**Welcome to Statistical Computing!**

In today’s world, data is everywhere. From grades in class to stats about your favorite sports team to poverty rates to public health information, you’ll find yourself bombarded by reams of data. With all these numbers, it becomes a challenge to make proper analyses without the proper tools. Pencil and paper or calculators simply won’t cut it anymore. We need to take advantage of recent advances in computing and statistical programming. And with that, welcome to your first introduction to the world of statistical computing!

There are many different statistical packages that exist today, including STATA (used in economics), MATLAB (used in engineering and some biological fields), SPSS (used in psychology and some biological fields), Minitab (used in Six Sigma in industrial sectors), SAS (used in business and academia), and JMP (developed by the makers of SAS for use in academic settings). Oh, there’s also Excel, but we don’t talk about that travesty. With all of the different languages available, it becomes difficult to choose a single language to prepare students for what they might encounter in the future. However, there is one language that offers many benefits for the typical student interested in doing data analysis. This is the language of R.

The genesis of R began in 1976 when its creators were working on the statistical language of S. The first alpha version of R appeared in 1993, with the first stable release occurring in 2000. Updates have continued over the years, with the most recent major release, R 3.0 in 2013 allowing for R to support numbers up to  (Trivia note: the name “R” is a play on the fact that the creators first worked on S as well as a fact that the two primary creators’ names began with the letter R.)

**Advantages of R**

R has several advantages that make it a worthwhile choice as a statistical package.

1. It’s free! You can download it to any personal computer and it has support for Windows, Mac, and Unix platforms. Simply go to the website [**http://www.r-project.org**](http://www.r-project.org)and follow the directions to download the program.

2. It’s used by a wide variety of individuals and companies. Although R was first developed primarily for use by members of academia, its use has grown over the years. Employees of Google routinely use R. Stock traders actually use it to process transactions.

3. Employers are routinely looking for individuals with expertise in R. A recent survey of job advertisements that sought specialty in a form of analytical software showed that R was the fifth most sought after specialty (ranking behind only Java, SAS, Python, and C). Analysis of trends also showed that R was gaining on SAS after recently passing SPSS.

4. A greater percentage of academic papers are using R. Although SPSS remains the leader in number of publications, that lead has dramatically shrunk in recent years.

5. It’s free! You can install R on as many computers as you wish without paying additional fees.

6. R is relatively easy to learn. Most commands are pretty straightforward and commands are fairly intuitive.

7. People are adding to R all the time. Individuals are always writing new code to address particular analyses or situations. These can be downloaded from the CRAN download page on the R website or directly through the R program.

8. Did I mention it’s free?

**Disadvantages of R**

Although R has several advantages, it is worthwhile to point out a couple of its shortcomings.

1. R is fairly memory intensive. If your dataset is massive, R may not be able to handle it. In this case, a language such as SAS may be more appropriate. Fortunately, the datasets we will be working with will be nowhere near the size limitations.

2. R can be slow at times. If you’re running simulation studies, this can become an issue (also related to the way R handles memory). If it is an issue, some options are to write the code in C and then have R handle the results. For our purposes, however, most simulations will be done in less than a second. At the worst, go make yourself a sandwich after starting the code. The program will be done by the time you get back.

3. R is a command-based language. There are very few drop down menus in the base program, which means you’ll have to know the command to run your analysis and type it in on your own. Although this can be daunting at first, you’ll find it relatively easy to adapt to. If you really need to have drop down menus, there are some programs that can be installed, such as R Commander.

4. The data stored in R does not appear in spreadsheet form. If you’re accustomed to Excel, this can be a difficult adjustment, since the data is typically stored behind the scenes. If you really need to see your data in spreadsheet form at all times, there are some programs that can be installed, such as R Studio.

5. While R use is growing in many companies, other companies prohibit its use. This is due to the fact that R is freeware with code that is very easy to create and edit. (Two of R’s advantages!) However, this can become an issue when companies are performing audits. For this reason, some companies insist on using languages such as SAS.

**Introduction to R**

Now that you’ve been introduced to what R is, let’s handle the basics of working in R. The very first thing I want you to do, and this should be the first thing you do every time you are going to do work in R that you may want to access later (e.g. for every single lab, or for any project work), is create an R script and save all of your commands there. To create a new script on a PC, go to File<New Document. On a Mac, you can simply use the keystroke <command>-N. In a script, you write commands just as you would in the R console. You can add comments (things you want to write to annotate your commands, or just old commands that you want R to ignore) by preceding something with a #. You will be required to submit your R script and your solutions to required problems for every lab. Please see <http://mazamascience.com/WorkingWithData/?p=958> for a nice example (the commands are excessively advanced, but they give you a nice feel for what a script can/should look like – except your output won’t be embedded).

**R as a calculator**

There’s no point in using a calculator or a second program on the side to do simple arithmetic problems when you already have R open.

The standard commands for the basic arithmetic operations are:

addition +

subtraction -

multiplication \*

division /

**Warning**: R needs to see the \* symbol if you want it to do multiplication. If you simply type:

(10 – 4)(4 + 6)

R will give you an error message. This needs to be written as:

(10 – 4)\*(4 + 6)

To raise a number to a power, use the **^** symbol. Thus,  can be evaluated by typing 4^3.

