R Programming
Week 3 : Intro to Loops

Reminder: Some of the exercises below require you to have mastered (1) the use of the `cat` function, and (2) the use of the `source` function.

Loopy stuff: <u>for</u> loops

Loops are one of the most important programming tools you will learn. The first loop we will learn is a **for** loop. A **for** loop runs for a certain number of steps, which <u>you define before any statements in the loop are executed</u>.

The basic syntax is

```
for(some sequence of steps)
{
   execute some statements
}
```

The sequence of steps is determined by the *loop* variable, typically *i*. So if, in the above example, suppose we have

```
for(i in 1:25)
```

in place of "some sequence of steps". This means that *i* will start at 1, R will execute some statements; *i* is incremented to *i* = 2 and statements are executed again; *i* is incremented to *i*=3 and statements are executed; and so on, until *i* = 25, at which point the loop executes the set of statements for the last time.

Always execute **for** loops from the editor or by sourcing an R file. Never type them in at the R prompt.

You can use a **for** loop to produce formatted output. For example, try

```
for (i in 1:10){
   cat(i, sep='');
}
```

Note the use of curly braces to enclose any statements involving a **for** loop. There are some circumstances where you don't need these, but for now, you should put ALL expressions belonging to a **for** loop in curly braces.

And you can easily modify the expression to produce a variety of outputs.

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

And cat() can even handle complex arguments, like

```
for (i in 1:10){
    cat(sqrt(i), sep='\n')
}
```

What is sqrt()?

Note that your loop variable *i* does not have to start at 1. Nor does it have to increase. You could easily do

```
for (i in 5:9){
    cat(sqrt(i), sep='\n');
}
```

or

```
for (i in -13:-28){
    cat(i, sep='\n');
}
```

Keep these straight: Be sure you distinguish between curly braces {}, parentheses ( ), and square brackets [ ]. The **brackets** are used to access elements of vectors, matrices, and dataframes. The **parentheses** are used to specify *arguments to functions* or *conditions for executing loops*. Finally, the curly braces enclose *statements to be executed*, either by a function or within the body of a loop.

3a) Modify one of the **for** loop statements above so that it runs from 1:9 and prints the following output to the R console:

```
*
*
*
*
*
*
*
*
*
```

3b) Modify your for loop so that it prints 10 asterisks, with each asterisk separated by exactly one pound sign (#), with no spaces or new line characters.

You will need to modify some arguments to the **cat()** function.

One of the most useful features of **for** loops is the ability to modify variables within the **for** loop. Consider the following bit of code:

```
x <- 2;
for (i in 1:4){
     x <- x^2;
}
```

Note that we start out with variable x = 2. It enters the for loop with a value of 2, but after the first iteration, it is now 2^2, or 4. But when it enters the loop the second time (i = 2), its value is still 4, but when it exits the loop, it is equal to 16. Thus, we have the following table:

Value of **x** before loop begins: 2

i	value of x at beginning of iteration *i*	value of x at end of the i'th iteration
1	2	4
2	4	16
3	16	256
4	256	(256^2) = 65536

Can you follow what is happening to the value of **x**? Don't use your computer – think about it and draw it out if you have to.

The next three homework problems are to be done by hand – no **R** allowed! Look at the **for** loop for each and fill in the table. In each case, I am asking you to determine what the value of some variable is at the START and END of each loop for each iteration, as well as the value of the loop variable *i*.

4)

```
dogs <- 10;
for (i in 1:5){
    dogs <- dogs + 1;
}
```

Initial value of *dogs*: ________

i	value of *dogs* at start of iteration *i*	value of *dogs* at end of iteration *i*

5)

```
meatloaf <- 0;
for (i in 5:9){
    meatloaf <- meatloaf - i + 1;
}
```

Initial value of *meatloaf*: ___________

i	value of *meatloaf* at start of iteration *i*	value of *meatloaf* at end of iteration *i*

Final value of meatloaf: ______________

6)

```
bubbles <- 12;
for (i in -1:-4){
    bubbles <- i;
}
```

Initial value of *bubbles*: _________

i	value of *bubbles* at start of iteration *i*	value of *bubbles* at end of iteration *i*

If these exercises seem trivial, they are! But you will find yourself working through loop logic time and time again, so we need to begin somewhere.

