There is an optional argument `sep = " "` that determines which character is placed between the elements as they are condensed into the single string.

```
var1 <- "Richard"
var2 <- "Penn"
var3 <- 45
out.text <- paste(var1, "works at", var2 ,
                                  "and it takes him", var3,
                                  "minutes to drive to work")
print(out.text)

## [1] "Richard works at Penn and it takes him 45 minutes to drive to work"
```

In-class activity 3: timer functions

- Write two functions: one that initiates a timer and a second that reports elapsed time.
- Your `timer.start` function should take a single optional argument that contains the name of the timer.
- It should return a list with two elements: the name of the timer and the time at which it was started.
- The `timer.stop` function should take the output of the timer.start function as an argument.
- The `timer.stop` function should calculate the elapsed time and report it in a friendly way that includes the name of the timer.
- When you have your timer.start and timer.stop functions written, use them to measure the speed of this R calculation:
  `tmp <- solve(matrix(rnorm(4000000),ncol=2000))`

Keeping track of time

Here are my functions:

In-class activity 3: timer functions

You should be able to run this code, using your own timer functions

```
t1 <- timer.start("Matrix inverse timer")
tmp <- solve(matrix(rnorm(4000000),ncol=2000))
timer.stop(t1)

## [1] "Matrix inverse timer took: 6.7236430644989 secs"
```

In-class activity 3: timer functions

What was your computer speed? http://mathmba.com/timer.html

```
#Step 3. #Read in the database connection library
library(RODBC)
#Step 4. connect to database
my.channel <- odbcConnect("STAT705X", uid="stat705_student",
                                      pwd="$[hOCC*TtKO~")
#Step 5. Run the SQL query using the SELECT SQL command to get data
query.result <- sqlQuery(channel = my.channel,
                  query = "SELECT * FROM TIMINGS")
head(query.result,3) # A look at what's in the result data frame

##   Timing  ID  FirstName Platform
## 1  6.080 103 Richard W.  Windows
## 2 13.780 104      Julio      Mac
## 3 17.815 105      Rahul      Mac

close(my.channel) # Closing the connection
```

Utility functions

- Scenario: you have a big data set (many columns) and want to have a look at the types of variables in the data set.
- In particular, you want to find the dichotomous variables, and see which are most strongly associated with the outcome variable.
- Specifics: the outcome variable is whether or not a website is compromised (Y = 1/0), the predictor variables are features of the web site (some of which are presence/absence types).

```
#Load in the ".Rdata" structure that contains this data frame
#Note the slightly different format than reading in a .csv file
#with "read.table"

load(url("http://mathmba.com/richardw/infection.Rdata"))
dim(infection) # Lots of columns to look at.

## [1] 20000   173
```

Utility functions

- First we need a function that we can apply to the columns of the data frame and see if a column is dichotomous.
- Then, if it is dichotomous, we want to look at the association between it and the outcome (called "Y" in this data frame).
- A natural measure of the strength of association is the "log-odds ratio".
- For the table:

a	b
c	d

the log odds ratio is defined as `log((ad)/(bc))`.

Example

		Y	
		NO	YES
X	NO	4	1
	YES	2	5

- The log-odds ratio is `log( (4 * 5)/(2 * 1)) = 2.303`.
- When the variables are independent the log odds is 0.
- When the log-odds is positive then there is positive association between X and Y.
- When the log-odds is negative then there is negative association between X and Y.

Utility functions

- We will build this analysis using utility functions.
- We will write the function `is.dichotomous` which will determine if a variable is dichotomous.
- `is.dichotomous` will return either a TRUE or FALSE.
- The function `logodds` will take a 2 by 2 table and calculate the log-odds ratio.

Find the dichotomous columns

- By definition, a variable is dichotomous if it only takes on 2 distinct values.
- We will create a `table` with the frequencies of the values in `x` and see if there are only two entries, that is, the `length` of the table is exactly 2.

```
is.dichotomous <- function(x){ #x is the input column
tmp <- table(x) # Make a table of the frequencies of the values in x
if(length(tmp) == 2){
  return(TRUE) #If there are two elements then it is dichotomous
  }            #and we immediately quit and return the result
return(FALSE)  # Otherwise, it is not dichotomous and we return FALSE
}
```

Find the dichotomous columns

The `is.dichotomous` function made use of the fact that once the `return` function is called, the function itself immediately terminates.

```
# Do some checking of the function
is.dichotomous(c(1,1,1))

## [1] FALSE

is.dichotomous(c(1,1,1,2,2,2,2,2))

## [1] TRUE

is.dichotomous(c("B","B","B","A","A","C"))

## [1] FALSE

is.dichotomous(c("B","B","B","A","A","A"))

## [1] TRUE
```

