R Notebook: Control Flow (and Loops)

Chelsea Estancona, adapted from Jeff Harden and Brice Acree

In R, we know that there are certain words/phrases that can help set conditions for how R interprets our code. Let's review:

If? If...else? For? While? Ifelse?

We can use these phrases to dictate our 'control flow' - the order in which R evaluates the statements we make. This is our way of telling R under what conditions we expect it to compute/evaluate/etc something (while certain conditions are true) - or how many times to execute an action (for a set number).

We sometimes use if or ifelse in single lines of code, as below:

```
a<-c(0,3,1.9,4)
ifelse(a<2, "yes", "no") #How is ifelse working here? Let's pick that apart.
```

```
## [1] "yes" "no"  "yes" "no"
```

But we can also make more extensive use of these conditions in the form of loops. Loops are our way of iterating in R- over multiple observations, over certain columns, etc.

```
for(i in 1:4){ #set the condition
  print(i) #within the loop, determine the action we're expecting as output
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

It's important to remember that like with function() in the last lab, i (or j, or really whatever) in a for loop is a placeholder- it keeps track of where we are in a process.

Let's look at a 'while' loop:

```
x<-2
while(x < 5) {
  x <- x+1; print(x);
  }
```

```
## [1] 3
## [1] 4
## [1] 5
```

While loops complete an evaluation based on a condition. For this, we do need the object x.

For loops can be a useful tool for editing data:

```
data(mtcars)

mtcars$stuff<-NA #create a new column

for(i in 1:nrow(mtcars)){  #for the first through final rows of a dataset...
  if ((mtcars[i, "mpg"]<20) & (mtcars[i, "disp"]>200)){  #set conditions
    mtcars[i, "stuff"]<-1} #recode
  else{
    mtcars[i, "stuff"]<-0 #do something else for other values
```

```
  }
}

mtcars$stuff
```

```
##  [1] 0 0 0 0 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0
```

We can mix functions and loops. Here, we create a function which takes a matrix as its argument, but then transposes the matrix in a loop (yes, there's a built in function for that, but we're going to do it anyway). Note that in this example, we can also nest loops.

```
mytrans <- function(x) { #function takes as its argument a matrix
  if (!is.matrix(x)) { #But! we can get R to give us a warning message if the object isn't a matrix.
    warning("argument is not a matrix: returning NA")
    return(NA_real_)
  }
  y <- matrix(1, nrow=ncol(x), ncol=nrow(x)) #make a matrix of 1, with rows and columns = to our input
  for (i in 1:nrow(x)) { #begin our for loop
    for (j in 1:ncol(x)) { #and nest another for loop
      y[j,i] <- x[i,j] #fill in y with transpose of x
    }
  }
return(y)
}