Using for loops with vectors

In the above examples, we used i directly in several mathematical operations. But it is more common to loop over elements of a vector to accomplish some particular task. Here we will construct a loop which runs over a vector of names and prints something senseless:

```
nameVector <- c("Charlie", "Helga", "Clancy", "Matilda", "Jones");

for (i in 1:length(nameVector)){
    cat("Nice to meet you,", nameVector[i], "\n");
}
```

Try it. Understand what is going on here? In this case, the iteration variable i is only used to access elements of the vector **nameVector**, which is passed as an argument to **cat()**, along with several character strings.

Suppose you need to take some vector and perform a common operation on each element of the vector, storing the results in a new vector. If we have a vector of financial assets for a set of individuals and we wanted to compound interest annually at a rate of 5.25% to each, we could do it like this.

```
#current financial holdings in thousands of dollars
# of a random sample of cornell grad students
bankAccounts <- c(10, 9.2, 5.6, 3.7, 8.8, 0.5);

interestRate <- 0.0525;

for (i in 1:length(bankAccounts))
{
     compounded[i] <- interestRate*bankAccounts[i] + bankAccounts[i];
}
```

Try this.

Oh. That didn't work. This failed because you are trying to assign a value to element i of a vector *compounded*, which doesn't exist. This is a very important point: you can always create variables and objects from scratch, but you can never try to assign values to elements of an object/vector/etc unless the object already exists. We will repeat the loop, but using rep(0, length(bankAccounts)) to **initialize** a vector of length 10 with values of 0. **Initializing variables** is a fundamental concept, so put this on the list of things to remember.

Repeat the loop above, but initialize the vector compounded before you try to use it in the loop:

```
compounded <- rep(0, length(bankAccounts));
```

7) Pretend that the function **sum**() does not exist. If *sumThis* is a vector of numbers, provide code that sums all of the elements of *sumThis*, incrementally. Do this using a **for** loop.

Do this with the following vector of numbers: (you should also check your results somehow):

```
set.seed(1);
sumThis <- rnorm(10);
```

8) Using a vector of all letters (*hint: what is *letters* in R?) and a **for** loop, do the following:

(i) print the letters "a" through "g" side by side, with no space between them; then

(ii) print the letters "a" to "g" on separate lines, in reverse order (g, f, e….a).

(iii) A bit trickier: Print the letters "a" to "g" separated by a space

(iv) Print letters "a" to "g" separated by "xx"

You can do these without a **for** loop, but I want you to do this the "hard" way (using for loop) here to get more practice with **for** loops.

9) Write a function "beerBottles" that takes an integer **x** as an argument (actually, I have already done this part for you). When you execute the function, `beerBottles(5)` , for example, you should have get the following output on your screen:

```
5 bottles of beer on the wall
4 bottles of beer on the wall
3 bottles of beer on the wall
2 bottles of beer on the wall
1 bottles of beer on the wall
```

Here is the template function. Fill in the relevant bits for SOME CONDITION and EXECUTE SOMETHING.

```
beerBottles <- function(x)
{
    for(SOME CONDITION)
    {
        EXECUTE SOMETHING
    }
}
```

*Hint: loops can run in reverse.

10) Fun with Fibonacci: The Fibonacci numbers are a famous series that run from 0 to infinity. The first two numbers in the series are 0 and 1, but all other numbers in the series are formed by summing the two previous elements of the series. Thus, if FIB is a vector corresponding the Fibonacci series, FIB[1] = 0 and FIB[2] = 1, but FIB[3] is equal to FIB[1] + FIB[2]. Likewise, FIB[4] = FIB[2] + FIB[3]. In general, for any element *n* of the series greater than 2, we can calculate the corresponding Fibonacci number as

```
FIB[n] <- FIB[n-1] + FIB[n-2]
```

Use a **for** loop to make a new vector containing the first 17 elements of the Fibonacci series. At each iteration, your loop should print something like "`Fib number 5 is 3`" Thus, you need to start by initializing a vector of length 17. You'll need to assign the first and second elements of the series before you begin the loop. Remember that your loop variable does not need to begin with *i* = 1 or *i* = 0.

Trickier stuff: nested loops

Loops can be nested within one another. This is surprisingly useful.
Let's go back to the compounded interest example. Suppose we now want to compound the interest annually, but across a period of 5 years. The **for** loop we discussed earlier only compounds for a single year. Try this (note that this example is a bit different from the previous – I've shortened bankAccounts to make it more manageable, and I've eliminated the *compounded* vector.

```
bankAccounts <- c(10, 9.2, 5.6);
interestRate <- 0.0525;

for (j in 1:5)
{
	for (i in 1:length(bankAccounts))
	{
		bankAccounts[i] <- interestRate*bankAccounts[i]+ bankAccounts[i]
	}
	print(bankAccounts)
}
```

There are a few things going on here. First of all, there is no compounded vector. Rather, we are changing the value of the bankAccounts vector by performing an operation (interestRate*bankAccounts + bankAccounts) and then storing the NEW (compounded) value in bankAccounts.

