Stat 405/705
Class 7
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Last time
- Logic and flow control structures
- Functions used in today's class
- Next time

Last time

- Writing functions
- The ellipsis argument

Conditional execution

- The key statement here is the `if` statement.
- It can, but doesn't have to be, accompanied by an `else` statement.
- `If`, evaluates a logical condition. If the condition is TRUE, an expression is executed.
- We have already seen it as a way to check if functions have the expected arguments.

```
if(TRUE){
        print("This will be executed")
}

## [1] "This will be executed"

if(FALSE){
        print("This will not be executed")
}
```

Conditional execution

We also saw the `if` statement when we created the `is.dichotomous` function.

```
# The is.dichotomous function made use of the "if" statement
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
}
```

We didn't require an `else` statement here because the function `return` exits the function immediately on being called.

Conditional execution

Here is how we used the `if` statement to check the type of variables being passed into the `logodds` function:

```
if (!(is.dichotomous(x) & is.dichotomous(y))){ #check argument
    stop("RW: Both inputs need to be dichotomous!")
  }
```

Conditional execution

This code will give us an example to work with, a pack of playing cards and a function that identifies aces.

```
# Make a pack of playing cards
CARDS <- paste(
rep(c("ACE","2","3","4","5","6","7","8","9","10","J","Q","K"),rep(4,13)),
rep(c("H","C","D","S"),13),
sep="_")

#The is.ace function. It will accept a vector (hand) of cards:
is.ace <- function(cards){
result <- rep(FALSE,length(cards)) #Fill the result with FALSE
#grep looks for its first argument in its second,
#and returns the location of the matches
result[grep("ACE",cards)] <- TRUE  # Overwrite FALSE with "TRUE"
                                   # if the card is an ace
return(result)
}
```

Conditional execution

The `if(LOGICAL) {Block 1} else {Block 2}` construct.
If the logical variable is TRUE then Block 1 gets executed, otherwise Block 2 gets executed.

```
set.seed(20170406)
#Draw 2 cards from the pack of cards, without replacement
my.hand <- sample(x=CARDS,size=2,replace=FALSE)# "sample" is very useful!

if(any(is.ace(my.hand))){ # any looks to see whether in a logical vector,
                          # any elements are TRUE. If so, it returns TRUE.
print(paste("Yes! I got an ace.
            Here's my hand:", my.hand[1], my.hand[2]))
} else{
print(paste("I didn't get an ace and here's my hand:",
            my.hand[1], my.hand[2]))
}

## [1] "I didn't get an ace and here's my hand: J_S K_S"
```

Be careful with the `else` command

The `if, else` construct: a pain point in R coding!

```
### This fails. R views the first "if" statement as complete
### and is confused by the else starting its own a line.
#if(any(is.ace(my.hand))){ # any looks to see if in a logical vector, any a
#print(paste("Yes! I got an ace. Here's my hand:",
#             my.hand[1], my.hand[2]))
#}
#else{
#print(paste("I didn't get an ace and here's my hand:",
#             my.hand[1], my.hand[2]))
#}
# Make sure that the "else" comes on the same line as the closing
# curly bracket to the "if" command.
```

Checking multiple conditions: the & and | operators

We often find ourselves checking compound conditions:

```
x <- c(TRUE,TRUE,FALSE,FALSE)
y <- c(TRUE,FALSE,TRUE,FALSE)

x & y  # Truth table for "&" (AND)

## [1]  TRUE FALSE FALSE FALSE

x | y  # Truth table for "|" (OR)

## [1]  TRUE  TRUE  TRUE FALSE

# The 'any" function is handy to check an entire vector
# to see if "any" elements are TRUE
any(x)

## [1] TRUE
```

A special if/else for vectors

- The `apply` and `lapply` are very elegant ways of applying a function to margins of an array or elements of a list.
- In theory you could loop around these structures with a `for` statement but it is not as efficient as the `apply` commands.
- In the same spirit as `apply`, if you would like to apply an `if/else` construct to each element of an entire vector without using a for loop there is an idiom to do this.
- It is the `ifelse` construct (all one expression).

```
ifelse(test, if TRUE return this, if FALSE return that)
```

A special if/else for vectors

Example use of `ifelse` construct:

```
# Make a character vector
my.words <- c("This", "sentence", "contains", "many", "words",
              "but", "some", "have", "few","letters")

# For those words with less than 5 characters return "Small word"
# For those words with 5 or more characters return "Big word"
# The return vector will have as many elements as the test condition

ifelse(nchar(my.words) < 5,"Small word","Big word")

##  [1] "Small word" "Big word"   "Big word"   "Small word"
##  [5] "Big word"   "Small word" "Small word" "Small word"
##  [9] "Small word" "Big word"
```

For loops

- We have avoided `for` loops so far this semester, but they have been implicit in the `apply` and `lapply` functions.
- In general, if you can use one of the `apply` family of functions for your calculations, then you are better off than if you had used a for loop.
- The basic use of the for loop:

```
for(i in 1:5){ # "i" will take on the values 1 through 5 within the loop.
print(paste("This is iteration: ",i))
}
```

```
## [1] "This is iteration:  1"
## [1] "This is iteration:  2"
## [1] "This is iteration:  3"
## [1] "This is iteration:  4"
## [1] "This is iteration:  5"
```

