# R Programming - Week 2

Mike Talley

2/20/2022

## Control Structures

### If-else

We can tell R to do something IF certain conditions are met. The ELSE is optional, but will specify what to do if the conditions are not met.

```
x <- 5

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

#or

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

### For loops

For loops are typically the most common type of loops used. They are most commonly used for iterating over the elements of an object (list, vector, etc). The basic idea is that you have a loop index which is typically called i, but if you have many loops you might say j, k, l, et cetera. And a loop index is going to take the i variable and each time it loops it will give it a value, and then will exit when finished.

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {print(x[i])}
```

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

```
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"
```

```
for(i in 1:4) print(x[i])
```

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

You can also have nested loops, or a loop within another loop. Be careful with this though, as going more than 2-3 levels deep is hard to understand/read. Like the movie inception!

```
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
```

## While Loops

This is the other basic looping function in R. This tests a condition, and if true, executes the loop body. Once it is executed, it will test the condition again, and so forth. They can result in an inifite loop if done incorrectly, so be careful!

```r
## This loop will count up to 10, and when it reaches 10, the condition is then FALSE, so the loop will

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
```

You can also use more than one condition in the test, which is handy. These conditions are ALWAYS EVALUATED FROM LEFT TO RIGHT. The following loop starts with z = 5. If z is between 3 and 10, it will print, and flip a coin. If heads, it adds 2 to z. If tails, it subtracts 2. This will continue until the original conditions are false.

```r
z <- 5

while(z >= 3 && z <= 10) {
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin == 1) { ## random walk
    z <- z + 2
  } else {z <- z - 2}
  }
```

```
## [1] 5
## [1] 7
## [1] 9
## [1] 7
## [1] 9
```

## Repeat

These loops are endless, and are uncommon in statistics, but could have useful applications elsewhere. Algorithms can be dangerous if they never stop, so it is useful to define a set stopping point.

## Next

This is used when you want to skip a portion or certain number of loops. Below is a next function where we skip the first two iterations.

```r
for(i in 1:10) {
  if(i <= 2) {   ## Skip the first 2 iterations until i > 2
    next
  }
  print(i)
}
```

```
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

## Functions

## Creating Our First Functions

```r
## Super simple function to add two numbers together.
add2 <- function(x, y) {
  x + y
}
add2(3, 5)
```

```
## [1] 8
```

```r
## A new function to return values greater than 10
above10 <- function(x) {
  use <- x > 10
  x[use]
}
above10(c(5, 10, 15, 20))
```

```
## [1] 15 20
```

```r
## A function to return values above another number that we choose.
above <- function(x, n) {
  use <- x > n
  x[use]
}
x <- 1:20
above(x, 14)
```

```
## [1] 15 16 17 18 19 20
```

```
## We can make n be the same number unless otherwise specified.
above <- function(x, n = 10) {
  use <- x > n
  x[use]
}
x <- 1:20
above(x)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
above(x, 14)
```

```
## [1] 15 16 17 18 19 20
```

So far these are super simple functions on a single vector. The following will be a function applied to a matrix or dataframe. This one will calculate the mean of each column. We will use a for loop, and each time it loops it will return the mean of the next column.

We need to start by determining the number of columns in the vector, so it knows how many times to loop. We then create the vector 'means', and define it as a numeric = to the value of nc. With that all set up, we can create our for loop. We plug in 1:nc to tell it how many times to loop i. When it loops, it will be creating a vector means[i]. That vector is created by getting the mean of each column, mean(x[, i]).

The removeNA section is an option we can build in to automatically remove any NA values. This would have been a game changer on last weeks quiz.

```
columnmean <- function(x, removeNA = TRUE) {
  nc <- ncol(x)                        ## How many columns does it have
  means <- numeric(nc)                 ## Defining means as a numeric vector
  for(i in 1:nc) {                     ## How many times are we looping? 1:nc
    means[i] <- mean(x[, i], na.rm = removeNA)
  }                                ## When running, what are we looking for?
  means                            ## Return means, so we list it at the end.
}

columnmean(airquality)
```

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

