R Programming: Week 2

Michael Gaffney

May 14, 2014

Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Most control structures are not used in interactive sessions, but rather when writing functions and longer expressions.

Common Structures

- `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 iteration of a loop
- `return`: exit a function

if

- Consists of a required "if" statement, an optional else clause (in R, equivalent to two if statements I think), and optional "else if" clauses
- Can put an assignment inside the structure

```
if (x>3) {
        y <- 10
} else {
        y <- 0
}
```

- Can also assign the structure to an object

```
y <- if(x>3) {
        10
} else {
        0
}
```

for

- for loops take an iterator variable and assign it to successive values from a sequence or vector. Most commonly used for iterating over the elements of an object (list, vector, etc.)
- This example takes i variable and in each iteration gives values 1, 2, 3, ..., 10, then exits.

```
for(i in 1:10) {
        print(i)
}
```

- Many different ways of iterating with for. The following loops are equivalent:

```
x <- c("a", "b", "c", "d")
#create and iterate through integer index sequence
for(i in 1:4) {
        print(x[i])
}

#seq_along takes a vector and creates integer sequence index
for(i in seq_along(x)) {
        print(x[i])
}

#index takes values from vector itself
for(letter in x) {
        print(letter)
}

#same as 1st; curly braces not needed if single expression
for(i in 1:4) print(x[i])
```

Nested for loops

- for loops can be nested, as is common with matrices.

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

#seq_len takes integer length and creates integer sequence
for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

- Nesting beyond 2-3 levels is often very difficult to read and understand.

while

- while loops begin by testing a condition. If it is true, then they exeute the loop body. Once the loop body is executed, the condition is tested again, and so forth.
  - Useful because easy to read
  - Must write carefully, because can result in infinite loops

```
count <- 0       #initiates index variable
while(count < 10) {
        print(count)
        count <- count + 1  #increment index
}
```

- Can test more than one condition
  - Always evaluated from left to right

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

repeat

- Initiates an infinite loop. Only way to exit is to call `break`. Not commonly used, but has uses, e.g. iterative objective function with a tolerance check.
- No guarantee of stopping. At minimum, requires an algorithm guaranteed to converge. Generally hard to predict how long will run for, which is dependent on tolerance.
  - Usually better to set a hard limit on the number of iterations, e.g., with a for loop, then report on whether or not convergence achieved.

```
x0 <- 1
tol <- 1e-3

repeat {
        x1 <- computeEstimate()
        if(abs(x1 - x0) < tol) {
                break
        } else {
                x0 <- x1
        }
}
```

next, return

- next is used to skip an iteration of a loop

```
for(i in 1:100) {
        if(i <= 20) {
                ##Skip the first 20 iterations
                next
        }
        ##Do something here
}
```

- `return` signals that a function should exit and return a given value.

Summary

- Control structures allow control of the flow of an R program
- Infinite loops should generally be avoided, even if theoretically correct
- Control structures mentioned here are primarily useful for writing programs; for interactive work the *apply functions are more useful.

Functions

Introduction

Functions are created using the `function()` directive and are stored as R objects just like anything else. Specifically, they are R objects of class "function".

```
f <- function(<arguments>) {
        ##function body
}
```

- Functions in R are "first class objects", meaning they can be treated much like any other R object
  - Functions can be passed as arguments to other functions (composition?)
  - Functions can be nested, i.e., 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.

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.

  ```
  f <- function(a, b = 1, c = 2, d = NULL) {

  }
  ```

  In addition to not specifying a default value, you can also set an argument value to `NULL`.

- R function arguments can be matched positionally or by name. Usually not a good idea to mess around with the order of arguments, because it's confusing, but it is legal, so the following calls are all equivalent:

```
mydata <- rnorm(100)
sd(mydata)
sd(x=mydata)
sd(x=mydata, na.rm=FALSE)
sd(na.rm = FALSE, x=mydata)
sd(na.rm = FALSE, mydata) #can do because only two arguments
```

- Named arguments are useful on the command line when have a long argument list and want to use the defaults for everything except for an argument near the end of the list
- Named arguments also useful if can remember the name, but not position, of argument, e.g., plotting functions which have tons of arguments.
- You can mix positional matching with matching by name, which is useful with long argument lists. 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. So the following two calls are equivalent:

```
>args(lm)
function(formula, data, subset, weights, na.action, method="qr", model=TR
>lm(data = mydata, y~x, model=FALSE, 1:100)
>lm(y~x, mydata, 1:100, model=FALSE)
```

