Handout #1
A "Crash Course" in R

Lesson 0: Motivation.

This package may not seem too appealing at first, because it's not particularly user-friendly. You might think to yourself, "Why can't we just work with Excel or some other program that I already know how to use?" Here are some motivational thoughts, albeit biased, taken from the R website:

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- *an effective data handling and storage facility,*
- *a suite of operators for calculations on arrays, in particular matrices,*
- *a large, coherent, integrated collection of intermediate tools for data analysis,*
- *graphical facilities for data analysis and display either on-screen or on hardcopy, and*
- *a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.*

The term "environment" is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

That last bit is probably the most important advantage. Statisticians don't usually like to be told what to do when they're conducting an analysis, and a lot of other packages (like Excel) are really inflexible. With R, if you don't like the way a built-in function works, you can write your own! Don't worry – you won't need to do this. Statistical algorithms in R are very good, and they have plenty of options. Most of them are direct adaptations of functions written for S-PLUS, and others have been contributed by smart people who tested them out thoroughly.

R also has the advantage that it can grow quickly. As they develop new methods, statisticians write "packages" that are easily downloaded from the website and added to the environment. And, in case you suspect that we're just using R because it's free (which is nice, of course), remember that it's based on the (not free) S environment, the preferred analytical system of many statisticians for over a decade.

Lesson 1: Working with numbers and generating data.

Arithmetic

R can do any basic mathematical computations. Some of the most commonly used functions:

+, -, *, /: Addition, Subtraction, Multiplication, Division
%%: Modulus
exp(x), log(x): Exponent, natural logarithm
sqrt(x), x^(p): Square root, pth power (p can be any number, so x^(0.5) =sqrt(x))
round(x, digits): Round a number to "digits" decimal places.

You'll note that the last few of these are "commands," in that they take a number as input. Commands in R always enclose their input in parentheses, and often have several options (for example, log(x, base) lets you compute logarithms using any base you specify, although the default is base *e*). To get help with any command, type ?[command.name] or use the "R functions" option on the help menu.

Try a couple of simple computations, and read the help info for a command.

Objects

Suppose you want to save the result of a computation or other command. You can create an "object" as follows:

Option 1: x<-3+5
Option 2: x=3+5

Both of these commands assign the value "8" to object x. Most people prefer Option 1, because it feels more like "assigning" something, but Option 2 requires one less keystroke per command. *Note: people who may work with S-PLUS should probably use Option 1, since the "=" assignment operator isn't permitted in the S language.*

Important facts about object names:
You can assign any name to an object, with a few exceptions:

1. Avoid names that are identical to built-in R functions (for example, don't call an object "sum" or "mean"). If you're not sure whether or not an object name is reserved, you can just try typing it into the command line to see if anything other than "Error: Object [your.object.name] not found" comes up. If so, pick another name!
2. Object names can contain periods, like "my.object", but can't contain underscores ("my_object" is off limits). Spaces are not permitted.

3. Avoid upper-case letters, unless you have a specific reason for using them. R is case-sensitive, so "MY.OBJECT" and "my.object" are not the same. You might want to use upper-case letters to separate words, like "MyObject," but periods can work well here, too.

Exercise 1: Create an object that is the product of e^3 and $\sqrt[4]{30}$. Create a new object that rounds your previous object to the 2nd decimal place. Subtract the first object from the second object. What do you notice about the displayed answer?

Locating and deleting objects:
The commands "objects( )" and "ls( )" will provide a list of every object that you've created in a session (or loaded in from a previous session). This is handy if you've forgotten what you called something, which happens from time to time. There are also options for these command that let you search for all object names that match a certain pattern (a character, number, word, etc.).

The "rm" and "remove" commands let you delete objects, which is useful if you're worried about filling up space or just want to clean up.

Vectors
Many commands in R generate a vector of output, rather than a single number.

The "c( )" command: Creates a vector containing a list of specific elements.

Try these examples:
c(1,4,5)
c(1:5)
c(1:5,5:1)
c(rep(1:3,3))

The "seq( )" command: Creates a sequence of numbers. This command has a lot of options: type ?seq to read about it.

Try these examples:
seq(5)
seq(1,10,by=2)
seq(1,10,length=2)

Operations on vectors

To select an element of a vector, we use []:

Example:
> x=seq(2,5)
> x[2]
[1] 3
To select multiple elements, use [] with a vector enclosed:

Example
```
> x=c(11:20)
> x
 [1] 11 12 13 14 15 16 17 18 19 20
>y<- x[c(3,5,8:10)]
>y
[1] 13 15 18 19 20
```
Alternatively, we can concatenate commands to do this example in a single step:
```
y<-c(11:20)[c(3,5,8:10)]
>y
[1] 13 15 18 19 20
```

Some commands that operate on vectors:

sum(x): Computes the sum of the elements of a vector.
mean(x): Takes the average of the elements of a vector.
sample(x, size, replace = FALSE, prob = NULL): Takes a random sample from a vector, with or with replacement.
sort(x): Sort the elements of a vector.
min(x), max(x): Returns the minimum or maximum values of a vector.
length(x): Returns the length of a vector.
summary(x): Returns the min, Q1, median, mean, Q3, and max values of a vector.