# try it
z <- matrix(1:10, nrow=5, ncol=2)
tz <- mytrans(z)
p <- data.frame(z)
mytrans(p) #what happens here?
```

```
## Warning in mytrans(p): argument is not a matrix: returning NA
```

```
## [1] NA
```

Monte Carlo simulation is one example of a situation where you might use control flow features of R. For example, let's use a for() loop to make sure OLS is unbiased. The basic idea is that we are going to create data where we know the true relationship between the variables, estimate OLS, record the result, then repeat. When the loop is done, we will summarize the repetitions.

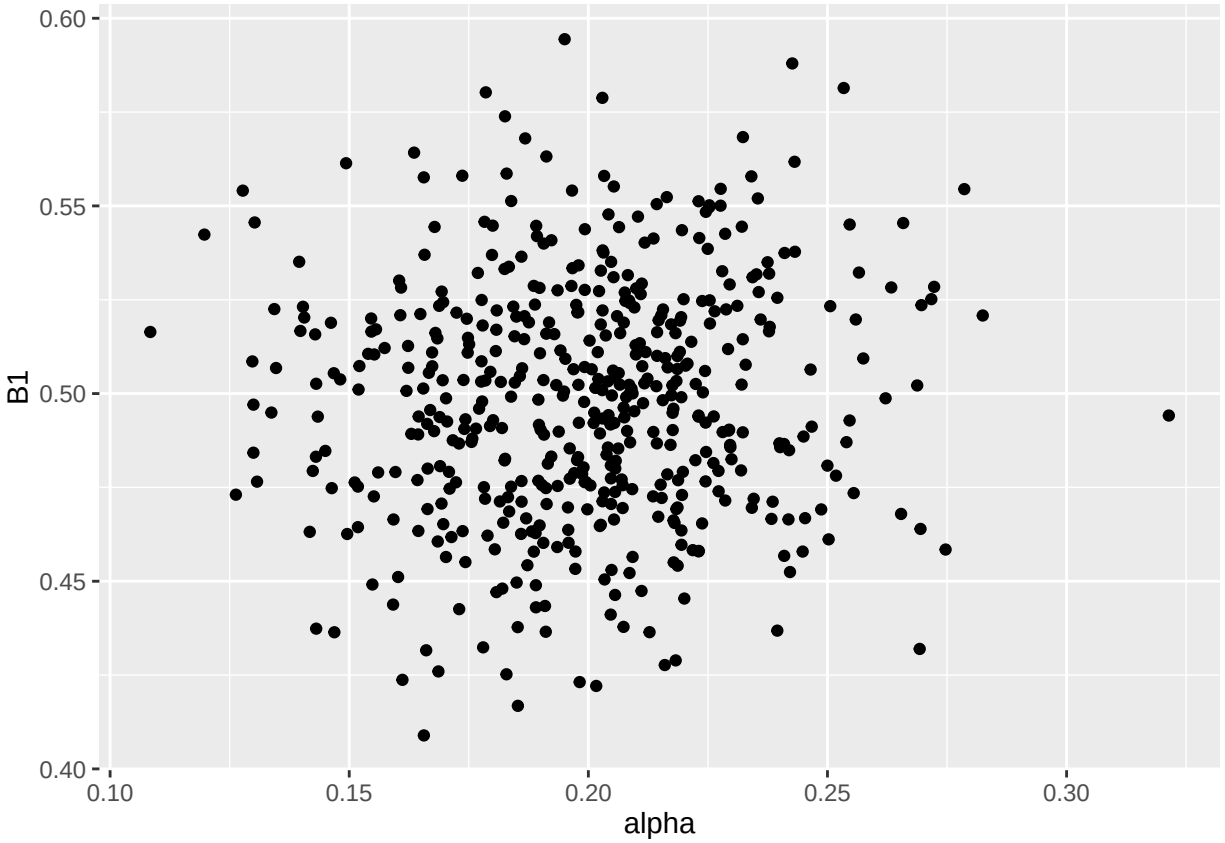
```
set.seed(4328) #RNG
sims <- 500 # 500 simulations
alpha <- numeric(sims) # Empty vector to store the 500 simulated intercept coefficients.
B1 <- numeric(sims) # Empty vector to store the 500 simulated X coefficients.
for(i in 1:sims){ # Start the loop
  X <- rnorm(1000) # Create a sample of 1000 observations.
  Y <- .2 + .5*X + rnorm(1000) # The truth: intercept = .2, B1 = .5, and some error.
  model <- lm(Y ~ X) # Estimate OLS
  alpha[i] <- model$coef[1] # Put the estimate for the intercept in the vector alpha.
  B1[i] <- model$coef[2] # Put the estimate for X in the vector B1.
}

mean(alpha) # Find the expectation of the estimates. You might get slightly different
```

```
## [1] 0.1991556
```

```
#numbers, but they should be close to .2 and .5.
mean(B1)
```

```
## [1] 0.4983075
```

```
library(ggplot2)
qplot(alpha, B1)
```



One additional function to be aware of: the 'apply' function. We can use this to easily 'apply' a given function over a matrix object. Apply is actually a family of functions (lapply, sapply) but we'll cover the basics here. When we can use these functions, they're more computationally efficient- but they aren't always applicable.

```
L<-matrix(rnorm(30), nrow=5, ncol=6)
L
```

```
##            [,1]       [,2]        [,3]      [,4]       [,5]       [,6]
## [1,]  1.0671700  0.6297291  0.03298088 -1.442323 -1.2496955  0.2554921
## [2,]  0.5008091  0.8461883  0.92502828 -3.028041 -1.1904978 -1.1296734
## [3,]  0.8756268  2.4215072  0.61408640 -1.247017  0.3697574  0.2085530
## [4,] -0.7475399 -1.6525459 -0.98673029  1.538207 -0.0993904  0.8844110
## [5,] -1.3343607  0.9975466  0.37273340  1.776904 -0.4027865 -0.2335153
```

```
apply(L, 2 ,sum) #The arguments of 'apply' are the object we're
```

```
## [1]  0.36170536  3.24242531  0.95809867 -2.40227092 -2.57261272 -0.01473262
```

```
#applying a function to, the 'margins' or the way we're applying
#the function, and the function itself.
```

Simulate the linear regression model, as above, changing the number of parameters in the 'true' model. Use

the lm function twice - to estimate the 'correct' model and to estimate a model with omitted variable bias. *Note: change how you sample one of your variables to make it dependent on the other!

Check the means of the resulting alphas and B1s (one from each model). What do you find? Why?