BioEE 758
Week 7
Handout / Homework

What we know:
1) Data in/data out
2) Data storage: vectors/matrices/dataframes
3) Loops!
4) Conditional execution

Important stuff that we don't know:
1) Functions

Less important stuff that we will still learn:
1) Working with character strings
2) Vectorization and R tricks
3) Advanced data structures

Control-flow statements in computer programming specify how statements are executed. Are they repeated in a loop? Are they executed only if some particular condition is TRUE or FALSE?

Loops:
for (condition)
while (condition)

Conditional execution:
if (condition)
if else (condition)
else

and finally, one you haven't learned yet:

break

Keyword **break** instantly and unconditionally transports you outside of whatever loop you happen to be stuck in. If you are not in a loop, calling break will generate an error. This works only for **while** and **for**.

Consider:

```
cheeseburger <- 0.25;
weight <- 150;
while (cheeseburgers on plate)
{
     weight <- weight + cheeseburger;
     if (weight > 500)
          break;
}
if (weight > 500)
     print('You broke the scale!');
```

In this case, you start out with weight 150, and **while** cheeseburgers are available, you continue eating. Your weight increases by `cheeseburger` each time you run through the loop, until you break the scale (and **break** out of the while loop).

What is the final value of x in the following bit of code?

```
x<-0;
while (x < 100)
{
     x <- x+1;
     while (x < 50)
     {
          x <- x+1;
          if (x > 10)
               break;
     }
}
```

Compare your answer above to the following:

```
x<-0;
while (x < 100)
{
     x <- x+1;
     while (x < 50)
          x <- x+1;
     if (x > 10)
          break;
}
```

If they are different, why are they different?

1) Revisit the geometric growth problem. Model the growth of the population with an average of 2 progeny per individual per generation. Start with an initial population size of n = 2 individuals (p = 0.333) and use a MAXPOPSIZE of 10000. Start small, but work up to where this gives you the distribution of 5000 simulated bouts of population growth. Use the **break** keyword to rescue yourself from the problems that will arise when you try to implement this. When this particular problem arises, you should store the result for this simulation as NA. Provide your code.

When you set.seed as 3 before the start of the simulation, how many failed runs do you observe out of 5000?

The 1944 **Harvard Mark-1**, world's first true programmable computer. This hulking machine was capable of 3 calculations per second. It was 52 feet in length and weighed 10,000 pounds. Perhaps unsurprisingly, the chairman of IBM at this time famously stated that "*I think there is a world market for maybe five computers.*"

Functions

A function is a self-contained bit of code that performs a task. It might sum a set of numbers, run a simulation, or print your name backwards 500 times.

Functions – and you have already been using them extensively – are incredibly useful for at least three reasons. First, using functions makes your programming **modular**. This makes your code easy to read and simple to implement. It enables you to independently test components of your code without having to put the whole thing together.

Second, **functions make your code reusable**. If you have a function to sum a vector of numbers, or a function to calculate the number of minutes remaining until you can dance naked under the moonbeams on winter solstice, you can easily put those functions in any future coding projects you

might undertake.

Finally, functions enable you to **isolate your code from unintended side effects**. Functions take as arguments whatever variables you pass to them, but any variables used within a function will be invisible to all other functions. Thus, you can safely modify the values of variables within a function without changing the value of variables with the same name in other functions. More on this when we discuss ***scope*** next week.

A general function might work something like this:

1. Get user input (*e.g.*, *arguments*)
2. Perform some operations (the *black box*)
3. Return something.

Actually, items 1 and 3 are not necessary. You can have functions that take no arguments and functions that return nothing:

```
takeNoArguments <- function()
{
     cat('this function takes no arguments\n');
     cat('it also\n');
     cat('returns nothing\n');
     cat('you never get something for nothing.\n')
}
```

But of course, many useful functions take arguments:

```
takeNoPrisoners <- function(prisoners)
{
     if (prisoners > 0){
          cat('Scram!\n');
          cat('This function takes no prisoners\n');
     }else{
          cat('yeah!!! got em!!!')
     }
}
```

To execute or ***call*** a function, you simply do what we have been doing all along: type the name of the function followed by parentheses, with any arguments to the function going in the parentheses. For the above example, try you can execute prisoners by doing `takeNoPrisoners(3)` or `takeNoPrisoners(0).`

To define a function, you use the **function** keyword like this:

```
myFunction <- function(arg1, arg2)
```

This says that you want you create a function named ‘myFunction’ which takes two arguments, arg1 and arg2. Below this line, you enclose the statements belonging to the function in curly braces:

```
{
    cat(‘this is my function’);
    cat(‘dont mess with it’);
}
```

Once you have defined your function, it is part of your workspace. Until you remove it, you can use it.

