Week 4 Stuff

What have we learned:

1) Data in, data out; data storage in vectors, matrices, data frames

2) Operators

3) Bare-bones function essentials

4) formatting output with **cat**

5) **for** loops

Executing compound blocks of code

You will frequently find yourself executing compound expressions in R - expressions that require more than one line of code. Consider (but do not enter) the following block of code:

```
x <- 1;
y <- 1;

for (i in 1:50)
{
    x <- c(x, rnorm(1, x[length(x)]));
    y <- c(y, rnorm(1, y[length(y)]));
}

plot(y~x, type = 'l');
```

Several major things to notice here:

1) Each statement to be executed sits on a line by itself.

2) Each statement ends with a semicolon (which tells R that it has reached the end of a statement to be executed). Note that **loops** do not end with semicolons, nor do **if** statements, but statements **within** loops end with semicolons.

3) Some statements "belong" to other statements or blocks of code. For example, the statements within the body of a **for** loop 'belong' to that for loop. We indent each such statement to the same level (in the above example, I have used 'tab' for each line within the **for** loop. If you have statements 'belonging' to some other statement, such as a loop, you must indent your code appropriately – it is **wrong** otherwise.

4) Note placement of parentheses, as specified previously.

In case you are interested, the code above simulates a 'random walk' in two dimensions and is just a simplified version of the model you would use if you wanted to simulate the evolution of some trait along a phylogenetic tree. The plot() at the end displays the path of trait values during the simulation.

Escape characters

What if you want to print – literally – the newline character? Or the tab character? The backslash '\' that specifies a newline or a tab ('\n' or '\t') is a special character in most programming languages and is not handled the way normal characters are handled.

Try `cat('\')`

To actually print the backslash, or '\n', or '\t', you need to use an 'escape character'. In R, the backslash \ is an escape character and lets you print special characters. If you want to print a single backslash, you have to do
`cat('\\')`
and to print a literal '\n', you do `cat('\\n')`. Same for tab.

Overview – Week 4

This week we continue our exploration of some powerful programming tools. Two of the most fundamental requirements of a good programming language are:

> (A) the ability to repeat a sequence of statements until some condition is met (looping).
>
> (B) Using a test to decide whether to execute a set of statements (branching, also known as conditional execution of statements).

Last week, we learned the **for** loop, one of two main types of loops used to repeat a sequence of statements depending on some condition.

This week, we will review the **for** loop and learn that other type of loop, the **while** loop. In some ways, the **while** loop is simpler than the **for** loop.

Alas, we are not quite ready for the quasi-omnipotence that comes with mastery of the art of *conditional execution*, so we will have to wait until next week for a full-blown treatment of the topic. Still, we will allow ourselves a taste of this advanced topic on the next few pages, where we briefly review the use of **for** loops.

Here is a more complex function using a **for** loop. This function takes some integer (e.g, a whole number) as an argument, tests whether the number is a prime number, and prints a message depending on the outcome of the test.

Remember that a number *n* is only prime if it cannot be cleanly divided by any numbers 2:(*n* - 1); thus, all such divisions must leave a remainder. This suggests an easy way to test for prime numbers using the modulus operator (%%), which you can think of as the 'remainder' operator (try 5%%2 or 10%%3 from the R prompt if this isn't clear!). Anyway, consider the following function (don't try this in R yet):

```
testPrime <- function(some_integer)
{
     is.prime <- TRUE;

     for (i in 2:(some_integer - 1))
     {
          remainder <- some_integer %% i;
          if (remainder == 0)
               is.prime <- FALSE;
     }

     if (is.prime == TRUE){
          cat(some_integer, " is prime!\n", sep='');
     } else {
          cat(some_integer, " is NOT prime\n", sep='');
     }
}
```

There is a lot going on here, not all of which we have covered so far. Let's look at this on a line-by-line basis.

```
testPrime <- function(some_integer)
```

This is the function declaration line. This creates a function *testPrime*, which takes a single argument, *some_integer*. Note that – as usual – there is nothing special about what you call a function or what you label its arguments. You could do **`isCatFood <- function(songbird)`** and it would work just the same, so long as you specified SOME valid function name and SOME valid argument name. The curly braces on the next line (and at the end of the function definition) enclose all of the **statements** belonging to function *testPrime*.

The basic idea of the function is as follows: we *initialize* a variable *isPrime*, which we assign the value TRUE at the start. We then loop over all integers between 2 and *some_integer* – 1, testing in each case for a remainder (if any integer divided by a smaller number (greater than one) leaves a remainder of zero, then the smaller number is a factor of the larger and the test number cannot be prime, right?)

We use the variable *remainder* to hold the results of the modulus operation. We then use an **if** statement to ask whether the remainder is equal to zero: if so, the number cannot be prime. And **if** the remainder is zero (and ONLY if the remainder is 0), the function executes a little bit of code which sets the variable *isPrime* equal to FALSE. That's all there is to it. If the function evaluates all `length(2:(some_integer – 1))` iterations without finding a factor of *some_integer*, then *isPrime* will still have its original TRUE value.