To find the square root of a number, the command is **sqrt**( ). Thus,  can be evaluated by typing sqrt(16).

When using Euler’s number, e, the command is **exp**( ). Thus,  can be evaluated by typing exp(2).

To find the natural log of a number, the command is **log**( ). Thus, ln 85 can be evaluated by typing log(85). *Note: In a typical statistics setting, natural log is written as* log*.* ln *is not used as frequently. This convention carries over to R. If you want to evaluate using a logarithm base 10, the command in R is* log10( ).

The standard rule of PEMDAS for order of operations holds in R.

**Your turn (not to be handed in):**

Evaluate the following:

1.  2. 

**Entering data into R**

There are two basic ways to get your data into R. You can either enter it by hand or you can import it from an existing file. Be aware that R uses terms such as rows, columns, and matrices to refer to strings of numbers or tables. Mastering these terms can help you access different parts of your dataset.

**Entering data by hand**

Let’s suppose that you have a number that you want to store in R. (Let’s say, the number 10.)

If you simply type the number into R:

> 10

it will spit back

[1] 10

and nothing will be saved.

To properly store your number, you need to give that number a name. (Let me call my number “x.”) To do this, simply type:

> x = 10

R will not spit anything back at you, but it now knows that x is equal to 10.

If you now type the letter x back into R:

> x

R will spit back

[1] 10

*Note: The*  [1] *at the beginning of the response from R is simply the position of that number in the stored variable. If there are a lot of numbers stored in your variable, they may wrap around when being displayed. If this occurs, the second row will have another number corresponding to the position of the first number in the second row. This can be beneficial if you’re trying to find a particular value or find the position of a particular number.*

Suppose we want to store a string of numbers in our variable x instead of a single number. To do this, we’re going to tell R that we want to **concatenate** a bunch of data. The command for this is simply **c**( ). Inside of the parentheses, we type all of our numbers, separated by commas.

For example, to store the numbers 4, 6, 8, 2, 3, 5 in the variable “numbers” we type:

> numbers = c(4, 6, 8, 2, 3, 5)

**Warning:** The names that you give your variables are case sensitive. Thus, if you want to, you can save one set of numbers as “numbers” and a second set as “Numbers” and R will view them as different sets. Variable names cannot begin with numbers. (“3CPO” is not a valid variable name, but “C3PO” is.) Many punctuation marks are also not valid to be included in variable names.

There are also a couple of shortcuts to entering strings of numbers.

If you simply want a string of the same number repeated many times, you can use the **rep**( ) command (for **rep**eat). All you need to do is tell it what number you want repeated and how many times.

If you type:

> y = rep(3, 8)

this will store in “y” the number 3 repeated 8 times.

> y

[1] 3 3 3 3 3 3 3 3

If you want to create a **seq**uence of numbers increasing by a particular interval, use the **seq**( ) command. You need to tell R what the first number is, what the last number is, and what the interval between numbers is.

If you type:

> z = seq(1, 3, by=0.25)

this will store in “z” a sequence of numbers from 1 to 3, increasing by 0.25.

> z

[1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00

There’s an even faster way to get a sequence of integers than using the seq( ) command. Suppose I want to store the integers from 4 to 15. To do this, I can simply use the colon **:**

**>** integers = 4:15

> integers

[1] 4 5 6 7 8 9 10 11 12 13 14 15

Note: Using the colon to create a sequence of integers will be very useful in the future when we’re doing simulation studies.

**Your turn (to be handed in):**

1. Find two ways to store the numbers 10, 20, 30, …, 150 as variables.

2. Store the numbers 3, 4, 5, …, 80, 80, 80, 80, 80, 80, 80, 80, 80, 80 as a single variable. *Hint: You can use other commands inside of the* c( ) *command.*

**Importing data from other files**

In many cases, we don’t want to enter data by hand. It already exists in a separate file, such as a .txt document or an Excel spreadsheet. We simply want to perform analyses on the data. How can we import the data into R?

The first step is find out what format your data is stored in and what is actually in the data set. Let’s consider the dataset “grades.txt” found on Moodle. Download the file and remember where you saved the dataset (this will be important later). If we open the file, we can see that it’s a table of data and the first row of this table gives the names of the columns. Also, we see that the columns are separated by white space. (Closer inspection reveals that there are tabs between the columns.)

How do we now import the data? First, we want to make sure that our active directory is the directory where the file is saved. (Unless we otherwise specify, R will only look for files in the current directory.) To do this, go to the drop down menu “File” on the upper left corner of the R window. Select “Change dir…” When the pop-up menu appears, select the directory where you saved the file.

Now that R is looking in the right place, we can tell it to read the data. Since the file is a .txt file, the command **read.table**( ) will be useful here. We want to tell R what the file name is and the fact that the first row is simply the column names. I’m going to import the data and save it as “grades.” To do this, I simply type:

> grades=read.table("grades.txt", header = T)

If you now type in

> grades

You can see the whole dataset has been imported and saved as the variable “grades”.

