Conditional Statments

We often want to check a conditional statement and then do something in response to that conditional holding, or not holding. Try (writing in RWinEdt):

```
d<-runif(1)
if(d>0.5){cat("\n\n mad benjamins \n\n")}
```

Here is what is happening:

1. `d` is a random uniform number drawn from (0,1)
2. we tell `R` : if `d` is greater than 0.5, then do the operation in the curly braces.
3. the operation in the curly braces is a `cat` statement (short for 'concatenate and print') which will print the contents of the quotation marks.
4. the `\n` are simply line breaks to impose a bit of space between the prompt and our output. Here that means two breaks either side of the text.

Perhaps we need `R` to do something if the condition isn't met. No problem (note the curly braces!):

```
if(runif(1)>0.5){cat("\n\n rock\n\n")}else{cat("\n roll \n")}
```

A simpler, 'hard wired' alternative to this is `ifelse()` which has three arguments: the first is the 'test' (what condition do you want to check?), the 'yes' (what should `R` do if this condition is met?), and the 'no' (what should `R` do if this isn't met?):

```
x<-4;y<-6
ifelse(runif(1)>0.5, x, y)
```

Suppose we want to check more than one condition. Then `&` will come in handy:

```
if(runif(1)>0.5 &rnorm(1)<0){print(x);print(y)}else{(plot(density(rnorm(100))))}
```

Notice the use of `;` to have `R` do a couple of things. Sometimes you'll see the use of `&&`. This means that the second conditional is only checked if the first one is true. This would make no difference to the example above.

Usefully, we can nest `if()` loops. Try the following in `R` -WinEdt and then `source()` it (don't paste it):

```
rm(list=ls())

if((r<-runif(1))>0.5){cat("r is",r,"\n")
    if(r<.6){print("0.5 to 0.6")}else
        if(.6<r&r<.7){print("0.6 to 0.7")} else{print("bigger than 0.7")}} else
            {(print("smaller than 0.5"))}
```

Lots of things to notice here:

1. `rm(list=ls())` simply clears whatever objects are in the memory (should be at the top of all your programs)
2. `(r<-runif(1))>0.5` checks the conditional *and* assigns the number to `r` in one go.
3. `{cat("r is",r,"\n")}` reports back the value actual value of `r` that has been assigned (notice the use of the commas) *outside* the quotation marks.
4. `if(r<.6)print("0.5 to 0.6")` this occurs conditional on `r` being greater than .5, but less than .6
5. `else if(.6<r&r<.7){print("0.6 to 0.7")}` is checked if `r` is greater than 0.5, and it is not less than 0.6.
6. `else{print("bigger than 0.7")}` prints something if the previous statements are false (but `r`>0.5)
7. `else{(print("smaller than 0.5"))}` is the last line of the program and matches the first conditional (i.e. we are now in the case where `r` is less than 0.5)

Writing Functions

The nice thing about `R` is that we can easily write new functions. Here's a simple example that plots graphics (but is not particularly useful):

```
sincosplot<-function(a=0,b=1){
                    if(a>b){stop(" need a<b")}

                    plot(sin(seq(a,b,0.01)), cos(seq(a,b,0.01)))
                    a<<-a; b<<-b
                    }
```

1. the function is called `sincosplot()`. It has features exactly like those of other, 'built in', `R` functions.
2. to call it, just type `sincosplot()`. Notice that it has *default* values for `a` and `b` which simply form the start and ends of the plot axes. If you don't specify any arguments to this function, it will use zero and one.
3. try messing around with the function calls: e.g. try `sincosplot(-1,2)` or try `sincosplot(1,3)`.
4. clearly, the function won't work if `a` is greater than `b`. We build in a warning message using `stop("need a<b")` which stops the operation, and issues a warning.

5. we might want to check what `a` and `b` were for later use. But there is a problem: outside of the function, `a` and `b` do not exist. As a result, we use a 'global assignment' in the function, denoted by `<<-`. This means we can take a look at the object `a` or `b` after the function has completed running.

We can call functions from other functions. Try:

```
genplot<-function(){

                    p<-runif(1)
                    n<-p/2
                    sincosplotter(n,p)

                    }
```

Notice we are giving some parameters, `p` and `n` that are then passed to `sincosplotter()`. Repeat the call a few times to see how it works.

Sampling

Quite often we have to sample from a (posterior) distribution we created. In general, we want to sample `n` values, but we want the sample we produce to be 'weighted'—in the sense that it is proportional to the mass—for each value of our discrete support (say, the thousand values of θ we created for our homework.

We could put together the numerical cdf, but it easier to use `sample` directly.

```
cand.theta<-seq(0,1,0.001)
p<-dchisq(cand.theta, df=15)
s<-sample(cand.theta,1000, replace=T, prob=p)
```

Here, I'm assuming that we first took 1000 values between zero and one, and then we worked out that the posterior,$\Pr(\theta|y)$, was χ^2_{15} (which is unlikely, but anyway). Then we sampled:

1. the object of our sampling was our candidate θ values.
2. we wanted a sample of size 1000
3. we `replace`d the candidates each time we sampled
4. we set `prob=p` which means we are sampling in proportion to the posterior we calculated.

