#**GETTING STARTED WITH R:**
# The website address for the R Project for Statistical Computing can be found at:
# http://www.r-project.org/
# Under "Getting Started" go to **Download R**
# Choose a nearby CRAN site
# Under "Download & Install R" choose your platform and follow the instructions
# For Windows, you want the **base** package.

# **FINDING USEFUL INFORMATION:**
**#** The R main page above contains much information including the online manual:
# **An Introduction to R** that may prove helpful.  Also, check **FAQs**.

# The **CRAN** sites also have extensive documentation including **Task Views** for subject
# areas, and documentation associated with **Packages**.
# In **Contributed**,  are extensive contributed documentation and tutorials of all kinds.
# Personally, I use Google a lot to find documentation to specific things.

# **RUNNING THE R INTERPRETER:**
# When the R Interpreter is activated by the host computer, up pops a window called "R Console".  The console, in terminology derived from ancient days when a computer could only be addressed by its attached keyboard, is the primary means of interacting with R by means of a script.
#To see a script in action, cut and paste the following red lines into the console on the final line where you see the prompt sign '>', and press 'Enter' or 'Return'.
#
**# Note that every line prefaced by # is written but otherwise ignored by the R interpreter.**
**#**
**#SAMPLE SCRIPT**
**iris**
**class(iris)**
**summary(iris)**
#
#As you can see, each line of the script that doesn't start with # consists of a command that produces a response from R.  Scripts such as this are exceedingly useful for producing prototypes or statistical results automatically.
#In fact, this whole document is one large script which you can also run if you like!
#When working with R, I typically have a small text editor open and work on the script line by line cutting and pasting as I go.
#From time to time, and at a end of the session, I save the script as a text file so next time I work with a statistical procedure I remember what I've done, and repeat all properly prototyped steps exactly.
#
#In what follows, we will play with commands one line at a time to see what they do.

**# USEFUL ACTIVITIES FOR SUMMARIZING DATA IN R:**

# Examples datasets come already installed with R that may be consulted right away.
# For instance, the famous iris dataset of Anderson.
# Type or cut and paste the following line into the R interpreter and see what happens be sure to also hit 'Return':

```
iris
```

# Note the structure of this data with rows (each iris flower often called statistical "objects" "replicates"or "individuals") and columns (variables). One variable includes the species name for each individual.

# This kind of data structure is typical in statistics. In R, the data structure is given a special name.
#Try:

```
class(iris)
```

# the class "data.frame" is R's way of specifying flexible kinds of data including both numbers and character information (as in the species name in the final column) along with labels for rows and columns.
# Most commands in the R statistical system consist of built-in functions with the typical structure f(x) including the function name f() and called object (or domain) x. Pushing Return on the keyboard causes the function to run, usually producing the function's result (or range) often including information printed to the console.

# For summary information on the iris dataset, try the function:

```
summary(iris)
```

#Now, each variable (column) of the iris dataset is summarized with minimum and maximum values, means and medians, quartiles – all good statistical information. Note also that the Species column contains counts for each of the three species names in the iris dataset.

# For pairwise plots of all variables, try:

```
plot(iris)
```

#Now, you get pairwise plots of all columns. Because unlike MathCad, R is specifically set up for doing statistical analysis, its functions are typically much more powerful.

#Note that some plots don't make much sense! Why? In all statistical analysis, your job will be to interpret reports such as this and decide which are meaningful and which are not.

# It is often useful to be able to extract particular pieces of data from larger data tables. In R, you can extract the columns using the symbol $. Type:

```
iris$sepal.length
```

# What you get is an error message "**NULL**" meaning that R reports nothing! It is important to compare this line with the column variable label "Sepal.Length" reported above. Note the difference? Now try:

```
iris$Sepal.Length
```

# The R statistical language requires that you be case specific for all names. It turns out we were lucky in the first place that "iris" was all in lower case letters. However, "Sepal.Length" has both upper and lower case letters, and we must type things correctly. "Sepal.Length" is different from "Sepal.length" and so on. Irritating, perhaps, but not a big problem now that you have been warned!

# To avoid much typing, it is possible to simplify names for different data columns by "attaching" a datafile to the console's current 'environment'. Type:

```
Sepal.Length
```

#The interpreter returns the complaint: "**Error: object "Sepal.Length" not found**". But if you:

```
attach(iris)
```

#and then:

```
Sepal.Length
```

# you can now view each data column by name directly. In R, the opposite of attach() is detach(). Try this function and see what happens.