For loops

But, you don't have to only iterate over consecutive integers, and you don't have to call the iterator "i":

```
my.words <- c("Here", "are", "some", "text", "strings")
# "word" is a variable in the loop that takes on the values in "my.words"

for (word in my.words){
        print(paste("The word is: [",word, "]",sep=""))
}

## [1] "The word is: [Here]"
## [1] "The word is: [are]"
## [1] "The word is: [some]"
## [1] "The word is: [text]"
## [1] "The word is: [strings]"
```

For loops

Using the `for` construct, we can run a simulation to see how likely (calculate the probability) the top two cards in a shuffled pack are to contain at least one ace.

```
set.seed(20160405) # set the seed for reproducibility
num.its <- 10000 # set the number of iterations
counter <- 0 # A counter that will count how often we get at least one ace
# Note the indentation to improve readability
for (i in 1:num.its) {
  my.hand <- sample(x = CARDS,size = 2,replace = FALSE)# draw 2 cards
  if (any(is.ace(my.hand))) { #Is there at least one ace?
    counter <- counter + 1 # Add 1 to the counter if there is an ace.
  }
}
print(paste("I saw", counter, "hands with an ace, in",
            num.its,"iterations"))

## [1] "I saw 1517 hands with an ace, in 10000 iterations"

#Aside: the exact answer from probability theory is 0.1493213
```

Jumping out of a loop before it is done: `break`

- Sometimes we want to iterate through a loop **until** an event happens.
- Once we see this event then we immediately exit the loop.
- The command to do this is `break`.
- We will estimate out how long on average it takes for an ace to occur in a deck of cards (find the expected waiting time).

Jumping right out of a loop: break

Note the "nested" for loops. When you break from within nested loops you break from the inner loop, not out of the whole loop.

```
num.its <- 10000 # set the number of iterations
results.container <-
  rep(NA,num.its)# A container to store the results of each iteration
for (i in 1:num.its) { #Start the simulation
  shuffled.deck <-
    sample(x = CARDS) # With no extra arguments sample
                          # will randomly permute its input
  for (j in 1:49) {
    #We must find an ace before the 50th card.
    if (is.ace(shuffled.deck[j])) {
      break #We break out of the inner "j" loop as soon as we see an ace
    }
  }
  results.container[i] <- j # We jumped out of the loop, with the value "j"
 }
print(paste("The expected waiting time for an ace is",
            mean(results.container), "cards"))
```

Testing code

When you are testing code that involves simulations, it makes sense to test with a small number of iterations, or else you will waste a lot of time waiting for the code to finish.

Jumping straight to the next iteration in a loop

- There are scenarios, where within a loop, you know that you want to jump immediately to the next iteration of the loop, without executing any more code within the current iteration of the loop. That is, you want to **skip** the remainder of the code.
- The command to do this is `next`. It jumps straight to the **next** iteration.

Jumping straight to the next iteration in a loop

Here's a basic example:

```
for(i in 1:8){
  if(i %% 2 ==0){
    print("Jumping straight to the next iteration")
    next  # Jump out on even numbers
  }
  print(paste("Iteration", i)) # Not executed for even numbers
}

## [1] "Iteration 1"
## [1] "Jumping straight to the next iteration"
## [1] "Iteration 3"
## [1] "Jumping straight to the next iteration"
## [1] "Iteration 5"
## [1] "Jumping straight to the next iteration"
## [1] "Iteration 7"
## [1] "Jumping straight to the next iteration"
```

The `while` loop

- A while loop will continue `while` its condition is TRUE.
- I must say, that I don't often use them, probably because I am usually doing Monte Carlo where the number of iterations is known in advance.
- The idea is that a `for` loop knows the number of iterations in advance, but a `while` loop is useful when the number of iterations is not known a-priori.

The while loop

Keep playing cards until we see an ace:

```
my.hand <- sample(x = CARDS,size = 2,replace = FALSE)# draw 2 cards
while( ! any(is.ace(my.hand))){ #Keep playing until we see an ace
  print ("No ace yet!")
  my.hand <- sample(x = CARDS,size = 2,replace = FALSE)# draw 2 cards
}

## [1] "No ace yet!"
## [1] "No ace yet!"
## [1] "No ace yet!"

print(my.hand)

## [1] "ACE_H" "4_D"
```

The `switch` construct

- If you ever find yourself writing deeply nested if/else structures, then you may prefer using the `switch` function.
- It is not a part of this course.

Module summary

Topics covered today include:

- Conditional execution: the `if`/`else` construct.
- Looping and flow control. The `for` loop.

Next time

- We have all the building blocks in place. Time to write some useful code.
- Monte Carlo simulations.

Today's function list

Do you know what each of these functions does?

```
any
break
else
for
grep
if
ifelse
next
while
```