At the end of each year, we print out the current bankAccounts vector. You can compare the numbers to the original to verify that each account grows by 5.25% each year (compounded just once).

And of course, the outer "j" loop corresponds to years. "j" really doesn't do much, except to tell the inner "i" loop to compound the interest again. When j = 5, the outer loop is done, and we have compounded 5 years worth of interest.

11) The three students in the above example have estimated annual expenditures for food, housing, and fun of: (in thousands of dollars)

```
house <- c(4.8, 3.8, 5.7);  food<- c(3.5, 4.3, 5.0);
fun <- c(7.8, 2.1, 10.5);
```
and incomes (through TAships) of
```
income <- c(21, 21, 21);
```

Modify the 5-year interest-compounding code above so that each year, it deducts the expenditures and adds the income (BEFORE compounding) for each student. You can assume (for example) that food[2] and food[3] correspond to the food expenditures of the 2nd & 3rd students, respectively. Etc.

You can even pass loop variables from an outer loop to an inner loop. Watch what happens in the following construct:

```
for (i in 1:5)
{
    for (k in i:5)
    {
        cat('i is ', i, ' k is ', k, '\n', sep='');
    }
}
```

You might expect this expression to print <u>25 lines of output</u>: 5 *i* loops multiplied by 5 *k* loops each. But this is not what happens. Look closely at the conditional part of the *k* loop: we specified **`for (k in i:5)`**. However, the value of *i* changes with each iteration of the i loop. When *i* = 1, the *k* loop runs from 1:5, but when i = 2, the k loop only runs from 2:5, and so on...when *i* = 5, the *k* loop runs only from 5:5, which is results in just a single run through the *k* loop.

12) Write a script that prints the pattern below. Save your script as file "invert_pyramid.R". When you source this file, it should print the pattern below to your R display. You need to use the function **cat()** to print output to screen and two **for** loops (nested). You can just include the code that does this in your main homework script file (you don't need to upload a separate file to Dropbox).

```
##########
#########
########
#######
######
#####
####
###
##
#
```

Important note on programming conventions: I will only permit you to use several loop variables (the stuff in the conditional part of the **for** loop). Acceptable variables include lowercase i, j, and k. Although R will allow you to use almost anything here, it is good programming practice to use the same variable names for certain operations. For example, some folks use *n* as a loop variable in **for** loops. However, this might cause you problems at some point, because *n* is a common variable for other things, like sample size. Thus, if you have a large project that uses some variable *n* to denote (say) sample size, you will overwrite the variable if you use it as a loop variable (thus causing you to lose whatever information was originally stored in it). R doesn't give you any warnings when it overwrites variables, so you have to be very careful that you don't inadvertently assign new values to important variables.

13) Challenge! Use nested **for** loops in a script 'pyramid_fun.R', which prints the following to the display when sourced:

```
ABCDEF
 ABCDE
  ABCD
   ABC
    AB
     A
```

Think about it! Tricky! You can do this with only 2 for loops (and maybe even 1). **This is optional (for now, anyway), but you'll demonstrate your mastery of loops if you can solve this!**

More data frame manipulation

14) The data file 'skinks.txt' consists of pitfall capture data for *Ctenotus* skinks from Lorna Glen station in central Western Australia. Columns of the data file include species names ('species'), the site (24 sites total), the individual trap (letter code), the mass (grams), snout-vent-length (mm), sex, and capture date of each lizard.

Create a dataframe 'lizards' by reading in 'skinks.txt'.

How many species of *Ctenotus* were captured at Lorna Glen? Answer this question exactly, using a combination of `length()` and `unique()`. What does `unique` do? I'm not telling.

15) Write several lines of code that (1) create a new data frame with all missing data (NAs) for sex and weight omitted. Hint: what does the function `is.na` do?

NOW: make yet another dataframe containing only individuals identified as “leonhardii”, using subsetting tricks that you already know. Be careful: does this dataframe have rownames?

FINALLY: Provide code that counts the number of male *leonhardii* individuals that were captured, unambiguously sexed, and weighed. Now repeat for females. Note that I have not necessarily denoted males and females by “male” and “female” – you can easily check the classifications used in $sex with the functions **unique()**. Ignore any questionable males or females.