Observe that the file name is in quotes, including the .txt part. The second piece, “header = T” tells R that there is a header row where the column names are given. If the file did not contain such a row, we would type “header = F”.

***There is an alternative to first directing R to the directory where the data file is saved. You can also use the command***

grades=read.table(file.choose(), header = T)

***This will open a pop-up window where you physically navigate to the file in a finder window and select the data file of interest. Most people find this easier.***

Note: The default setting for the read.table( ) command is that white space separates different data points. This may not be true for your dataset. Sometimes a tab (\t) or a comma may be used to separate data points. For example, if we were importing data on states and encountered “New Jersey,” the space between “New” and “Jersey” would tell R that the two words were different pieces of data. Thus, we would need to specify how breaks between data points are indicated. This can be done by adding the **sep=** statement. For example:

> grades=read.table("grades.txt", sep = “\t”, header = T)

would indicate that a tab was used to separate the data points.

read.table( ) works well for .txt files. However, your data may come from an Excel spreadsheet. How do we import these?

First, although there are some packages that you can install to handle these cases, the base version of R cannot import .xls or .xlsx files. One of the primary reasons for this is that Excel spreadsheets can have multiple pages. To solve this problem, take the particular spreadsheet that you want to import and save it as a .csv file. Then, the procedure works as before, using the command **read.csv**( ). As an example, I’ve provided the “grades.csv” file, which is the same as the previous dataset except saved as a .csv file. To import this file and save it as “grades2”, type:

> grades2 = read.csv(“grades.csv”, header = T)

Since a .csv file already separates the data points using commas, you don’t have to worry about the separation type. (It’s in the name: **C**omma **S**eparated **V**alues.)

**Your turn (to be handed in):**

3. Upload the dataset “lifeexpect.csv” which gives the life expectancy for each state, the nation as a whole, and the District of Columbia. Give the code you used.

4. Upload the dataset “mccoy.txt” which gives the rushing statistics of former Philadelphia Eagles running back LeSean McCoy. Give the code you used.

5. **Challenge!** Upload the dataset “yahoostock.txt” which gives trading information about Yahoo stock over several months. (Note: If you blindly attempt to import the data using our first example for the grades, you’ll get an error message. This error message may give you a hint about how to solve the challenge.) Give the code you used.

**Working with the Data**

Now that we’ve imported the data into R, we can start to mess around with it. Let’s look at the test scores that we’ve saved as “grades”. Right now, the data is stored as a matrix, with each column having its own name.

Suppose we want to only look at the first test. There are a few ways to access this column alone.

1. Since the column is called “Test1”, we can extract this column alone by typing:

> grades$Test1

2. Since the column is the second column in the “grades” table, we can extract it by typing:

> grades[ ,2]

This second technique looks a little confusing at first, but it is especially helpful if you don’t have column names. How does it work? It uses row and column notation. Here’s another example that gives a good illustration. Suppose I want to find Bob’s score on the second test. Note that Bob is in the second row of data (the column headers don’t count) and the second test is the third column of data. To get the score, I would type in

> grades[2,3]

Thus, the first number gives the row and the second gives the column.

But what about my example above, where the number for the row was blank? This simply tells R that I want *all* the rows. Similarly, if I wanted all of Bob’s information, I would type:

> grades[2, ]

which would give me everything in the second row.

There is a third way, although I’ll put a big **Warning** with it. If I type

> attach(grades)

this will create new variables whose names correspond to each of the column names. If I then type: > Test1

it will give me all of the first test scores.

This looks so much easier, so why does it get a **Warning**? The **attach**( ) command will use the column names and overwrite any existing data that has the same name. So, go ahead and use the attach command. Just be sure you’re not accidentally overwriting something you want to save.

An alternative to the attach() command that I like a lot is the **with()** command, which acts as a temporary version of attach(), and doesn’t overwrite anything permanently. For example, if I wanted to get the average grade on Test1, I could simply write:

with(grades, mean(Test1))

Now that we can work with just one piece of the data, let’s finish our introduction with some basic mathematical operations on the data. Suppose I want to see how much better the students did on their first test compared to their second test. To find this difference, we can do basic subtraction, but applied to the vector of the test scores.

> with(grades, Test1 – Test2)

[1] -6 14 38 9 16 4 5 20 -3 3 15 15 56 33 27 17 21 5 25 21 -2 24 13 22 -8

Suppose I want to curve the second test by adding 10 points to everyone’s score:

> with(grades, Test2 + 10)

[1] 108 92 72 86 93 96 99 80 102 101 98 72 47 60 78 93 81 104 87

[20] 91 106 81 92 87 104

Suppose I want to find the average test score on the third test. The command for this is **mean**( ).

> with(grades, mean(Test3))

[1] 77.68

Some other commands that you may find useful (and fairly intuitive) are **median**( ) and **sum**( ).

**Your turn (to be handed in):**

6. Find the average difference between the high and low prices of the Yahoo stock.

7. Find the total number of rushing yards for LeSean McCoy over his career.

8. If you give a weight of 15% for the first test, 20% for the second test, 25% for the third test, and 40% for the final test, find the final averages for our 25 students. (As a bonus, give them a letter grade as well.)