**# PLOTTING WITH R:**
# Now, let's do something useful with a single column of the iris dataset. After attaching the iris datafile to the environment, try:

```
hist(Petal.Length)
```

# A histogram like this is useful for investigating the distribution of the individual measurements of this variable (called "values" in this "sample" of measurements) in order to make a guess at the distribution of all possible values (called the "population" of measurements). This distinction is very important in statistics.

#To make the histogram more useful, you can specify the number of bins using “nclass” and colors to the bars as follows:
#

```
hist(Petal.Length,nclass=25,col="gray",border="red")
```

#
#Now, let's make a scatter plot of two variables. We will place Sepal.Length on the x axis and Sepal.Width on the y axis:
#

```
plot(Sepal.Length,Sepal.Width,col="red",pch=21,bg="green")
```

#
**# VECTORS, MATRICES AND USING AN INDEX:**
# Note that in R, as with many statistical programs, a single column of data such as Sepal.Length is called a "vector". A vector is simply an ordered list of numbers (sometimes other things) with the order indicated by an "index" indicating placement within the list.

# To access individual items within a vector in R, we use []. For instance, try:

```
Sepal.Length[7]
```

#What does this number mean? Compare this with the entire data frame, and find the 7th item in vector Sepal.Length.
# An entire list of data numbers, consisting of vectors side by side is called a data "matrix". The data frame "iris" consists of a data matrix of four variable vectors plus a vector of species names.
# To access any piece of information in a matrix, [] may be used as well. Here, however, you must specify both row and column indices:

```
iris[3,4]
```

# Can you find this number in the data matrix?
# Now try:

```
iris[3,5]
```

# Here the R interpreter tells you that the word "setosa" sits in this spot and is one of three possible alternatives (called "levels") including "setosa", "versicolor" and "virginica". This is the way R describes nominal variables (variables with discrete alternatives that need not be ordered) that it terms “factors”.

# For a little more fun, vectors consisting of multiple values from the data frame can be extracted by using an index vector we make on the fly, using the c() concatenation function:

```
iris[c(3,4,5,6,7),2]
```

# Compare with the entire iris data set to see how this works. As an extension, here's a powerful (and cool) way to make a vector by specifying start and end points in a series of indices using ":".  Try this and see what it does:

```
iris[c(1:6),3]
```

**#ASSIGNMENT AND EVALUATION:**

# One of the powerful features of R, like many programming languages, is the ability to name new variables and assign them new values.  This is done by use of an "assignment" operator.  In R, the assignment operator "<-" or "=" (two different ways to say the same thing) places values you give it into a variable you name.  Let's name a new vector variable called "NewVar" and assign it the values "1,2,3,4,5,6,7,8,9,10":

```
NewVar <- c(1:10)
```

#Assuming all went well, the console reports absolutely nothing.  This is because the assignment function lacks output that would print to the console.
#Now, let's evaluate the variable we have just made:

```
NewVar
```

#The evaluation shows that you have placed the values derived from the concatenation function c()inside NewVar.  Now let's make another:

```
NewVar2 = c(5,6,7,8)
```

# and evaluate:

```
NewVar2
```

# Easy.

**# BUILT-IN FUNCTIONS:**
We can now use "functions" to do many important things.  For instance, to calculate the "mean" (average) of a vector, use the built-in R function "mean()" placing whatever variable you want within the parentheses:

**mean(NewVar)**

# The median, with "median()":

**median(NewVar2)**

# Can you find the median of Sepal Length in iris?

**median(Sepal.Length)**

Or, how about the mean of the first 50 rows in of iris for Petal Width?

**mean(iris$Petal.Width[1:50])**

# There are many other useful functions in R, such as:

```
min(NewVar)             # minimum value in vector
max(NewVar2)            # maximum value in vector
sum(NewVar)             # adding the elements of the vector.
length(NewVar2)         # finding how many numbers occur in the vector.
var(Petal.Length[1:50]) # for the variance of Petal Length for iris setosa.
```

# What kind of variance does the function var() in R calculate? It remains for you to prototype it!
# Many more functions may be found by typing:

**help.start()**

# As you have probably noticed, learning about and remembering the syntax of a programming language such as R is a major challenge and fundamental to using it effectively. To find more information about any built in function in R, type a "?" followed by function name, eg:

**?var**

**# PARTITIONING DATA:**
#It is often very useful to look at some variables according to values exhibited by another. For instance, looking at the iris dataset:

**iris**

# one can see that data for iris species “setosa” are found in the first 50 lines, data for “versicolor” in the next 50 lines, and for “virginica” in the last 50 lines. We can calculate the number of lines, minimum value, maximum value, mean value, standard deviation, and

variance for one variable by applying the above functions.  For Sepal.Length in species “versicolor”, try:

```
length(Sepal.Length[51:100])
min(Sepal.Length[51:100])
```

# As you can see, this is somewhat tedious, and requires manually checking rows in the iris dataset to determine which belong to the species “versicolor”.