- Function arguments can be *partially matched*, which is useful for interactive work. Means don't need to type complete name, as long as there is a unique match.
  - Order of operations when matching an argument is:

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

Lazy Evaluation Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

- In this example, the function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

```
f<-function(a,b) {
        a^2
}
f(2)
##[1] 4
```

- In this, example, "45" gets printed before the error is triggered, because b did not have to be evaluated until after print(a). Once the function tries to evaluate print(b), it has to throw an error.

```
f<-function(a,b) {
        print(a)
        print(b)
}
f(45)
##[1] 45
##Error: argument "b" is missing, with no default.
```

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="1", ...) {
        plot(x,y,type=type, ...)
}
```

- Generic functions use ... so that extra arguments can be passed to methods (generic functions don't do anything, except for dispatching methods in OOP)

```
>mean
function(x, ...)
UseMethod("mean"
```

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

```
>args(paste)
function(..., sep=" ", collapse=NULL)  #concatenates a variable number of

>args(cat)
function(..., file="", sep=" ", fill=FALSE, labels=NULL, append=FALSE)
```

- One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched. Can't use positional or partial matching for arguments that come after ..., because no way for R to know if passing to ... or a different arg.

```
>args(paste)
function(..., sep=" ", collapse=NULL)

>paste("a", "b", sep=":")
[1] "a:b"

>paste("a", "b", se=":") #se not partially matched to sep
[1] "a b :"
```

Coding Standards

1. Always use text files (typically ASCII, at least in US and UK) and a text editor, because least common denominator, and can be read by everybody. RStudio does this by default.
2. Indent code. Idea that different blocks should be spaced to right differently to reflect flow.
   (a) Improves readability
   (b) By default, RStudio indents one space, which sucks. Change this in Preferences->Code Editing->Tab Width. At minimum, 4 spaces; 8 spaces is ideal. Select code and hit CTRL-I to indent.
3. Limit width of code / fix line length (80 columns?)
   (a) Prevents lots of nesting and very long functions, which hinder readability
   (b) In RStudio, can be changed with Preferences->Code Editing
4. Limit the length of individual functions. Theoretically can go on forever, but should limit to one basic activity.
   (a) Nice to have entire function on a single page / screen
   (b) Helps debugging

Scoping Rules

Binding Values to Symbol How does R know which value to assign to which symbol? E.g., when run

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

how does R know what value to assign to lm? Why doesn't it give the value of lm that's 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 working at command line and need to retrieve the value of an R obkect, the order is roughly

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, which can be found by calling `search()`

- The *global environment* (the user's workspace) is always the first element of the search list, and the **base** package is always the last.
- Order of packages on the search list matters
- Users can configure which packages get loaded on startup, so cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions, so it's possible to have an object named c and a function named c.

Scoping Rules

- The *scoping rules* for R are the main feature that make it different from S
- The scoping rules determine how a value is associated with a free variable in a function
  - Consider the following function:

    ```
    f <- function(x,y) {
            x^2 + y / z
    }
    ```

    It has two formal arguments–x and y. In the body of the function there is another symbol z, which in this case is called a *free variable*. Free variables are not formal arguments and are not *local variables* (which are assigned inside the function body)

- R uses *lexical scoping* or *static scoping.* A common alternative is *dynamic scoping.*
- Related to the scoping rules is how R uses the search *list* to bind a value to a symbol
- Lexical scoping is particularly useful for simplifying statistical computations.

Lexical Scoping Lexical scoping in R means that

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An *environment* is a collection of (symbol, value) pairs, e.g., x is a symbol and 3.14 might be its value.
- Every environment has a parent environment; an environment may have multiple "children".
  - The only environment without a parent is the empty environment.
- Each package has a namespace, which is like an environment
- A function + an environment = *a closure* or *function closure*

How is the value of the free variable searched for?

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the *parent environment.*
- The search continues down the sequence of parent environments until hitting the *top-level environment;* this is usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until hitting the *empty environment.* If a value for a given symbol cannot be found once the empty environment is searched, then an error is thrown.

Why does this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace.
  - This behavior is logical for most people, and usually the "right thing" to do.
- However, in R (unlike C) you can have functions defined *inside other functions*–in this case the environment in which a function is defined is the body of another function. Can be thought of as *constructor* functions. In this example, another function is returned as its value:

```
make.power <- function(n) {
        pow <- function(x) {
                x^n
        }
        pow
}

>cube <- make.power(3)
>square <- make.power(2)
>cube(3)
[1] 27
>square(3)
[1] 9
```

Exploring a Function Closure What's in a function's environment?

Call `ls()` on `environment(func)` to see what's in the environment, and `get()` on a symbol and environment to get the value of that symbol.