```
columnmean(airquality, FALSE)
```

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

Because our function automatically removes NA, we still have the option to include them, we just need to specify that in the command.

## More Info About Functions

Functions can be passed as arguments to other objects. They can also be nested within other functions. The return value will be the last expression in the body to be evaluated.

Functions also have "named arguments", which could potentially have default values. The "formal arguments" are the arguments that are included in the function definition. formals() will return a list of all the formal arguments. It's important to know that not every function call in R uses all of the formal arguments. Function arguments can be missing or have default values.

### Positional Matching and Matching by Name

You can combine positional matching and matching by name. This is kind of confusing, but generally, a function matches its arguments IN ORDER (formula -> data -> subset -> weights -> etc.). If your function explicity names something, it does not matter what order it is. The formula will remove that from the list, and fill the other arguments IN ORDER. In the following example, we can specify the data and model arguments, so both of these options fill the remaining arguments in order.

args(lm)

lm(data = mydata, y - x, model = FALSE, 1:100) lm(y - x, mydata, 1:100, model = FALSE)

Named arguemnts can be useful in the command line since they will show up in a list of arguments.

### Partial Matching

Function arguments can also be partially matched. This is useful for interactive work, but not so much for programming. R will look for an agument name that matches the partial name you give it. The order of operations that R will follow when given an arguemnt is:

1: Check for exact match to a named argument. 2: Check for a partial match to the named argument. 3: Check for a positial match.

## Defining a Function

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

You can choose to specify a value (a) or not, and you can also choose to define something as NULL, which is a valid entry.

### Lazy Evaluation

Arguments are evaluated "lazily", meaning only as needed. In the following functions, we name two argumetns as (a, b). When we run the function however, we only give it one value. R will positionally match that to (a, ) and choose to leave ( ,b) alone. Because ( ,b) is never evaluated, it does not throw an error.

In the second example, it does throw an error, but only after a was evaluated. Remember, functions evaluate in order, and the last defined argument is what will be returned.

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

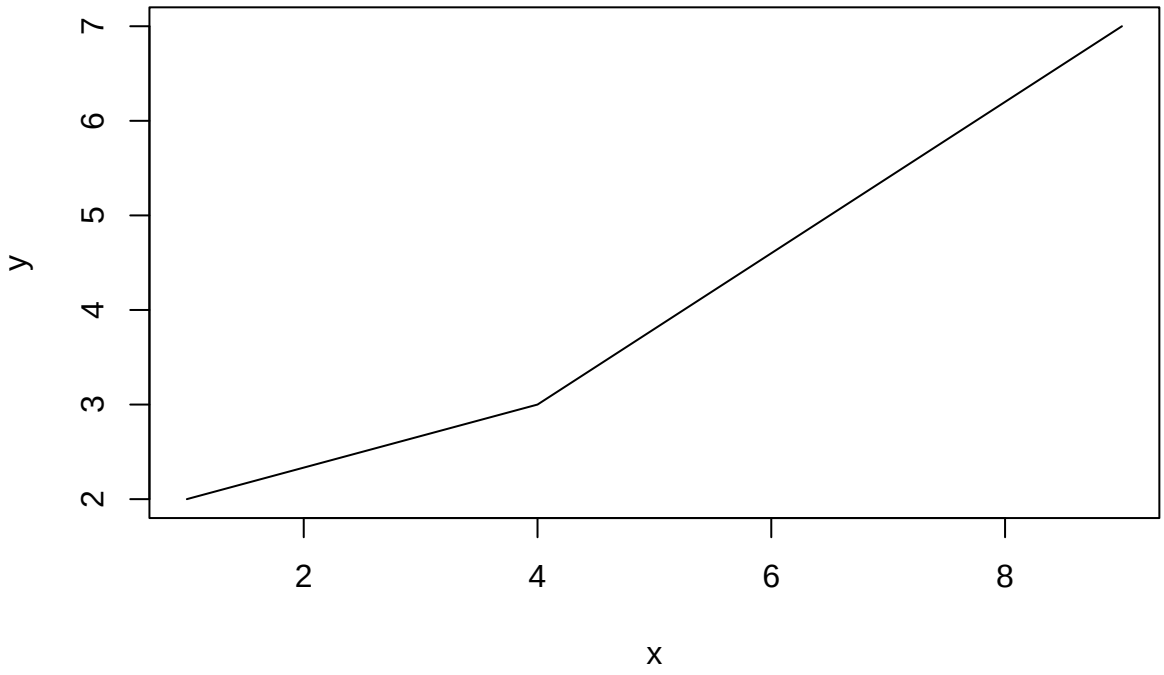
```
## [1] 4
```

```
# f <- function(a, b){
  # print(a)
  # print(b)
# }
# f(45)
```

### The "..." Argument

The ... argument defines a variable number of arguments that are to be passed on to other portions of the function. It's often used to extend a function without having to copy all the arguments each time. In the following example, it passes all the function arguments down to plot, and so on.

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...)
}
myplot(c(1,4,9), c(2,3,7))
```