#Alternatively, one can use “==” (double equal sign indicating logical evaluation rather than assignment) and allow R to do the counting for you:

```
max(Sepal.Length[Species=="versicolor"])
mean(Sepal.Length[Species=="versicolor"])
sd(Sepal.Length[Species=="versicolor"])
var(Sepal.Length[Species=="versicolor"])
```

#An easier way to combine such functions is to use the function called “tapply” creating variables like this:

```
xbar=tapply(Sepal.Length,Species,mean)
n=tapply(Sepal.Length,Species,length)
mn=tapply(Sepal.Length,Species,min)
mx=tapply(Sepal.Length,Species,max)
s=tapply(Sepal.Length,Species,sd)
v=tapply(Sepal.Length,Species,var)
```

# and then using the “column combine: function:

```
cbind("NUMBER"=n,
"MINIMUM"=mn,
"MAXIMUM"=mx,
"MEAN"=xbar,
"STD DEV"=s,
"VARIANCE"=v)
```
#Note use of multiple lines only to make this more readable! R doesn’t care as long as you break the line properly such as at a comma.

# The result is a tabulation of these variables for each species in turn.  Note in the command above, that words in " " are used to specify labels; the symbol ' 'work also, but should not be intermixed.  Also, it is important to use the generic symbol " obtained from a simple text editor, and *not* the similar-looking “or ” obtained from a word processor such as MS Word.

**PLOTTING PARTIONED DATA:**

# The easiest plot to make for portioned data are default box and whisker plots made using the generic plot() function when you pass a factor to the function in the first position:

**plot(Species,Sepal.Length)** # Species is a factor variable in the first position of plot()

# Alternativelly, one can call the boxplot() function directly, and if desired, display each box with a waist (informally indicating 95% confidence intervals). Note the different syntax:

**boxplot(Sepal.Length~Species,notch=TRUE)**

# We can histogram each species separately by making the following Sepal.Length variables:

```
SL.setosa=Sepal.Length[Species=="setosa"]
SL.versicolor=Sepal.Length[Species=="versicolor"]
SL.virginica=Sepal.Length[Species=="virginica"]
```

# Then by plotting:

```
 par(mfcol=c(3,1))  # setting par() function for plots in 3 rows and 1 column
hist(SL.setosa,nclass=15,col="red")
hist(SL.versicolor,nclass=15,col="blue")
hist(SL.virginica,nclass=15,col="green")  # making the histograms
par(mfcol=c(1,1))  # resetting par() back to a single plot.
```

# The last line is a good idea unless you intend to continue plotting graphs in groups of three indefinitely. A similar call using "mfrow" allows graphing in rows instead of columns:

```
par(mfrow=c(1,3))
hist(SL.setosa,nclass=15,col="red")
hist(SL.versicolor,nclass=15,col="blue")
hist(SL.virginica,nclass=15,col="green")
par(mfrow=c(1,1))
```

# To allow comparison between histograms, limits based on maximum and minimum values (observed on the graphs or calculated above) can be applied to the x and y axes:

```
par(mfcol=c(3,1))
hist(SL.setosa,nclass=15,col="red",
xlim=c(4,8),ylim=c(0,10))
hist(SL.versicolor,nclass=15,col="blue",
xlim=c(4,8), ylim=c(0,10))
hist(SL.virginica,nclass=15,col="green",
xlim=c(4,8), ylim=c(0,10))
```

```
par(mfcol=c(1,1))
```

# Now for making scatter plots with multiple coded points, we make variables by extracting Sepal.Width for each Species:

```
SW.setosa=Sepal.Width[Species=="setosa"]
SW.versicolor=Sepal.Width[Species=="versicolor"]
SW.virginica=Sepal.Width[Species=="virginica"]
```