Find the dichotomous columns

Now we are ready to apply our function to each column in the data frame. The `apply` function works on matrices, but on seeing a data frame will coerce it to a matrix. This will not be a problem, if all we want to do is determine if a column has two distinct values.

```
#This code will identify the dichotomous columns
dichot.cols <- apply(infection,2,is.dichotomous)
dichot.cols
```

```
##     Y    X1    X2    X3    X4    X5    X6    X7    X8    X9
##  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
##   X10   X11   X12   X13   X14   X15   X16   X17   X18   X19
## FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
##   X20   X21   X22   X23   X24   X25   X26   X27   X28   X29
## FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
##   X30   X31   X32   X33   X34   X35   X36   X37   X38   X39
##  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
##   X40   X41   X42   X43   X44   X45   X46   X47   X48   X49
##  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##   X50   X51   X52   X53   X54   X55   X56   X57   X58   X59
##  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

Find the dichotomous columns

Again, it is a good idea to check that this has worked:

```
table(infection[,"Y"])   # Yes, dichotomous

##
##     0     1
## 18951  1049

table(infection[,"X3"]) # Yes, dichotomous

##
##     0     1
## 19929    71

table(infection[,"X4"])# Not dichotomous

##
##    1    2    3    4    5    6    7    8    9   10   11   12
## 9265 2311 1238  823  616  458  391  358  263  271  212  202
##   13   14   15   16   17   18   19   20   21   22   23   24
##  167  153  142  120  121  104  110  109   85   72   78   55
```

Create a function for the log odds ratio in a 2 by 2 table

To provide a context we will create a 2 by 2 table:

```
#### This will create the example 2 x 2 table we saw earlier
x1 <- c( rep("NO",5),rep("YES",7))
y1 <- c(rep("NO",4),"YES",rep("NO",2),rep("YES",5))
table(x1,y1)   # Check the table to make sure it is correct

##       y1
## x1     NO YES
##   NO    4   1
##   YES   2   5
```

Create a function for the log odds ratio in a 2 by 2 table

We will have this function work **only** when it is passed two dichotomous variables. The arguments are named x and y and will be two columns from the data frame.

```
logodds <- function(x,y){
  if (! (is.dichotomous(x) & is.dichotomous(y))){ #check arguments
    stop("RW: Both inputs need to be dichotomous!")
  }
  tmp.table <- table(x,y) # This is the 2 by 2 table
  lor <- log( # Here's where the calculation happens
    (tmp.table[1,1] * tmp.table[2,2]) /
      (tmp.table[1,2] * tmp.table[2,1])
  )
  return(lor)
}
```

Create a function for the log odds ratio in a 2 by 2 table

It's time to test the function and make sure it returns what we think it should.

```
#### Test the function
#### It's a good idea to have some test cases for which you know the answer
#### Recall, we created x1 and y1 earlier on
table(x1,y1) #The inputs
```

```
##      y1
## x1    NO YES
##   NO   4   1
##   YES  2   5
```

```
logodds(x1, y1)# Check the  calculation. It should equal log(10)
```

```
## [1] 2.302585
```

Apply the log odds

We now want to apply the log odds function to the relevant columns of the infection data frame. The key command here is apply:

```
result <- #We'll put the answer into "result"
sort( # Find the columns with the highest log odds ratios
  apply(
    infection[,dichot.cols], # Only work with the dichotomous columns
    2, # The function works on columns, not rows
    logodds, # The function we are applying
    y=infection[,"Y"] # But, our log-odds function takes two arguments.
    # We specify the second argument by using the ... slots
  ),
  decreasing = TRUE #The order in which I want the sort
)
```

Looking at the output

Let's have a look at the `result`:

```
result[1:20] # The top 20 variables.

##         Y       X87       X88        X3      X146       X85
##       Inf       Inf       Inf 7.211523 2.895822 2.894920
##      X163      X127       X61       X77       X78      X170
## 2.590158 2.270348 2.201720 2.049216 1.868439 1.831908
##       X14      X162       X73      X101       X89       X79
## 1.796203 1.796203 1.718499 1.691034 1.660507 1.597020
##       X90       X91
## 1.541675 1.540564

table(infection$X3,infection$Y) # have a look at one of the top variables

##
##          0     1
##   0  18950   979
##   1      1    70

# Yes, it is hugely predictive
```

How long did this calculation take?

```
ort <- timer.start("Odds ratio timer") #Start the timer

result <- #We'll put the answer into "result"
sort( # Find the columns with the highest log odds ratios
  apply(
    infection[,dichot.cols], # Only work with the dichotomous columns
    2, # The function works with columns, not rows
    logodds, # The function we are applying
    y=infection[,"Y"] # But, our function takes two arguments.
    # We specify the second argument by using the ...
  ),
  decreasing = TRUE #The order in which I want the sort
)
timer.stop(ort) #Stop the timer

## [1] "Odds ratio timer took: 0.780004978179932 secs"
```

Looking at the output

- In practice you will create a suite of useful functions.
- You store these in their own file.
- Then in your current project code, you can `source` these functions in at the beginning of the program and have them available.

Module summary

Topics covered today include:

- Writing functions
- Function arguments
- Ellipses
- Good practice; have some known test cases to apply your functions against

Next time

- Iteration: logic and flow control

Today's function list

Do you know what each of these functions does?

```
... (ellipses)
paste
system.time
Sys.sleep
Sys.time
table
```