Loops

Loops, like crystal methamphetamine, are easy to use, and easy to abuse. They are the workhorses of much of the `R` that gets written, partly because they are so straightforward to write. This is unfortunate is some ways, because they are often inefficient. Throwing caution to the wind, try the following in `R` -WinEdt:

```
for(i in 1:1000){
                hist(rnorm(100),col=i, main=paste("Picture",i,sep=" "))
                }
```

1. this is a `for` loop: the give away is in the first line. We are saying, 'for' `i` between 1 and 1000, do the thing in the curly braces.
2. here that is: draw a histogram of 100 random normal points, color it with color number `i` and then call it `Picture i` where `i`, of course, is just a number.
3. actually there are not 1000 colors in `R` 's palette, so it recycles some.
4. the `paste` command takes the following syntax:

```
paste(object 1, object 2, what separates object 1 and 2)
```

Of course, from a programming perspective, having this loop run every time you run the program maybe annoying. One option is simply to wrap it into a function. So:

```
homework<-function(){ for(i in 1:1000){
                hist(rnorm(100),col=i, main=paste("Picture",i,sep=" "))
                }
                        }
```

Which means that the `for` loop won't run until we call it via `homework()`

Most of the time, we want to loop through a matrix (or data set) take something from that matrix and put it somewhere else.

First off, create a matrix—initially filled with missing values—to take the fruit of our labors:

```
output.mat<-matrix(NA, nrow=30,ncol=1)
```

Now, suppose we have a matrix like this

```
data.mat<-matrix(runif(900), nrow=30)
```

(i.e. a 30×30 with 900 random numbers). And we want to go row by row taking the mean of the row and outputting it. This would work:

```
for(arthur in 1:nrow(data.mat)){ #loop starts here
                            output.mat[arthur]<-mean(data.mat[arthur,])

                            }
```

Notice:

1. we use `#` to make comments to ourselves (which `R` won't read)

2. we can use pretty much anything for our index: here it is `arthur`
3. the end of the index is the number of rows in `data.mat` (which is 30)
4. now, we are assigning the mean of the $arthur^{th}$ row of `data.mat` to the $arthur^{th}$ row of `output.mat`

So what's the problem? The loop is perfectly accurate, but it is slow, and laborious to code. Much more quickly we could have used:

```
output.mat.2<-apply(output.mat, 1, mean)
```

The `apply()` command works as follows:

1. the first argument is simply the object to be operated on (here it is the `data.mat`)
2. the third argument tells `R` the function to be operated (here we want the `mean`)
3. the second argument tells `R` *how* to operate on the object. Here we want to take the mean of each *row*, so we use `1`. If we want the operation done by column we use `2`.

`apply()` is an example of a 'vectorized' operation we are taking the `data.mat` all at once and performing our operation. Sometimes we need the `list` version (`lapply`), but be careful depending on what type of object you want out at the end.

We may sometimes have cause to place `for` loops within other `for` loops (but generally try to avoid). Here is an example:

```
letters.mat<-matrix(NA, nrow=10, ncol=5)
 for(m in 1:10){
                for(n in 1:5){letters.mat[m,n]<-(letters[n])}
                }
```

Which writes the letters (which are built into `R`) `a` through `e` ten times over into a matrix called `letters.mat`. So, row 1, column 1 of `letters.mat` will be an `a`, row 2, column 1 of `letters.mat` will be an `a`, row 1, column 2 of `letters.mat` will be a `b` and so on. Notice that the way we are indexing the loops matters here. If we put `letters.mat[n,m]` instead of `letters.mat[m,n]`in our code, we'll get the dreaded

```
Error: subscript out of bounds
```

There are alternatives to `for` which are used in different circumstances. Examples are `while` and `repeat` which are often used together.

Other Types of Loops

The syntax for `while` is `while(condition) expression` which means while a particular condition holds, the `expression` will be executed. `repeat(expression)` simply repeats the `expression` operation again and again and again. This type of thing turns up a lot in monte-carlos. Consider the following:

```
n<-1
f.mat<-matrix(NA, nrow=10, ncol=n)

while(n<36){

        repeat{

                f<-rbinom(10,1,0.5)

                if(sum(f)%%2==0){break()}

                }



        f.mat<-matrix(cbind(f.mat[,1:n-1],f), nrow=10, ncol=n)

        n<-n+1
        }
```

Here, while `n` is less than 36, I need to repeatedly sample from a binomial (with a sample size of 10) until an even number of the sample are ones. So, for example, $[1, 0, 0, 0, 0, 0, 0, 1, 1, 0]$ won't do, but $[1, 1, 0, 0, 1, 1, 0, 1, 1, 0]$ is fine. As soon as I get a sample fulfilling my requirements, it should be stored (in `f.mat`). Notice that I have to increment the loop with `n+1` or else the condition `n<36` will be true forever, and the program will never stop.

Just to make this point, consider

```
repeat(cat("\n Arthur, when are we getting married?\n\n"))
```

which won't end until you hit `esc` or `STOP`.