Enter the following function:

```
greeter <- function(name)
{
    cat('Hello, ', name, '\n');
}
```

The greeter function takes a variable as an argument and performs the greeting, as in `greeter(name = ‘greedo’)`. What happens if you fail to supply argument `name`? And what happens if you just type the name of the function without any parentheses? In the latter case, it simply prints the function definition to your R console. This is why

```
sum()
```

and

```
sum
```

give different output on your screen. When you type `sum()`, you are executing the function `sum`, but you aren’t sending it any argument, so it sums nothing and returns the sum of nothing. However, when you just do `sum`, you probably get something like this:

```
function (..., na.rm = FALSE)
.Internal(sum(..., na.rm = na.rm))
<environment: namespace:base>
```

What does this garbage mean? The first line gives you the function and its *default arguments* (explained below). The ***.Internal*** means that the code for the function is inaccessible: generally, this means that the code is not written in R, but in some other language (in this case, C). Because C is a faster language than R, the R people have programmed most commonly used functions in C, and when you call the function in R, it is actually sending those arguments to a bit of code written in C. The line with **`:base`** means that the function `sum` is part of the R base package, which is downloaded by default when you install R on your system. If none of this makes sense, don't worry about it. Just know that many functions R provides are ***internal*** functions – the code for which *you cannot access and cannot modify*.

Return values

A function *returns* at most one object. The object can be a list, a vector, a dataframe, or a matrix, but it cannot return multiple objects. As you will learn in a few weeks, *lists* may contain a hodgepodge of all sorts of things, so this isn't as much of a limitation as it might seem.

Default arguments

Functions can take default arguments. You have already dealt with these when using functions like `read.table()` and `sample()`. Let's look at the greeter function again, this time with a default:

```
greeter <- function(name = 'Don Juan')
{
    cat('Hello, ', name, '\n');
}
```

This creates the function `greeter`, but provides a default argument for name if the user fails to specify something. In the `greeter` function, with no default, calling `greeter()` with no argument generates an error, but the second version doesn't even complain. It just uses whatever default argument you provided.

2. Give the default arguments for the following R functions. Where applicable, list the argument(s) required by the function such that it does not return an error message:

a) `rnorm`

b) `sum`

c) `t.test`

d) `sample`

e) `sort`

Scope

Variables and functions in R (and other languages) have *scope*. *Scope* refers to the *range of operation* of a variable or a function. If a variable is defined within a function, we say it has *function scope* and is only defined within that function. This means that if you define a variable within a function, that variable is entirely separate from anything else that exists within your workspace, even if you are already using that variable name elsewhere. For example:

```
x <- 'pizza';
cat('outside the function, x is\n',x,'\n\n');
func <-function()
{
    x<- 'lemonade';
    cat('but inside the function, x =', x);
    cat('\n\n');
}
func();
cat('after executing the function, x is ', x);
cat('\nThe value doesnt change\n');
cat('because of scope! I love scope!');
```

Put this in a script file and source it. You might think that the value of **x** should change after executing the function `func`, but it doesn't.

We will make a much bigger deal of this next week.

Validating input

Our final topic is the validation of input. If you have a function `greeter` that is supposed to accept a single name (e.g., a character string), how do you ensure that the user is in fact sending a character string as an argument, rather than a vector of names? Or a number?

Given the `greeter()` function on page 6, see what happens when you do `greeter(5)` and `greeter(pi)`. Clearly, an inability to separate numbers from people would be problematic for any prospective Wal-Mart greeter.