```
>ls(environment(cube))
[1] "n" "pow"
>get("n", environment(cube))
[1] 3
>ls(environment(square))
[1] "n" "pow"
>get("n", environment(square))
[1] 2
```

Lexical vs. Dynamic Scoping Given

```
y <- 10

f <- function(x) {
        y <- 2
        y^2 + g(x)    ##y and g are free variables
}

g <- function(x) {
        x*y               ##y is a free variable
}
```

what is the value of `f(3)`?

- With lexical scoping the value of y in the function g is *looked up in the environment in which the function was defined*, in this case the global environment, so the value of y is 10.

- With dynamic scoping, the value of y is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*–in R this is known as the *parent frame*). So the value of y would be 2.
- When a function is *defined* in the global environment and is subsequently *called* from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping. E.g.

```
>g <- function(x) {
+ a <- 3
+ x+a+y #y is free variable
+}
>g(2)
Error in g(2): object "y" not found
>y<-3
>g(2)
[1] 8
```

- Other languages that support lexical scoping include Scheme, Perl, Python, and Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory.
- All functions must carry a pointer to their respective defining environments, which could be literally anywhere.
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

Vectorized Operations

Makes it easy to write code without looping, etc. Commonly found in computational languages, e.g., MATLAB

- Many operations in R are *vectorized,* meaning they can happen in parallel, making code more efficient, concise, and readable.

```
>x <- 1:4; y <- 6:9
>x+y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE TRUE TRUE
```

```
> x >= 2
[1] FALSE TRUE TRUE TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x*y
[1] 6 14 24 36
> x/y
[1] 0.16666667 0.2857143 0.3750000 0.4444444
```

- Vectorized Matrix Operations are also possible: `* and /` result in element-wise multiplication (division), while `%*%` results in true matrix multiplication.

Dates and Times in R

R has developed a special representation of dates and times.

Dates

- Dates are represented by the `Date` class
  - Stored internally as the number of days since 1970-01-01
- Dates can be coerced from a character string using the `as.Date()` function

```
>x <- as.Date("1970-01-01")
>x
[1] "1970-01-01"
>unclass(x)
[1] 0
>unclass(as.Date("1970-01-02")
[1] 1
```

Times

- Times are represented by the `POSIXct` or the `POSIXlt` class
  - Times are stored internally as the number of seconds since 1970-01-01
  - POSIX is a data standard
  - POSIXct is just a very large *integer* under the hood; useful when want to store times in something like a data frame (think POSIX count)
  - POSIXlt is a *list* underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month. (think POSIX list)

- Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function

```
>x <- Sys.time()  #already in POSIXct format
>x
[1] "2013-01-24 22:04:14 EST"
>unclass(x)
[1] 1359083054
>x$sec
Error: $ operator is invalid for atomic vectors #integer
>p <- as.POSIXlt(x)
>names(unclass(p))   #recall POSIXlt is a list, so names
[1] "sec"  "min"  "hour"  "mday"  "mon"
[6] "year"  "wday"  "yday"  "isdst"
>p$sec
[1] 14.34
```

- `strptime` function is used if dates are written in a different format; check ?strptime for exactly what formatting strings to use to convert

Working with Dates and Times

- There are a number of generic functions that work on dates and times
  - `weekdays:` give the day of the week
  - `months:` give the month name
  - `quarters:` give the quarter number ("Q1",...)
- You can use + and - on dates and times, and do comparisons. Can't always mix different classes when doing so, as in this example:

```
>x <- as.Date("2012-01-01")
>y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
>x-y
Warning: incompatible methods for "-"
Error: non-numeric argument to binary operator
>x <- as.POSIXlt(x)
>x-y
Time difference of 356.3 days
```

  - These operators even keep track of leap years, leap seconds, daylight savings, and time zones.
- as.POSIXct accepts a tz="GMT" argument