Module 2

Nathan Cardoso

Control structures in R

Control structures in R allow you to control the flow of execution of teh program, depending on the runtime conditions. Some common structures are if-else. for, while, repeat, next, break and return.

The if control structure

The general syntax is as follows. The else section is optional. Suppose you want to execute something based on a certain condition but do nothing if the condition is not true; it is possible to use just an if.

```
if (<condition>) {
  ## do something
} else {
  ## do something else
}
```

We can also have multiple if conditiions as follows

```
if (<condition>) {
  ## do something
} else if (<condition2>){
  ## do something diff
} else {
  ## do something diff
}
```

This is a simple example of an if-else.

```
x <- 5
if(x>3) {
  y <- 10
} else {
  y <- 0
}
y
```

```
## [1] 10
```

Another way of doing the same is:

```r
x <- 7
y <- if(x>3) {
  10
} else {
  0
}
y
```

```
## [1] 10
```

The for loop

For loops take an iterator variable and assign it successive values from a sequence or vector, For loops are most commonly used for iterating over elements of an object (list, vector, etc).

```r
for(i in 1:10){
  print(2*i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

This loop takes the i variable and in each iteration of the loop assigns it a value starting from 1. It exits when the vector 1:10 comes to an end.

There are different ways to use a for loop when we want to index the elements in various R objects. All the three loops written above have the same behavior.

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

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

```r
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"
```

We can also have nested for loops. A use-case could be iterating through the elements of a matrix.

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

However, nesting is not advisable beyond 3-4 levels as it becomes difficult to read and understand.

The while loop

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed the condition is tested again and so forth.

```
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
```

While loops can potentially result in infinite loops if not written properly. You have to make sure that the condition that stops the loop will actually occur. It is considered safer to use a for loop that has a definite limit on the number of times it will execute.

We can also test multiple conditions in a while loop

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

Conditions are always evaluated from left to right.

repeat, next and break

Repeat initiates an infinite loop; these are not commonly used in statistical applicaitons but they do have their uses. The only way to exit a repeat loop is to call break.

```
x0 <- 1
tol <- 1e-8

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

The loop above is a bit dangerous because there is no guarantee it will stop. It depends entirely on the result of the computeEstimate() function. It is better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence between x0 and x1 was achieved or not.

next is used to skip an iteration of a loop

```
for(i in 1:100){
  if(i<=10){
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

The return signals that a function should exit and return a given value. It is sometimes used with loops as well.

Functions

Functions are created by using the function( ) directive and are stored as R objects like anything else. In particular, they are R objects of the class "function".

```
f <- function(<arguments){
      ##Do something interesting
}
```

Some important points

- 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

Lets take an example of a simple additive function.

```
add2 <- function(x,y){
          x+y
}
```

In this function, we did not explicitly use return because R will automatically return the last expression. This function had only one expression which is the first / last.

```
add2(4, 7)
```

```
## [1] 11
```

```
add2(-1, 3)
```

```
## [1] 2
```

Arguments in functions

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 the function.
- Not every function call in R makes use of all the formal arguments.
- Function arguments can be missing or have default values.

A slightly more complicated function, that will subset a vector of numbers and return a subset of those numbers that are greater than 10. Here, a vector is the argument.

```r
# A vector for illustrations
myvec <- (c(3,5,11,6,13,19,8))

above10 <- function(x){
  vector1 <- x > 10
  x[vector1]
}
above10(myvec)
```

```
## [1] 11 13 19
```

This can be generalized into a more useful function, with no value hardcoded in it. Now this function has two named arguments; a vector and the integer.

```r
above <- function(x,n){
  vector1 <- x > n
  x[vector1]
}
above(myvec, 7)
```

```
## [1] 11 13 19  8
```

```r
above(myvec, 15)
```

```
## [1] 19
```

The function can be improved by specifying a default value for n to avoid errors if a user forgets to enter the value

```r
above <- function(x,n=10){
  vector1 <- x > n
  x[vector1]
}
above(myvec, 7)
```

```
## [1] 11 13 19  8
```

```r
above(myvec, 15)
```

```
## [1] 19
```

```r
above(myvec)
```

```
## [1] 11 13 19
```

Argument matching

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent.

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

```
## [1] 0.8957324
```

```r
sd(x = mydata)
```

```
## [1] 0.8957324
```

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

```
## [1] 0.8957324
```

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

```
## [1] 0.8957324
```

Even though it is permitted, mixing up the positions of function arguments is not recommended as it becomes more confusing to the person reading or reviewing the code.

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

```r
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 function calls are equivalent.

```r
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 except for an argument near the end of the list. Named arguments can also help if you remember the name of the argument and not its position in ht e argument list.

Function arguments can be partially matched, which is useful for interactive work. The order of operations when given an argument is: Check for exact match of a named argument → Check for a partial match → Check for a positional match.

Lazy evaluation of function arguments

Arguments to functions are evaluated lazily, so they are only considered when needed.

```
f <- function(a,b) {
    a^2
}
f(2)
```

```
## [1] 4
```

```
f(2,4)
```

```
## [1] 4
```

Both the function calls execute without throwing an error because the function never actually uses the argument b. 2 will get positionally matched to a.