It can also be used in examples like the paste and cat functions. It has no way of knowing how many entries there will be, so it leaves that argument as ... ### * A caveat to using ... is that anything that comes after it MUST be explicitly defined. Makes sense, because R does not know when the list of arguments starts and ends otherwise.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
args(cat)
```

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

```r
paste("a", "b", sep = ";")
```

```
## [1] "a;b"
```

```r
#versus
paste("a", "b", se = ":")
```

```
## [1] "a b :"
```

## A Diversion on Binding Values to a Symbol

How does R know what values to assign to symbols, especially if we write duplicates of pre existing values. An example would be lm <- function(x) ( x * x ), plus the normal command lm. How does it know which we are reffering to?

1. Search the global environment (your work envir. in R), for a name matching the one requested. So it would return the function we made.
2. Search the namespaces of each of the packages on the search list. To see what it's searching, use search().

The order of packages in the search list matters! If you load a package into R, it automatically gets put into position 2, and everything else will get shifted down a level. It IS possible to have an object and a function named c, occupying different namespaces.

## Scoping Rules

This is the main feature seperating R from the S language. These rules determine how a value is associated with a free variable in a function. R uses what is called "lexical/static scoping", which is the alternative to dynamic scoping. This is specifically usefuly for simplifying statistical computations.

The following function has two formal arguments, (x, y). The body of the function also has a free variable, called (z), because it was not defined in the function header.

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

### Lexical Scoping

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

But what is an environment..?

- A collection of pairs, usually consisting of a symbol (x) and a value (3.14).
- Every environment has a parent environment. It's possible for an environment to have multiple "children" environments as well.
- The only environment without a parent is the empty environment.

- A function + an environment = a closure or function closure.

Order of search for the free symbol... env. -> parent env. -> parent env -> etc. -> top-level (global) env -> search directory -> "empty environemnt". If not found by the empty env, a code is thrown.

Usually, functions are defined in the global environment, and variables exist within the users workspace. This is the "normal" thing to do, which is how other languages work, like C. In R however, you can also define a function INSIDE another function / environment. Function inception!

A downside to lexical scoping is that all objects must be stored in memory, which gets harder as datasets get exponentially larger. Also, functions will then need to carry a pointer to their respective defining environments, making the code a little more complicated.

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

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

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

```
square(2)
```

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

In the above example, the cube command assigns 3 to the value of n, and the square command assigns 2 to the value of n. When you run cube(3), n was already defined inside the make.power command, so cube(3) is defining x.

## Exploring a Function

We can explore what is inside of a function's environment using the list objects command ls(). We can also use the get() command to pull out a specific object.

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

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

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

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

```
ls(environment(square))
```

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

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

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

## Application of Scoping

Optimization routines like optim(), nlm() and optimize() requires you to pass a function where the argument is a vector of fixed paramters. Sometimes it may be useful to keep these parameters fixed.

Super complicated example, but using a function that contains all the info needed and can be changed at will.

```
make.NegLogLike <- function(data, fixed=c(FALSE, FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + B)
   }
}
```

## Coding Standards

1. Always try to use text files / a text editor when writing code. Its the easiest way to keep track of easily usable code.
2. Indent your code. How much is up for debate, but use indents to help someone distinguish what's happening in your code.
3. On that note, limit the width of your code. (80 columns?).
4. Limit length of functions. Ideally, limit each function to a logical seperate tasks. This helps to visually see the whole function, and also helps to find where in the code problems are occuring, rather than somewhere vague in a massive function.

## Dates and Times

Dates are recommended by the "dates" class, and times can be "POSIXct" or "POSIXlt". Dates are stored as the number of days since 1970-01-01, and times are recorded as seconds.

POSIXct: Times are recorded as a massive integer of seconds. This is useful in a dataframe where only one class of vector can be used.

POSIXlt: Times are recorded as a list, including seconds, day of week, year, month, day of month, etc.

There are some generic functions that work on time, like; weekdays(), months(), quarters().

strptime() is a useful function, as it turns written dates (January 10, 2019 10:19) into a POSIXlt class.

### Operations on Dates and Times

You can use operations (really just + and -, as well as logical ones <, >, ==) on dates and times. This allows you to determine time between dates.

These functions can also allow you to specify timezones, and R still keeps track of leap years and other things and adjusts accordingly.