Let's revisit `greeter`, but now we'll use a very useful input validation function **stop()**. **stop** terminates function execution and returns an error message. You need to use an **if** statement to set the conditions under which stop is supposed to execute, and you can specify the appropriate error message as an argument. Try this with the version of `greeter` below using both a numeric argument and an argument consisting of a vector of multiple character strings (e.g., `c('Brenda', 'Walt', 'Pinnochio')`).

```
greeter<-function(name = 'Don Juan')
{
    if (!is.character(name))
        stop('this program doesnt do character flaws');
    if(length(name) > 1)
        stop('too many names for a poor little program');
    cat('hello, ', name);
}
```

One advantage to this sort of error-checking is that the error message indicates what was wrong with the input.

Problems

3) Write a function that takes as arguments the **length** and **width** and **height** of a box (in cm). The function should calculate the volume and print "the volume of the box is...[volume]". It should return the number of mail order kittens that would fit in the box if the mean kitten volume is 1500 cm^3. You can't send part of a kitten, so some rounding may be necessary.

4) Write a function that that takes an integer as an argument and, using a **for** loop, evaluates the factorial *n!*, where

*n! = n * (n-1) * (n-2) * ..... 3 * 2 * 1*

The function should return the answer. Of course.

5) Once again, back to the geometric growth problem. Write a function, `doGrowthSimulation()` that takes two arguments: the initial population size, and the population growth parameter ($p = 0.333$). The function should return the number of generations to reach MAXPOPSIZE, which should be defined outside of your function. Plan for contingencies: return ***NA*** if the population size goes to zero.

6) A random sequence simulator. Write a function that simulates random DNA sequences (4 letters, actg). As an argument, the function should accept the length of the sequence to be generated, and it should return an error message if this argument is inappropriate. This is similar to the random drift problem.

Once the sequence has been simulated, you'll have a vector of letters; you can bind these together into a single character string using `paste()` [see help on `paste` for the default argument you need to change to make this work]. In other words, your program should return an object of length 1 that might look like this:

`"acgggtcatccgatag"`

7) Write a function that solves for the roots of a quadratic equation

$ax^2 + bx + c = 0$

given coefficents *a*, *b*, *c*. I suspect that this is mostly forgotten algebra by now, but you can find the roots *x1* and *x2* of this equation as:

x1 = -b + sqrt(b^2 – 4ac) / 2a
x2 = -b - sqrt(b^2 – 4ac) / 2a

Your function should return an error message if coefficients a, b, or c are non-numeric or otherwise invalid input (hint: think `!is.numeric(x)`).

Note that there are four possible outcomes here: (1) the user enters invalid input; (2) the equation has two real roots (x1, x2 are different); (3) the equation has two repeated real roots (x1 and x2 are identical); and (4) the equation has complex roots. A complex root occurs when the root includes an imaginary number (like sqrt(-1)). Imaginary friends are OK, but we don't like imaginary numbers in ecology & evolution.

Use an initial if statement to check for valid input and exit with a `stop(...)` statement if output is invalid (try `stop("invalid input")`. Then use an `if...elseif...else` construct to check for outcomes 2:4.

Your function should return nothing if roots are complex (that is, if *b^2 – 4ac < 0*), but it should print something indicating that roots are complex.

If there is a single root (if *b^2 – 4ac == 0*), print something like "This equation has two identical real roots:....".

Your function should use `cat` to write output to screen, and should return a vector of length one or two depending on whether there are real identical or non-identical roots.

8) Revisit #12B from last week (Bouncing Brownie with a reflecting boundary). Write a function `move()` which takes four arguments: Brownie's current position along one axis (e.g, x or y), the standard deviation of her step size, and the (minimum and maximum) limits to her movement (e.g., 0 and 10 in our example). Allow for the possibility that there is no limit (e.g., minbound = NA). The function should return her new position, and it should not allow her to step over a boundary (the function should implement the reflecting boundary). One advantage to writing the function like this is that you can then use it model motion in any arbitrary number of dimensions.

If Brownie is "high as a kite", so to speak, we might need to model her movements in three dimensions. With the function `move()`, this becomes straightforward (and the code looks much better!). Likewise, if she is surfing the space-time continuum, we might wish for an improved four dimensional model.

9) Off the wall but strangely relevant: what is the **numerical value** of each of the following expressions? (yes, each has a real, numerical value).

a) `5 > 2`

b) `(3 + 4  > 2) & 3 < 2`

c) `5 != 2`

d) `(c(3, 2, 1) >= 2) + 1`