

The goal today is to learn how to use some of R's features as a programming language. You may find that R is really useful for writing simulations. You may also find it helpful when you want to automate something that R is doing. If you have some programming experience, then today will be pretty simple for you- you only need to learn some of the specifics of how to program in R. My goal for today is to teach those people who have never programmed before some of the basics of programming. Two basic tools for thinking about programming are conditional execution (i.e. making a decision and acting on it) and iteration. You may have not realized it, but we have already used these tools. For example, the statement `vector[other.vector > some.number]` is a compact way of saying:

- 1) for each entry in `vector`, examine the value of the same index in `other.vector`
- 2) *if* the value of `other.vector` is greater than `some.number`, *then* return the associated value of `vector`
- 3) otherwise, do nothing

In this example, R has implicitly used both iteration and conditional execution. For each value in the vector, R evaluates a logical expression and does one thing or another based depending on the value of that logical expression. Further, R iterates through the entire vector. These two tools give you a powerful start on programming.

Lets start with a couple examples from my own work.

### A **while** loop:

I have been interested in how natural selection at one locus changes the allele frequencies at other loci through linkage disequilibrium. I had information on the multi-locus genotypes in a population before and after natural selection, and I wanted to see if the change in frequencies of the other alleles could be explained by linkage disequilibrium. The problem was that I couldn't find an explicit solution to the problem, because selection took place over an unknown number of generations. I decided to simulate the problem. The simulation needed to do two things a whole bunch of times:

- 1) Each generation, the frequency of the selected genotypes should be multiplied by a selection coefficient.
- 2) All of the genotype frequencies should then be normalized so they add to one.

The simulation should do this until the frequencies of the selected genotypes reached the observed values.

Repetitive tasks are done using loops. In this case we want to use a `while` loop. The general syntax for a `while` loop is:

```
while(logical condition) {
    expressions;
}
```

This will look pretty unfamiliar to some of you, so lets look at it carefully. When R first encounters the `while` statement, it takes a look inside the parentheses, which will have some sort of logical expression (such as `x > 47`). If this expression is false, then it skips everything and goes on to the next command. (Or waits for you to give it more commands.) If this expression is true, then R will execute the expressions inside the braces. It will then go

back to the beginning, evaluate the logical condition, and either move on or execute the expressions again. In this way, R will keep executing these expressions until the logical condition is no longer true. (If what you do inside the loop does not effect any component of the logical condition, R will be trapped inside an infinite loop.)

So lets look at the code for the specific simulation that I wanted to run. First, I created a vector with the 10 genotype frequencies I observed before the start of selection.

```
> x=c(0.07801418, 0.1418440, 0.06382979, 0.2056738, 0.02836879,
0.2765957, 0.02836879, 0.1418440, 0.02836879, 0.007092199)
```

The last three represented the selected genotypes. In the actual population, these three genotypes represented 71.4% of the population post-selection. So I wanted to run the simulation until these three genotypes reached that frequency.

```
> SelectionStrength=1.001
> while(x[8]+x[9]+x[10]<.714) {
+
+ #apply selection
+
+ x[8]=x[8]*SelectionStrength;
+ x[9]=x[9]*SelectionStrength;
+ x[10]=x[10]*SelectionStrength;
+ #now normalize frequencies
+
+ x=x/sum(x)
+ }
```

So the first thing I did was come up with an arbitrary number to use for the selection strength. The results didn't change much when I varied this value. (You can verify for yourself later on.) Then I did the while loop. Note that the loop when on until the three selected genotypes reached the observed frequencies. Each time through I simply multiplied the selected genotypes by the selection strength, and then renormalized. Make sure that you understand what each line does. ( Anything following a # are comments, and are ignored by R. It is a good idea to comment your code. Comments will enable you to go back and figure out what you were doing.)

```
> x
[1] 0.02711740 0.04930437 0.02218696 0.07149134 0.00986087 0.09614349
[7] 0.00986087 0.57122779 0.11424552 0.02856139
```

## A for loop

In some of my work examining kinetic variation of the enzymes lactose permease and beta-galactosidase, I wanted to measure activity of these enzymes from several strains. The measure of activity is the rate of change of the optical density of a solution with time, which is estimated by a simple linear regression. I also wanted to assess the fit of each of those regressions. The problem was that I had recorded data on 72 different samples, so I obviously didn't want to do those regressions by hand. Fortunately, computers are really good a doing repetitive tasks, if we can figure out how to tell them to do it.