Exercise 2: Use R to find the sum of the odd integers between 113 and 153.

Exercise 3: Write a command to generate a random permutation of the numbers between 1 and 20 and save it to an object. *Hint: This is equivalent to sampling, without replacement, from the sequence of integers between 1 and 20.*

Logical operators. Suppose we want to select only elements of a vector that meet a given criterion. We can do this using logical operators:

>, <, >=, <= : Greater than, less than, greater than or equal to, less than or equal to
==, !=: Equals, does not equal (logical)
&, ||: And, or

Example:
```
> x<-c(1,5,7,8)
> x>5
[1] FALSE FALSE  TRUE  TRUE
> x!=8
[1]  TRUE  TRUE  TRUE FALSE
```

Exercise 4: Write a command that tests for whether elements of the vector "x" are both greater than 5 ***and*** not equal to 8.

Once we can do the tests, we'd like to be able to select subsets of vectors that meet our criteria. To do this, we put the test inside the vector, so that we extract only the elements that we want:

```
> y=x[x>5]
> y
[1] 7 8
```

Another command that can come in handy here is "which," as in, "Which elements of my vector are greater than 5?"

```
> which(x>5)
[1] 3 4
```

Note that which returns the *position numbers*, not the values themselves.

Exercise 5: For the random permutation vector that you generated in Exercise 3, write a command to select only the even integers in the vector.

Writing simple functions

We might find that we're performing a series of steps a number of times and want to write a function to automate the process. This is very easy to do in R, since a function is just a string of commands. Suppose, for example, that we want to write a function that will compute the sum of squared differences from the mean of a numeric vector. We'll call our function "ssq( )", and our input is a vector of any length. Here's one way to do it:

```
ssq<-function(x)
{
sum((x-mean(x))^2)
}
```

That's it! Try it out…

Exercise 6: Write a function called that computes the Inter-Quartile range for a numeric vector (that's Q3-Q1, or the difference between the 1st and 3rd quartiles). *Hint: Use the summary( ) function!*

Multidimensional Data: Matrices and Arrays

The command to create a matrix in R is matrix(data, nrow, ncol), where "nrow" specifies the number of rows and "ncol" specifies the number of columns. For example, to create a 4x4 matrix filled with 0's, you would type matrix(0,4,4).

Type the following commands and compare the output:
matrix(rep(1:5,each=5),5,5)
matrix(rep(1:5,each=5),5,5,byrow=T)
t(matrix(rep(1:5,each=5),5,5))

To select elements of a matrix, we use [,]. For example, A[2,3] returns the element in the 2nd row and 3rd column of matrix A.

To select an entire row, leave the column index blank, and vice versa to select an entire column. For example, A[2:4,] returns rows 2-4 of matrix A with all columns included.

Common matrix operations:
cbind(x,y): Bind two vectors or matrices with equal numbers of rows into a larger matrix.
rbind(x,y): Bind two vectors of matrices with equal numbers of columns into a larger matrix.
dim(x): Returns a two-element vector containing the number of rows and columns in a matrix.
nrow(x), ncol(x): Returns the number of rows or columns in a matrix (respectively).
apply(x, margin, function): Applies a function to the rows (margin =1) or columns (margin=2) of an array. For example, apply(x,1,sum) computes row sums.

Exercise 7: Create the following matrix using as few steps as possible and save it to an object.

```
1     1     1     1     2     2     2     2
1     1     1     1     2     2     2     2
1     1     1     1     2     2     2     2
1     1     1     1     2     2     2     2
3     3     3     3     4     4     4     4
3     3     3     3     4     4     4     4
3     3     3     3     4     4     4     4
3     3     3     3     4     4     4     4
```

Lists. A list can have any number of dimensions, although typically three are sufficient for practical purposes. For example, suppose we want to divide the matrix in Exercise 7 (call this "x") into its four quadrants and store them in a single object. This can be done easily using a command like this:

```
>y=list(x[1:4,1:4],x[1:4,5:8], x[5:8,1:4], x[5:8,5:8])
```

What does the output look like? How would you select the last two rows of the third matrix in the array?

Exercise 8: The grand finale! Generate random samples of size 100 from the Bernoulli(*p*) distribution for each of the following values of *p*: 0.1, 0.2, 0.3, 0.4, 0.5. Create a matrix or array to store your output, and compute the means and variances for each sample. How close are your results to what you would expect to see in each case?

Coming up next… Real data!