```
f <- function(a,b) {
    print(a)
    print(b)
}
##f(30)
```

The '30' got printed first before the error was triggered. This is because the b did not have to be evaluated until after the print(a). Once the function tried to evaluate print(b) it had to throw an error.

The "..." Argument

The ... argument indicates a variable number of arguments taht are 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.

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, recycle0 = FALSE)
## NULL
```

Paste function is used to concatenate different strings together to create one string or a vector of strings. It can take a variable number of arguments depending on how many strings are there. Similar to paste, cat will concatenate a number of strings and paste the result to a file or the console

```
args(cat)
```

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

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

Scoping rules

Symbol binding

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

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

```
## function (x)
## {
##     x * x
## }
```

how does R know what value to assign to lm? lm also exists in the stats package of R.

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 hte command line and need to retrieve the value of an R object, the order is roughly: Search the global environment for a symbol name matching the one requested → Search the namespaces of each of the packages on the search list. The search list can be found by using the search() functions.

```
search()
```

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

The global environment is the workspace; consisting of all the things you've defined or loaded into R. If there is a symbol there that matches the name requested, then it will take that symbol. So in this case, it takes the value of lm from the environment as we have defined it above.

The global environment is equivalent to the user's workspace and is always the first element of the search list and the base package is always the last. The order of the packages on the search list matters! Users can configure which packages get loaded on startup so we 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. R has different namespaces for functions and non-functions

R Scoping rules

The scoping rules in R are what make it different from the original S language.

- Scoping rules determine how a value is associated with a free variable in a function.
- 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 turns out to be particularly useful for simplifying statistical calculations.

Consider the following situation

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

This function has 2 formal arguments x and y. In the body of the function there is another symbol z. In this case, z is called a free variable. The scoping rules of a language determine how values are assigned in free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body). Lexical scoping in R means that the values of free variables are *searched for in the environment in which the function was defined.*

- If the value of a symbol is not found in the environment in which a function is defined, then the search is continued in the parent environment.
- The search continues down the sequence of parent environments till we hit the top-level environment; this is usually the global environment or the namespace of a package.
- After the top-level environment, the search continues down the search list till we hit the empty environment. If the value for a given symbol cannot be found once the empty environment is arrived at, an error is thrown.

What is an environment?

- An environment is a collection of symbol, value pairs i.e. x is a symbol and 10 might be its value.
- Every enviornment has a parent environment; it is possible for an environment to have multiple "children".
- The only environment without a parent is the empty environment.
- A function + an environment = a closure or function closure.

Why is scooping important?

Typically, a function is defined in the global environment, so that the values of free variables are found in the user's workspace. This behaviour is logical for most people and is usually the "right thing to do". However in R you can have function defined *inside other functions.* Languages like C don't let you do this.

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

This function returns another function as its value. make.power returns pow. In pow, x is the named argument while n is a free variable. However, n is defined in the make.power() function. Because make.power() is the environment in which pow() is defined, it will find the value of n in that environment

```
cube <- make.power(3)
square <- make.power(2)
```

Result of make.power(n) is a function. This function will take a value and raise it to the nth power

```
cube(3)
```

```
## [1] 27
```

```
square(3)
```

```
## [1] 9
```

Exploring a function's environment

The ls function can be used to retrieve the environment of a function.

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube))
```

```
## [1] 3
```

Understanding the difference in lexical and dynamic scoping

What would be the value of f(3)

```
y <- 10

f <- function(x) {
      y <- 2
      y^2 + g(x)
}
g <- function(x) {
      x*y
}
```

In the f() function, y and g are free variables. In the g() function y is a free variable. With lexical scoping, he value of y in the function g is looked up in the environment in which the function was defined; so for g() the value of y is 10. With dynamic scoping, the value of y is looked up in the environment in which the function was called. So the value of y will be 2.

```
f(2)     ##Lexical scopimg
```

```
## [1] 24
```

When a function is defined in the global environment, and is subsequently called in the global environment then the defining and calling environment is the same. So both lexical and dynamic scoping give the same results.

Consequences of lexical scoping

In R, all objects are stored in memory. All functions must carry a pointer to their respective defining environments, which could be anywhere.

Application : Optimization

Coding Standards

- Use a Text Editor: Always write your R codein a text editor and save it as a text file. This ensures compatibility and accessibility for others.
- Indent Your Code: Proper indentation is crucial for readability. Indent blocks of code to visually represent control flow. A common recommendation is to use at least four spaces for indentation.
- Limit Code Width: Keep the width of your code to a maximum of 80 columns. This helps maintain readability and prevents horizontal scrolling.
- Limit Function Length: Functions should perform a single task. Avoid making functions too long; ideally, they should fit on one screen. This aids in debugging and understanding the code structure.

Date and Time

Some practice on writing and improving functions

A function to compute the mean of each column in a dataframe.

```
colmean <- function(x){
    cols <- ncol(x)
    means <- numeric(cols)
    for(i in 1:cols){
      means[i] <- mean(x[, i])
    }
    means
}
```

Using the airquality dataset to test the function.

```
colmean(airquality)
```

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

We do not get valid numbers for the firsst two columns. If any of the entries in a row or column are NA, then the result of mathematical operations like sum, mean, etc on those rows or columns will also return NA. A simple solution to this problem is to first eliminate the rows or entries having NA values in the dataset. We can add a boolean argument to the function; if TRUE the NAs will be removed from the dataset. The default value is set to TRUE as this is a common operation.

```
colmean <- function(x, removeNA=TRUE){
    cols <- ncol(x)
    means <- numeric(cols)
    for(i in 1:cols){
      means[i] <- mean(x[, i], na.rm = removeNA)
    }
    means
}
```

In this function, we call the mean on each column. So the NA values for each column (which is interpreted as a vector) are removed. Because its being applied on the column, just the values which are NA in that column get removed rather than the entire row having an NA value.

```
colmean(airquality, TRUE)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```