Class Exercise 2

We will explore some useful features of the `seq` and `rep` commands. I find that I often have to build a table of indices, either for an experimental design, or for a categorical data analysis problem. For example, suppose I have two factors **A** and **B**; **A** has 3 levels, **B** has 4 levels. I could enter the following commands to create a matrix of indices:

```
A=c(1,1,1,1,2,2,2,2,3,3,3,3)
B=c(1,2,3,4,1,2,3,4,1,2,3,4)
cbind(A,B)
```

Did we obtain all possible combinations of **A** and **B**? The above approach could be tedious for large data sets. The sequence for **B** is easier to create, so we'll do that first:

```
B=rep(1:4,3)
```

The above command repeats the sequence 1 through 4 three times. **A** requires us to repeat each index 4 times before moving on to the next index, so we need an entire vector `(4,4,4)` to indicate how often each element is repeated. As the second argument to `rep`, we could use either `c(4,4,4)` or the second `rep` command shown below. The first method is actually a little more transparent, though we use the second method below, since it is a little more compact.

```
A=rep(1:3,rep(4,3))
A
B
cbind(A,B)
```

The following command works even better when, as in the above example, each element of the sequence is repeated the same number of times:

```
A=rep(1:3,each=4)
A
```

Each of the 3 elements in the sequence is repeated 4 times before moving on to the next index. Suppose the indices of **A** were 10, 20 and 30, and the indices of **B** were 5, 10, 15, 20. How would you generate a table for **A** and **B**? Use the `seq` (rather than `c()`) command in constructing **A** and **B**.

An even simpler method to generate a set of indices is through the `expand.grid` function. Try the following code and compare the output to your previous output. Can you use `expand.grid` to generate similar output to the table you produced in the previous paragraph?

```
expand.grid(A=1:3,B=1:4)
```

Sometimes, we are in a situation where we need to set aside the space for a matrix or vector because we plan to update it; for instance, in a user-built function with an iterative loop. In this example, we define **a** as a null vector, and then add on to it:

```
a=c()
a=c(a,1)
a
```

This works too:

```
x=NULL
x=c(x,1)
x
```

The following command sets up a null matrix with known dimensions:

```
x=matrix(nrow=3,ncol=4)
x
x[2,3]=1
x
```

We have to be careful about specifying arguments; suppose we had typed:

```
x=matrix(3,4)
```

What does **x** look like? To set up a null matrix, we should have used:

```
x=matrix(NA,3,4)
```

This approach actually works with any constant value.

Setting up a null list requires a slightly different approach, we need to use the `vector` function and specify the vector type as **list**. run the following commands and comment on the list vector

```
PARMlist <- vector("list")
PARMlist[[1]]=1:3
PARMlist[[10]]=2:6
PARMlist
```

When we used **read.table** in class, we assumed we had tab-delimited data. I would like you to read the same data set as before in comma-delimited format. You can find `brainbod.csv` on my website; download it and be sure to change the working directory in R to the directory where you saved the file:

```
brainbod.df=read.table("brainbod.csv")
brainbod.df
```

Did it work? Try this command instead, in which we specify that commas are the text delimiters:

```
brainbod.df=read.table("brainbod.csv",sep=",",header=T)
brainbod.df
```

We can also use the following command, designed specifically for comma-delimited data:

```
brainbod.df=read.csv("brainbod.csv")
brainbod.df
```

`read.table` and `scan` have many additional features. I usually clean up the data in something more convenient, like Excel, save it in a tab-delimited file, and then read it in R, rather than fix all the data entry problems while reading a file into R.