EOS 864 - INTRODUCTION TO R

R is an "interpreted language" – statements will be executed as you type them at the R prompt. You will typically perform analyses with several statements, which are (1) either typed directly on the command line, or (2) entered into a text file that is executed all at once. We will start with the command-line approach, but you can save your typed commands, or your entire R session to use in the future.

The Basics. The most simple usage of R is as a calculator. Type the following statements at the R prompt ("`>`"), *e.g.,* for the first one, just type "3" and press "return".

```
> 3
[1] 3
> 3 + 5
[1] 8
> 3.4 + 6
[1] 9.4
> x = (3.4 + 6)/4.
> x
[1] 2.35
> y = sin(pi*x)
> y
[1] 0.8910065
```

For the statements that return a value, the values are numbered – in these examples, a single value is returned, and indicated by "`[1]`".

If you are new to programming, we refer to statements like `x=10` as an "assignment". The equals sign indicates that the value to its right is being assigned to a variable called `x`. Also, it is important to remember that the names `x` and `y` above are chosen by the user, and they could just as easily have been called `yyy` or `Fred` (try it). More about this later.

Spacing generally doesn't matter, and statements can span multiple lines. We'll talk more later about formatting your R code for readability.

```
> z = x+sin(pi/4)* (x -
+ 4.5 + log(4.5    ) /
+ 44.9)
> z
[1] 0.8534074
```

Also note that R has several pre-defined useful constants (*e.g.*, `pi` in the example above), including letters and the names of the months:

```
> pi
[1] 3.141593
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"  "p" "q" "r"
[19] "s" "t" "u" "v" "w" "x" "y" "z"
> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

There are also built-in data sets for testing, see more with data(), and ?(datasetname) will tell you more about the data.

```
> airquality
    Ozone Solar.R Wind Temp Month Day
1      41     190  7.4   67     5   1
2      36     118  8.0   72     5   2
3      12     149 12.6   74     5   3
4      18     313 11.5   62     5   4
5      NA      NA 14.3   56     5   5
6      28      NA 14.9   66     5   6
7      23     299  8.6   65     5   7
      ...etc...
```

Data Types. R provides a number of different types of objects for storing data. The most basic is called a vector. A vector is an ordered sequence of numbers, character strings, logicals (boolean values `TRUE` or `FALSE`), or factors (class variables). There are many ways to create new vectors, but here are some of the most common:

1. Use the concatenation function "`c()`"

```
> x = c(2,4,5,60)
> 2*x
[1]  4  8 10 120
```

Note that each value in the vector 'x' has been multiplied by 2.

2. Use the sequence function "`seq()`" specify a range of values. Notice that you can type

```
> y = seq(1,4)
> y/3
[1] 0.3333333 0.6666667 1.0000000 1.3333333
> seq(1,3)
[1] 1 2 3
> seq(1,3,by=0.1)
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7
[19] 2.8 2.9 3.0
```

In the last line you can see a very common way to modify the default action of a function, by specifying an argument that modifies the basic action of the function – in this case, the 'by=' specifies the step-size between the min/max values of the sequence. Also both these examples illustrate that arithmetic operations can be performed on vectors in the same way as with individual numbers (or scalars). In fact, a scalar is just a vector with only one element:

```
> q = 99
> q_vector = c(99)
> q - q_vector
[1] 0
> q == q_vector
[1] TRUE
```

Try this – put your new R math skills to the test by creating a vector of temperature (F) values values running from zero to 100 at half degree increments. Convert that vector to degrees C.

The test for equality "==", like other such tests ("<", ">", "<=", ">=", "!="), yield a logical (TRUE or FALSE) value.

```
> n = c(3,4,100)
> m = c(4,4,4)
> n < m
[1]  TRUE FALSE FALSE
> n <= m
[1]  TRUE  TRUE FALSE
```

Like the earlier arithmetic operations, these logical functions are performed pair-wise on vectors.

Vectors, like most R data objects, can be subsetted ("sliced"), using the [] notation.

```
> n[2]
[1] 4
> n[2]*m[3]
[1] 16
> n[1:2] + c(m[3], 5) + seq(1,2)   # help(":")
[1] 8 11
```

That last one combines a whole bunch of things that you have just learned. Does it make sense?

All R data objects have at least two intrinsic attributes: *length* and *mode*.