# Now we make a plot using function "plot". Limits xlim and ylim are specified to allow plotting of all points in the graph. We then add points for the others using function "points":

```
plot(SL.setosa,SW.setosa,pch=19,col="red",
        xlim=c(4,8),ylim=c(2,4.5))
points(SL.versicolor,SW.versicolor,pch='v',col="blue",
        xlim=c(4,8),ylim=c(2,4.5))
points(SL.virginica,SW.virginica,pch=22,col="green",
        xlim=c(4,8),ylim=c(2,4.5))
```

# Of course, points are color coded using "col" and different symbols are used using "pch". To find available options, consult:

```
?plot
?par
?points
?pch
```

# **SAVING PLOTS:**

# To save your histograms or plots, it is a simple matter of cutting and pasting them into your favorite word processor such as MS Word. They can then be printed out in the normal way.

# **READING AND WRITING DATA:**

# Writing and Reading data from external files is an important aspect of any statistical analysis. Simple text files are the most general way to exchange data between formats and nearly all programs have ability to do this in one way or another. To write the "iris" data table to a text file, the easiest way is to cut and paste. Open a text file editor, and then cut and paste normally. Be sure to include the first line containing names of the variables. Use your text editor to make a simple text file named "iris.txt" and place this within R's working directory. You can find out where the working directory is located by looking under "File/Change dir" on the R console.

# After writing the file, let's see how to read it back into R.   For this, we will make a new variable called "newIris".  Use the function "read.table" and then list the file.

**newIris=read.table("iris.txt")**
**newIris**

# As you can see, read.table in R has correctly interpreted the "iris.txt" file and read all the data points into the appropriate columns.  From this, you can obtain summary information like before:

**summary(newIris)**

# To import iris.txt into MS Excel, open Excel and then under "File/Open" choose R's working directory, and "All Files" in the "Files of type" box.  Open "iris.txt" and follow Excel's formatting instructions.  Click the "Delimited" radio button with "Start import at row 1" then, click "Next".  Check the "Space" box and now field delimitation is shown by vertical lines.  The lines should correctly separate each data point within its own field.  Now click "Next".  Now you can change data format or just accept the defaults, click "Finish".  At this point, everything should look like the original and you can save the file as a normal Excel worksheet.  To reverse the process and import a data file into R from Excel, it's best to have Excel write a simple text file.  Open your data in Excel, and choose "File/Save As…" Make a new name for your file such as: "IrisFromExcel" and "Tab delimited Text" in the "Save as type" box. Excel then complains that changing to text format may loose formatting information, but say "Yes" anyway. Now exit Excel WITHOUT SAVING (this preserves your original file in Excel).  Now, on the R console, make a new variable and use function "read.table" again:

**Iris2=read.table("IrisFromExcel.txt")**

# And to verify all went well:

**summary(Iris2)**
**Iris2**

**# READING FILES FROM OTHER FOLDERS/DIRECTORIES:**
# Finding files in other folders/directories on your computer can be easily done
# using the embedded browser function file.choose():
**Newfile=read.table(file.choose())**  # choose a file in the normal way!
**Newfile**
# To reset your working directory, you can use File/Change dir… from the console, or:
**setwd("c:/temp")**  # for directory "work" on the c drive.
# To find your current working directory, use:
**getwd()**

**# USING & INSTALLING PACKAGES:**
#Useful functions abound in R's extensive and ever-growing system of libraries. In general, there are two kinds of software "packages": internal and external.

#You can find internal packages available to you (as part of the base system or already installed external packages) in the R console using the "Packages" menu item and "load package". Up will pop a list and you can choose one. R then responds with a relatively long statement which is the R program language equivalent to using the menu. Don't worry about this. Following this, any function that is part of the package will be available for use.

To call an internal package from a script, use this function:
**library(car)** # loading the installed library {car} previously downloaded from CRAN.
**?car** # for documentation.

#External libraries are handled in a similar way, but they must be downloaded and installed first. Go to the "Packages" menu item and choose Install package(s). This sends you to the CRAN mirror list. Choose your favorite site. In a few seconds a Packages list will appear. Choose one, and it will be downloaded and installed to your internal package library. After that, you can load the package as described above. You only need to download and install an external package once.

To install an external package from a script, use:
**install.packages("car")** # Note quotes here!

#To download, install, and load a package by script:
**#first DOWNLOAD AND INSTALL the "psych" external library:**
**install.packages("psych")**
**#THEN DO THE FOLLOWING:**
**library(psych)**
**?psych # To find useful commands in the {psych} package**
**attach(iris)**
**harmonic.mean(Sepal.Length)**