Here is what I wanted to do:

For each sample:      1) Do a regression of optical density on time  
                          2) Record the slope of the regression  
                          3) Record the  $R^2$  of the regression

In this case, we want to use a `for` loop, because we know ahead of time how many times we want to go through the loop. The general syntax of a `for` loop is:

```
for (variable in vector) {
  expression
}
```

Here is what all of those things mean:

`variable` is something like `i` which takes a different value each time through the loop. The first time through the loop, `variable` has the value of the first entry in `vector`. The second time through, `variable` will have the value of the second entry in `vector`, and so on, until we have gone through every entry in `vector`. Each time, we do all of the things listed in `expression`.

Here is a really simple `for` loop:

```
> x=c(29,47,83)
> for(i in 1:3){
  print(x[i]-1)
}
```

You should see that we went through the loop three times, and each time printed the value of `x[i]-1`.

So we can start our more complicated example by importing the data

```
> data=read.table("kinetics.txt", header=T)
```

The first column contains the times at which the measurements were taken, and the rest of the columns refer to the 72 samples. Remember that the goal here was to regress the sample values on time, and record the slope and the  $R^2$ . To use this loop, we will need to find a way to move through the data frame, one column at a time. Fortunately, this is easy to do in a manner analogous to the way R indexes vectors.

Recall that you can get the  $i$ th entry in vector `x` with `x[i]`. You can do likewise with matrices. The only difference is that you need to provide two numbers, one for the column, and one for the row. `x[i,j]` is the entry in the  $i$ th row and  $j$ th column of the matrix `x`. Further more, `x[i,]` refers to the  $i$ th row of the matrix `x`, and `x[,j]` refers to the  $j$ th column. Thus, you could refer to the column with the times as either `data$time` or `data[,1]`.

To get started doing our regressions, we need to have one vector for our estimates of the slope, and one for the  $R^2$ .

```
> slope=vector(mode="numeric", length=72);
> r.sq=vector(mode="numeric", length=72);
```

These calls create vectors of the right length. The semi-colons at the end of the line aren't strictly necessary, but tell R that one command has ended. They are a good idea if you are going to save a bunch of commands in a file and paste them into R for later use. This way, you can give R a whole bunch of commands at a time, and it will execute them all in order.

The loop looks like this:

```
> for (i in 2:73) {
+
+   x<-lm(data[,i]~data[,1]);
+   slope[(i-1)]<-as.numeric(coefficients(x)[2]);
+   r.sq[(i-1)]<-summary(x)$r.squared;
+
+ }
```

We start with the `for` command. We are going to use the indexing variable `i`, and it will take all integer values from 2 to 73. (This is because we have observations in columns 2 to 73. The first column contains our independent variable, time.) In the first line inside the loop, we fit a linear model of the values in one column by the time. We then save the model to the object `x`. In the second line, we save the slope from the model into our vector `slope`. This is a complicated line, so let's unpack it. First, I knew that we could get the slope and intercept from a regression by calling `coefficients` on the model:

```
> coefficients(x)
(Intercept)    data[, 1]
0.0663779482 0.0002376337
```

But we want just the slope, which looks like the second entry. So I tried

```
> coefficients(x)[2]
    data[, 1]
0.0002376337
```

Which takes the vector `coefficients(x)` and returns the second entry. But we don't want the label `data[, 1]`, so we can tell R to ignore all that other stuff by coercing this value into a number with the `as.numeric` call.

```
> as.numeric(coefficients(x)[2])
[1] 0.0002376337
```

Finally, we save this answer to our vector `slope`. When I first wrote this code, I saved it to `slope[i]`, but I found that this gave me a vector where the first entry was NA. This is because our loop starts with a value of 2, so

that our first slope would be saved in the second entry, and our last would not be saved at all. So instead, we save to `slope [ (i-1) ]`.

Finally, we want to save the  $R^2$  of each fit to the vector `r.sq`. When I first wrote this code, I didn't know how to get the  $R^2$  value in an easy to use form, but I knew that it had to be there somewhere- it is printed every time you call `summary` on a `lm` object. So I simply searched the web for information, and found this webpage: <http://www.biostat.wustl.edu/archives/html/s-news/2003-04/msg00081.html>

There are several listservs for help with R, but I have not yet had to email anything to them, because my questions have inevitably been asked and answered before. This web site suggests getting  $R^2$  with `summary(x)$r.squared`, which is just what we need. `summary(x)` actually has a whole bunch of potentially useful information on a linear model. We can see what these are with:

```
> attributes(summary(x))
$names
 [1] "call"          "terms"          "residuals"      "coefficients"
 [5] "aliased"        "sigma"          "df"              "r.squared"
 [9] "adj.r.squared" "fstatistic"     "cov.unscaled"

$class
[1] "summary.lm"
```

Once we save the  $R^2$  term to our vector, we have finished one trip through the loop. R then moves through the expressions in the loop, with the counter `i` taking the next value in the vector, until we have moved through the all the numbers given in the `for` call. At this point, we have extracted the slopes and the  $R^2$  values from our data.