```
> length(n)
[1] 3
> mode(m)
[1] "numeric"
```

It's pretty obvious what length means, and the mode indicates the fundamental data type (*i.e.*, numeric, character, logical, or factor). This will become more important later when the objects and functions we use are more complicated.

Here's a quick look at character data types:

```
> pals = c('mike', 'steve', 'maria', 'maggie')
> length(pals)
[1] 4
> mode(pals)
[1] "character"
> pals == 'steve'
[1] FALSE TRUE FALSE FALSE
> pals.named.steve = (pals == 'steve')
> mode(pals.named.steve)
[1] "logical"
```

The length of the character string doesn't matter, regardless of the number of characters, it is still a single item.
Also note, that a variable name must START with a letter, although it can contain alphanumeric characters (`a-z`, `A-Z`, `0-9`), the underscore (_) and the period (`.`).

Vectors, like other data types, can have additional attributes. These are optional, and supplied by you, the programmer.

```
> aww = c(23, 44.45, 2002)
> names(aww) = c('count', 'value', 'year')
> aww
  count   value    year
  23.00   44.45 2002.00
> attributes(aww)
$names
[1] "count" "value" "year"
```

Name attributes especially come in handy with large tables of data ("data frames"), which we will get to later.

Basic Plotting. Here we will generate some data from within R, and try out some simple plotting functions. The `rnorm()` function creates a vector of normally distributed values, and the `runif()` function generates a vector of uniformly distributed values. Like most other functions they have default settings that can be changed, but let's not worry about that too much right now. The `plot()` function invokes the simplest, but most commonly used graphics tool in R.

```
> x = runif(100)
> y = rnorm(100)
> z = x + 0.2*y   # add a little noise to x as a function of y
> plot(x)
> hist(x)
> plot(y)
> hist(y)
> plot(x, z)
> plot(x, z, type='b')
> plot(x, z, type='p', xlab='Width', ylab='Height')
```

This is a good time to introduce the "help" utility. All R functions have a help page, *e.g.*,

```
> help(plot)
> ?plot
```

R graphics are highly customizable, but it takes a while to get the hang of the odd naming for the parameters. What follows is a more complicated plot example with several commonly-used statements and features. Let's run this example, then break down what each line is doing.

```
> x = seq(0, 2*pi, by=pi/100)                                    # 1
```

```
> y.true = sin(x)                                                    # 2
> y.obs = y.true + rnorm(length(y.true))                             # 3
> y.smoo = smooth.spline(x, y.obs)                                   # 4
> y.low = lowess(x, y.obs)                                           # 5
> plot(x, y.obs, pch=16, cex=0.6, col='darkgray', ylab='y')          # 6
> lines(x, y.true, col='pink', lwd=3)                                # 7
> lines(y.smoo$x, y.smoo$y, lwd=2, col='purple')                     # 8
> lines(y.low$x, y.low$y, lwd=2, col='green')                        # 9
> legend('bottomleft', c('True', 'Spline', 'Lowess'), lwd=c(3,2,2),  # 10
+     col=c('pink','purple','green'))
> index.med = which(y.obs == median(y.obs))                          # 11
> x.med=x[index.med]                                                 # 12
> y.med=y.obs[index.med]                                             # 13
> points(x.med, y.med, pch=4)                                        # 14
> text(x.med, y.med, paste("Median:",format(y.med, digits=4)), pos=4)  # 15
```

Note that the "`#`" sign is the comment character. Everything to its right is not interpreted by R.

1. Generate the independent variable vector `x`. What is the length of `x`?

2. The underlying function for this example is just part of a sin wave.

3. Add some gaussian noise to simulate a data set. Remember what `length()` does.

4-5. Perform a spline smoothing and lowess regression (Don't worry about what these do right now, we'll talk in more detail about these later – we are just creating some data to explore plotting options).

6. This high-level plotting command generates the basic elements of your graph. If you are going to plot a lot of things together it's a good idea to plot the data with the most variability first, to get the axes right, but there are other ways to do it. Input parameters are used to select symbol type (`pch`), symbol size (`cex`), line color (`col`), and to override the y-axis label (`ylab`).

7-9. Add lines to the plot using `lines()`.

10. Add a legend. This is a somewhat unfriendly function. It took me a while to get used to the fact that it does not care what you actually plotted. You can put anything in the legend.

11. Let's say, for some reason, we want to show where the median value is located. The where function will return the index (position) of the element in the vector that meets your specification.

12-13. This is the data point we are looking for. Simple array indexing using the [] notation.

14. Plot that point using the `points()` function, this time using an X (denoted by pch=4). Look at help(pch) for some other options on setting up symbol shape and color. And here's a nice graphic that shows point style options
http://rgraphics.limnology.wisc.edu/images/miscellaneous/pch.png

15. Add some text at that location. The `paste()` function simply combines strings together. Try the following at the R command prompt, then see how it is used when adding text to the graph:

```
> paste('the answer is', log(1.31))
```

The `format()` function allows you to trim the number of digits when printing out values. After trying the following, try printing your x vector values

```
> paste('the answer is', format(log(1.31), digits=3))
```

A final note about plotting parameters before we move on. Most parameters that involve a choice (*e.g.*, symbol type, line type, color), can be specified by name or number. The power of the latter can be seen here:

```
> plot(x, y.obs, pch=16, cex=0.6, col=rainbow(length(x))[rank(y.obs)])
```

Here's something fun – the plotting options that use a numeric value can be set with a vector of values – remember that cex indicates the size of the point – here we plot values in two vectors, and set the point size to the value in a third vector:

```
> noise = abs(y.true - y.obs)
> plot(x, y.obs, pch=16, cex=noise)
```

Or a slightly more complicated example involving factors:

```
> cut(noise, 3, c('L','M','H'))
[1] L L L M L L H M L L M L L L L M M L H M M L L L L L L M M L L M M L
[36] L L L L L L M L L M M L M M L L L L M L L M L L M L L M L M L M H M M
[71] L L L L H L L H L L L L L L M L L H M L M L L M L M H L M L L M L L L
[106] M M L L L M M L L M H L L L L L M L L L L L L L M L L M L H L L H M L
[141] L L L M L L L L M L M L L L H M L H M H M L M L M H M L L L L L H L L
[176] M M M M H L L M L M L L M L L M M M L L M M L M M M
Levels: L M H
> noise.level = cut(noise, 3, c('L','M','H'))
> plot(x, y.obs, pch=as.character(noise.level), cex=0.6)
```

What exactly does `cut()` do? Use the R help utility; type `help(cut), or ?cut`.

More on data structures. The other commonly used data types in R that we will encounter in this class are: lists, arrays, data frames, and time series. I will briefly introduce these here, but their uses and idiosyncrasies will be more apparent when you actually use them.

1. A list is just an ordered collection of objects of pretty much any type (mixing and matching types is ok in a list). Probably not the sort of thing you would see in your data.

```
> list(23, 'resp', c(1,4,2), 33.3, list('day',5), 1024)
```

2. An array is a vector (ordered sequence of numbers) with a dimension attribute:

```
> dat = c(1,3,7,9,11,35)
> dat
[1]  1  3  7  9 11 35
> dim(dat)                  #a simple vector with no dimensions
NULL
> dim(dat) = c(2,3)         # we can assign dimentions to the vector
> dat                       # now this vector has become a 2row, 3col array
     [,1] [,2] [,3]
[1,]    1    7   11
[2,]    3    9   35
> dat[2,]                   # specify row/col to specify array index
[1]  3  9 35
> dat[,2]
[1] 7 9
```

3. A data frame is an ordered collection of vectors, subject to the following rules: (1) all the vectors must have the same length; (2) all the elements within each vector must share the same data type (*e.g.*, numeric, factor, character, or logical). Conceptually, you can think of this data in rows and columns, as you would when using a spreadsheet. It has has a rectangular shape, with a certain number of rows and columns (like an array), but the columns can be different types (like a list). You will use data frames a lot. Let's make a simple one now:

```
> my.data = data.frame(age=seq(10), wt=rnorm(10,30),
+ grp=sample(c("A","B"),10,repl=T))  # see what sample does? If not, ask me.
> my.data
> my.data[4,]     # fourth row
> my.data[,2]     # second column
> names(my.data)  # what are the column names?
> my.data$wt# access the column called "wt" using the dollar sign
> plot(my.data$age, my.data$wt)
> plot(wt ~ age, data=my.data, pch=as.character(grp)) # another way to access
data frame columns?
> age
> attach(my.data)
> age             # what just happened? Type "help('attach')".
> plot(grp, wt)   # we'll see how to plot factors next time!
> detach(my.data) # try to remember to do this
```

As you can see, data frames provide an extremely useful and flexible data structure. Also, you might have noticed that different types of objects respond to functions (*e.g.*, `plot()`) differently. Let's try a different data frame, remember that R has several of them pre-loaded:

```
> trees
> help(trees)
> names(trees)
> summary(trees)  # often very useful
> plot(trees)     # special plotting method for data frames
```

Enough for today? Questions/comments?