The final bit is just to print the results to the display, using cat. **If** the number is prime, tell us! If **else** (e.g., NOT prime), tell us! **If** and **else** statements are the subject of class next week and we will not dwell on them now. But they are fairly intuitive.

Let's test whether the number 25 is prime, without yet entering the function in R. Fill out the following information for the first seven iterations of the loop in *testPrime*.

Value of argument to *testPrime*: _____

Iteration	***i***	***isPrime* value at START of iteration**	**Value of *remainder* at end of iteration**	***isPrime* value at END of iteration**
1				
2				
3				
4				
5				
6				
7				

Final value of *isPrime*: _____

TOTAL number of iterations of loop: _____ (how many times are statements executed?)

Now: enter the *testPrime* function in the editor and send it to R. Try it with a few numbers: 5, 6, 25, 32, 47, 49, 12345, ...

The WHILE loop

The basic form of the while loop is

```
while (SOME CONDITION is TRUE){
    execute statements
}
```

The **for** loop runs for a pre-defined number of iterations. In contrast, the **while** loop keeps running until the conditional part of the expression fails. At this point, the loop is terminated.

We might represent the formative years of graduate school with the following **while** loop.

```
while(thesis_idea_sucks){
    get_New_Thesis_Idea();
}
doThesis();
```

Here, the function *doThesis()* is only executed when *thesis_idea_sucks* is FALSE. Thus, you *get_New_Thesis_Idea*, which might or might not be a bad idea. If it is a bad idea, *thesis_idea_sucks* is TRUE, and you stay in the loop; you have to *get_New_Thesis_Idea* yet again. But if you get a GOOD thesis idea, *thesis_idea_sucks* is FALSE, because the idea is good! This means you get to exit the while loop and proceed to *doThesis()*.

The **while** loop might make more sense with a simple example. Try running the following code in your R editor:

```
MAX <- 7;
index <- 0;
while(index <= MAX){
    index <- index + 1;
    cat('index value current: ', index, '\n', sep='');
}
cat('loop is done\nValue of index at ');
cat('loop termination was ', index, '\n', sep='');
```

Here we start out with `index = 0`, but we increment `index` by 1 with each execution of the loop. We included a `cat()` statement to print out the value of index after we have executed the loop statements each iteration. When does the loop stop executing? Note that **`index`** is a **counter variable**, because it keeps track of the number of times the statements within the body of the loop have been executed.

As an aside, there is no special significance to the fact that the final line is written with two `cat()` statements. I just did it to make everything fit neatly between the page margins.

1) Last week, you used a **for** loop to sum all of the numbers from 1:17. Repeat the exercise, but use a **while** loop. You will need to use a counter variable (like *index* seen in the preceding example).

Here's another while loop. In this, we are simulating exponential population growth of a bacterial population that starts with a single individual. Each generation, all individuals in the population instantly divide into two 'daughters' or progeny.

```
MAXPOPSIZE <- 2000;
curr_pop_size <- 1;
popvector <- curr_pop_size;
progeny <- 2;

while (curr_pop_size < MAXPOPSIZE){

     curr_pop_size <- curr_pop_size*progeny;
     popvector <- c(popvector, curr_pop_size);
}
```

Here popvector holds the population size at each generation.

2) Make a plot of the population size as a function of generation time, noting that the first value in *popvector* corresponds to generation $t = 0$.

Let's make it more challenging. In the above example, we could easily figure out exactly how many generations it would take to get to the MAX by some straightforward mathematics, so the **while** loop is in some ways unnecessary. But what about the case where we have a **stochastic** process of population growth? Under this model, each individual does not necessarily leave two progeny; rather, some will leave 0, some will leave 1, others 3 and so on.

We will consider a straightforward extension of the above model where *on average* each individual leaves two progeny, but with a bit of random variation in the number of offspring. One way of modeling such randomness is to assume that progeny numbers follow a particular type of probability model known as a geometric distribution. We don't need to worry about the details of this for now. All you need to know is that R provides a handy function `rgeom()` for simulating numbers from a geometric distribution. The geometric distribution has a parameter p which determines the average number of progeny that we can expect. In the above example, where we have 2 progeny per individual in the population, this parameter is $p = 1/3$. Thus, `rgeom(1, 0.3333)` represents one 'stochastic birth event', where the resulting number is the outcome of a random process which leaves, on average, two progeny in the next generation. You can generate a vector of (random) births from five individuals with `rgeom(5, 0.3333).`

3) Draw 5000 random numbers from a geometric distribution with parameter p = 0.3333. What is the mean of this set of numbers? Plot a histogram of the numbers: this is the distribution of progeny for a population of 5000 individuals that give birth to 2 progeny (on average) each generation. How many progeny did this population give birth to? Repeat the random-draw part several times with the appropriate p parameter to convince yourself that each individual is in fact giving birth to an average of two progeny.