Lets say that what I really want to do with these numbers is to write them to a file. We can write to a file with the function `write.table`, which writes a data frame to a file. To do this, we will bind together `slope` and `r.sq` into a single data frame with the function `cbind`. (At this point, you should know that you can go to the help for information on how this function works.)

```
> z=cbind(slope, r.sq)
> summary(z)
      slope          r.sq
Min.   :-8.060e-05   Min.    :0.03993
1st Qu.: 9.210e-06   1st Qu.:0.81267
Median : 5.932e-05   Median :0.98292
Mean   : 9.983e-05   Mean    :0.83550
3rd Qu.: 9.440e-05   3rd Qu.:0.99773
Max.   : 8.298e-04   Max.    :0.99955
> write.table(z, "regressions.txt")
```

Check the file `regressions.txt` in your working directory to see that it worked. If you wanted to round a few decimal places off those numbers (not something that I would recommend at this point- don't throw away information!), use the function `round`.

**Try combining loops:**

In the first example, I told you that the end result didn't matter much on the choice of selection coefficient. You can test this assertion for yourself by running the simulation a whole bunch of times for different values of `SelectionStrength`. To do this, run a `for` loop, where each time through the `for` loop you execute the simulation with a different selection strength. You are going to need to record the values of the genotype frequencies at the end of one run of the simulation to a column (or row) of a matrix. To do this, I would create the matrix of the proper size at the very beginning. You can do this with the `matrix` command. After you get all of your simulation results, you should plot them to get a sense if my assertion was reasonable.

**Using random numbers:**

Many times simulations are useful for situations where randomness is involved. R works well for random numbers, and has a bunch of probability functions. For example, if you were writing a population genetic simulation that included genetic drift and mutation, you would probably use the `sample` function to do genetic drift by sampling with replacement from one generation to form the next, and using draws from a random binomial (`rbinom`) to figure out how many mutations occur in a specific generation.

**Writing your own functions**

If you do specific kinds of calculations many times, you may want to write your own functions. For example, I when I wanted to calculate linkage disequilibrium of a whole bunch of data sets, I went ahead and wrote a function that did it for me, rather than doing all the necessary commands each time. I wanted a function which would take a pair of vectors (one for each locus) and return several LD statistics. The function looked like this:

```
ld.pair = function(x1, x2){
  table=table(x1, x2); #get the contingency table

  t1=as.numeric(table[1,1]); #get the cell values
  t2=as.numeric(table[1,2]);
  t3=as.numeric(table[2,1]);
  t4=as.numeric(table[2,2]);
  N=t1+t2+t3+t4; #The total number of obs
  p1=(t1+t2)/N; # the freq of an allele
  p2=(t3+t4)/N;
  q1=(t1+t3)/N;
  q2=(t2+t4)/N;
  g1=t1/N;
  g2=t2/N;
  g3=t3/N;
  g4=t4/N;

  D=g1*g4-g2*g3; # one of the stats
  rho=D/(p1*p2*q1*q2)^.5; # another
  X=rho^2*N; # the test statistic
```

```
p.value=1-pchisq(X, df=1); #associated p-value

if(D>0) {          #see Hedrick for explanation
D.prime=D/min(p1*q2, p2*q1)
} else {
D.prime=D/min(p1*q1, p2*q2)
}

results=list(D=D, rho=rho, p.value=p.value, D.prime=D.prime);
return(results);

}
```

Most of this doesn't matter much unless you are interested in calculating LD. What you should take home is the general form. We start with a definition of the call. We tell R that we will call `ld.pair` and give it two vectors. Inside the braces defines what to do with the data we give the function. At the end, I defined a list of output which I thought was important, and then returned it. If someone was to call this function and assign the output to an object, this is what would be saved.

One other thing of note: I used another type of flow control- the `if/else` statement. This bit of flow control works like this: When R encounters the `if` statement, it evaluates the logical expression. If it is true, it executes the code inside the braces. If it is false, it goes on to the `else` statement, if there is one. (We could write code without any `else` statements.) Somewhat annoyingly, `else` needs to be on the same line as the closing brace of the `if` statement preceding it, or R won't recognize it.

So that is the basics of programming in R. Once again, sitting down and playing with R is going to be your best way to learn.