Stat 405/705
Class 5
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Last time
- Prediction in `lm` and `glm`
- Writing functions
- Functions used in today's class
- Next time

Last time

- Linking R to a database
- Analytics engine architecture
- Joining/merging data frames
- Linear and logistic regression in R: the `lm` command

Generic functions

In both the `lm` and `glm` contexts we can use the same set of functions:

- `summary`
- `predict`
- `residuals`
- `plot`

These are so called `generic` functions. They look to see the class of the object that they are operating on, and then call the appropriate specific version of the generic function.

Writing functions

- Functions let you organize your code into re-usable chunks
- Functions have three parts:
  1. The arguments to the function (`formals`)
  2. The R-code that implements the function itself (`body`)
  3. The environment in which the function's variables are found (`environment`)
- The R command to create a function is `function`. Sometimes we write the functions ourselves, often we find someone else's implementation and make use of that (there are probably a couple million functions available in R by now).

Writing functions

The R syntax to write a function:

```r
# A function to calculate the percent difference between two numbers
#
percdiff <- function(first,second){
result <- (second - first)/first * 100 # The calculation itself

#The "sprintf" function gives you complete control over output formatting

#Three places of decimals followed by a % sign
#It will return a "character" so not useful for further calculations!

final.result <- sprintf("%.3f%%",result)
return(final.result) #Explicitly return the result
}
percdiff(first = 5, second = 4) # Now call the function

## [1] "-20.000%"
```

What happened?

- The function named `percdiff` has two "slots" to fill.
- The names of these slots are `first` and `second`.
- The function is written in terms of the names of the slots.
- When the function is evaluated: "`percdiff(first = 5, second = 4)`" the slots are replaced with whatever is in the argument values (5 and 4 in this case) and the result returned from the function.
- Note: If you don't put a `return` statement in the function, then the function will send back the result of the last line of evaluation in the body of the function.
- Suggestion: always include a return statement explicitly.

The pieces of this function

```
body(percdiff) # The body of the function

## {
##     result <- (second - first)/first * 100
##     final.result <- sprintf("%.3f%%", result)
##     return(final.result)
## }

formals(percdiff) # The arguments to the function

## $first
##
##
## $second

environment(percdiff) # The environment in which the function was called

## <environment: R_GlobalEnv>
```

Calling functions

```
#There are many ways to call this function:
percdiff(first = 3, second = 4) # Naming the arguments explicitly

## [1] "33.333%"

percdiff(3,4) # Dropping the arguments in directly, not named

## [1] "33.333%"

percdiff(second = 4, first = 3) # When named, order doesn't matter

## [1] "33.333%"

percdiff(second = 4,3) # Naming one, not the other

## [1] "33.333%"

percdiff(s = 4, f = 3) # Abbreviating the names (particularly dangerous)

## [1] "33.333%"
```

Writing functions

For readability and reliability I strongly suggest using the complete named form:

```
percdiff(first = 3, second = 4) # Naming the arguments explicitly

## [1] "33.333%"
```

Writing functions

A function to find the most frequently occurring level of a factor:

```
mostoften <- function(x){
result <- sort(table(x),decreasing=TRUE)[1]
return(result)
}

# Fake some data to illustrate use of the function
set.seed(20160328)
#The "sample" command below will be the workhorse of Monte Carlo
a <- as.factor(sample(x=LETTERS,size=1000,replace=TRUE))
mostoften(x=a) # the argument named "x", takes on the value "a".

##  T
## 51
```

Default arguments

A function can have default argument values assigned in the argument list. Note the additional argument, n, with the default value 1:

```
mostoften <- function(x,n=1){ #Pick off the top n items, default = 1 item
result <- sort(table(x),decreasing=TRUE)[1:n]
return(result)
}
mostoften(x=a) #Called with the default n = 1


##  T
## 51


mostoften(x=a,n=3) #Called with the n = 3


## x
##  T  W  B
## 51 49 46
```

Checking the arguments

- It usually is a good idea to look at the values being passed into a function, to see if they make sense for the function to evaluate.
- If they don't make sense, then the options are either to `stop` the function, or to give a `warning` message and then coerce the values to something that does make sense.
- It's your choice (and that of other programmers) what to do, so you need to be aware of what the function may automatically do to its arguments.

Stopping the function

I only want this function to evaluate if the argument is a factor. Stopping the function:

```r
mostoften <- function(x,n=1){ #Pick off the top n items, default = 1
#The next line will look at the argument to see if it is a factor variable
#We haven't discussed "if" yet, but will do so soon.
if(!is.factor(x)){ #The "is.factor" function returns TRUE or FALSE
        stop("RW: This function is only meant to work on factors!")
        }
result <- sort(table(x),decreasing=TRUE)[1:n]
return(result)
}
b <- rnorm(100) #Create a numeric variable and try out the function
mostoften(x = b, n = 3)

## Error in mostoften(x = b, n = 3):  RW: This function is only meant to
work on factors!
```

Creating a warning

Making a warning instead:

```
mostoften <- function(x,n=1){ #Pick off the top n items, default = 1
  if(!is.factor(x)){
    warning("RW: This function is only meant to work on factors!
            I am going to coerce it to one.")
    x <- as.factor(x) #Coercion
  }
  result <- sort(table(x),decreasing=TRUE)[1:n]
  return(result)
}
b <- rnorm(100)#Create a numeric variable and try out the function
mostoften(x = b, n = 3)

## Warning in mostoften(x = b, n = 3):  RW: This function is only meant
to work on factors!
##             I am going to coerce it to one.

## x
## -2.51072755618174 -2.11789073001765 -2.09755016836162
##                 1                 1                 1
```

Environment and scope

- **Environment**: a collection of objects like data frames, functions etc.
- R has a nested set of environments. The top-level environment is called the global environment. R-Studio will show the objects in each environment.
- When you create a new function, it has its own environment.
- **Scope**: where and when can objects been seen in a program?
- A function can find variables in its own environment, plus those environments in which it is embedded.
- The consequence is that a function can use variables that are found in the global environment, without them being passed directly into the function.
- I strongly suggest, you do not do this because it can lead to odd behaviour that is very hard to debug!
- Good practice: use arguments for all the variables that will be used in your function. You will have much more predictable results.

Environment and scope

The following type of function works in R (though is NOT recommended):

```
a <- 10
f1 <- function(x){ #This is only a  function of x
x + a
}
f1(x = 5)  #This function finds "a" in the global environment

## [1] 15

a <- 20    #But, some joker changes a, and now
           #the function no longer gives the same answer as before,
           #even though the function call is identical.
f1(x = 5)

## [1] 25
```

The main reason **we** want to understand this, is that it can be the root cause of a lot of bugs.

Returning many objects from a function

- It is quite possible that you want to return a variety of objects from a function, not just a single one.
- The natural way to do this is to create a list inside the function and then to return the whole list.
- We'll create a list that contains the most frequent level of a factor, and the rarest one.

Returning many objects from a single function

```r
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(x = 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 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
```

Module summary

Topics covered today include:

- Writing functions.
- Argument checking.
- Stop and warning.
- Environment and scope.
- The ellipsis argument.

Next time

- More on writing your own functions (examples) (II)

Today's function list

Do you know what each of these functions does?

```
... (ellipses)
body
environment
formals
function
glm
if
is.factor
lm
mean
predict
read.csv
return
sample
sprintf
stop
summary
warning
```