Now to do some real stochastic simulation. We can easily model population growth where there is random variation in the number of progeny per generation by realizing that `rgeom(curr_pop_size, 0.33333)` gives us the distribution of the number progeny per individual in a population of size `curr_pop_size` (note that if you repeat the simulation, you'll get a slightly different distribution due to the stochasticity of the birth events). The result is thus a vector of progeny numbers, and we can sum it to get the total number of progeny. (We are assuming in this example that only progeny survive to the next generation: individuals who have had the opportunity to give birth die at the end of each generation).

```
MAXPOPSIZE <- 5000;
curr_pop_size <- 8;
popvector <- curr_pop_size;
set.seed(1);

while (curr_pop_size < MAXPOPSIZE){
     babies <- rgeom(curr_pop_size, .33333);
     curr_pop_size <- sum(babies);
     popvector <- c(popvector, curr_pop_size);
}
```

4) How many generations does it take to exceed the MAXPOPSIZE? Repeat for 10 replicates, using 1:10 as the seed for the random number generator (eg., for run 5, use set.seed(5)) and record the number of generations required to exceed MAXPOPSIZE. What is the mean number of generations required to exceed MAXPOPSIZE across the 10 replicates?

5) Is a **for** loop appropriate for the above simulation? Why or why not?

6) Based on your analysis of the geometric distribution of progeny, is it necessarily the case that population size will always increase? Why or why not?

7) Are you impressed with how easy it is to simulate stochastic population growth? You should be – if you understand the nuts and bolts of this example, you are pretty close to gaining the power to simulate genetic drift, null models of ecological communities, trait evolution via Brownian motion, etc in similar fashion. R is so *rad* that we don't even need to use `babies` – we can cut the middleman and just do
`curr_pop_size <- sum(rgeom(curr_pop_size, 0.3333))` and it works just as well!

Note: if you have ever wondered what a Markov chain is, the preceding simulation is an example of one!

8) The following code is supposed to print the squares of integers from 1 to 10. What is wrong with it?

```
for (i in 1:10);
{
    cat(i, ' squared is ', i^2, '\n', sep='');
}
```

9A) While doing fieldwork in some faraway land, you and a field assistant pick up a parasite that grows exponentially until treated. Your case is more severe than your assistant's: you have picked up 400 parasites, and your assistant has only picked up 120. However, your tough immune system is better able to deal with the parasites, and the parasite population only grows by 10% in your body each day, compared with 20% per day in your assistant.

Write code that uses a **for** loop to calculate the number of parasites in your body and your assistant's body over the next 30 days. Then, using these vectors of parasites at each time point, draw a single plot of parasite load as a function of time for 30 days. Now make a similar plot, but do *log-transformed* parasite loads as a function of time.

Plot hint: You may need to refresh your knowledge of commands *xlim*, *ylim*, and the function *points( )*.

9B) Assume each parasite has a fixed mass of .000001 kg. Assuming no constraints on growth of the parasite population in your assistant, how long before she harbors a mass of parasites equal to her body mass (55 kg)? Use a **while** loop to model this: exit the loop when the mass of parasites exceeds her body mass and be sure you can recover the number of days required to acquire such a large parasite load.

Review questions to keep you on your toes (also homework!).

10) Write an R script "format.R" which, when sourced, reproduces exactly the following output on your screen (don't worry about the font or color...):

```
        this is tab-indent. now i will show you
the new-
line character: \n

and the backslash is \

this formatted output stuff is slick!!
```

Even more dataframe manipulation

11) Create a dataframe 'lizards' by reading in 'skinks.txt', which you used last week. You will frequently encounter situations in your research where it is advantageous to use the loop statements we learned today to repeat a particular analysis or statistic on selected subsets of a dataframe. For example, let's suppose that we want to sequentially compute the mean snout-vent-length ("snout_vent") of each species of skink in 'lizards'. We can easily envision a situation where, with a large number of species, this would prove exceedingly tedious to do manually.
Rather, we might do:

```
for (i in 1:length(vector of species names))
{
        statement 1: get relevant subset of data frame 'lizards'
        statement 2: get species mean value, store results
}
```

First, you will need to get a vector of species names. (hint: what can the function `unique`() do for you?). Note that you will need to *initialize* a vector to store the mean snout_vent lengths for each species!

***Hint: remember what you learned about the **na.omit()** function. There are some hidden NAs tucked away in the data (this is real data!).

12) Using the looping construct described above, create a new vector of the largest (snout_vent) sizes recorded for each of the *n* species. Yes, there is a 'shortcut' function that finds the maximum of a set of numbers, but I am going to make you guess. R is a fairly intuitive language and I am confident that you can do this.

Use 'names()' and species_name_vector to assign names to the elements of the vector.

13) Using a loop, make another vector that contains the number of sites ('site') at which each species was found. Make sure this vector has names, as above. Write a line of code that extracts the species found at the maximum number of sites.

14) Write a single line of code that extracts the name of the species with the largest snout-vent length, using the original lizards dataframe.