C7041 Experimental Design and Analysis

# T01.04 R and Rstudio intro

*R is a language. Use it every day, and you will learn it quickly.*

## Tips before beginning

**Always type all commands into an R script directly**. This will reinforce the commands and make it easier to learn. Making mistakes is useful and they are easy to correct. In my experience **if you copy and paste you will learn slower and have to study harder**.

**Consider documenting concepts we cover in your R scripts by making notes** (comments) to yourself. We'll go through the process of creating and using scripts shortly.

Feel free to look around on the web for additional sources of R information if you are curious. One of my favorites for the basics is **Quick R, found at** [**http://www.statmethods.net/**](http://www.statmethods.net/)**.** Part of learning R and eventually becoming literate and then proficient is being able to discover solutions to problems you encounter by using resources in the R help system and on the internet. This is simply how R works.

You will need to get into the line of thinking that **you WILL make minor errors while learning to use R** and that **you MUST learn to figure out what went wrong by using the R help system**.

**You should work alone keeping your own script but have at least one “R buddy”.** You and your R buddy should communicate and share what each other are doing. If you have a problem, you should first ask your buddy if they can help. This sharing of the learning experience is proven to speed up learning and retention of programming and statistical concepts for non-programmers (and it is more interesting than just typing code, so I recommend trying it!).

Please do not hesitate to ask for help if you encounter a problem – let this be interactive. The first questions when you ask a question will be “what is the problem”, ”is there an error message”, “what have you done so far to fix it” (i.e., be proactive in your own problem solving, but we will build to this slowly).

**Code you type into your R script will look like this**

**Output from the R console will look like this**

## 01) Start Rstudio

I assume here you are using a Windows computer with the latest version of R and Rstudio installed.

**Download the file “Script template.R”** and save it to your “working directory” – the place where you would like to save and document your work in this module.

**Rename the template file to something sensible** that has meaning to its contents (e.g. “EDA R and Rstudio basics.R”)

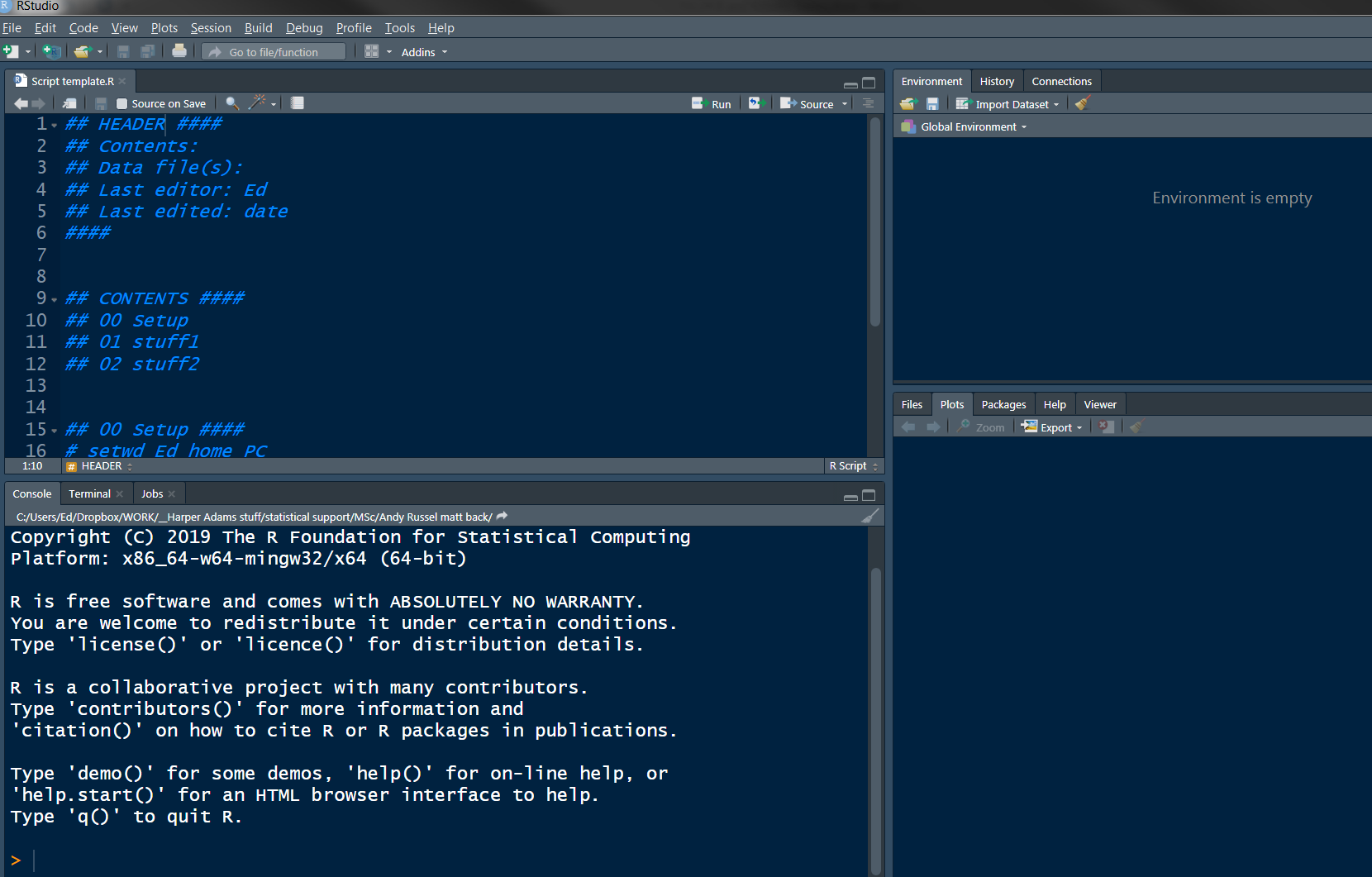
**Edit the information in the header section to personalize it**. Save scripts often, and make sure the location you save it to will be accessible later.

Look at a neighbors script and compare notes on how your HEADER section appears.

## 02) The Rstudio Graphical User Interface (GUI)

This is not like your Dad's computer program (possibly more like your granddads…)!

The Rstudio GUI is made to be flexible and customizable. Yours may not look exactly like mine in the image below (unless you change it to appear so). The main windows we will use at first are the script window (upper left in the image below), the text output window, or console (lower left), the Global environment (upper right – NB this is “R Space”), the graph output window (Plots, lower right)



R can be used as a simple calculator. Let's try a few commands.

Type:

**2 + 2**

in your script.

Make sure your cursor is on the same line as this text and hit CTRL + ENTER (the default command to submit code to R in Rstudio in Wondows) and you should see:

**> 2 + 2**

**[1] 4**

Note that the “>” symbol appears in the console on a line “echoing” your command. Also, a line number “[1]” appears on the line without output arising from your command.

You should be able to figure out the basic mathematical operation symbols

\* (multiply)

- (subtract)

/ (divide)

Try a few of these now.

R even knows some common terms like “pi”.

**pi**

**> pi**

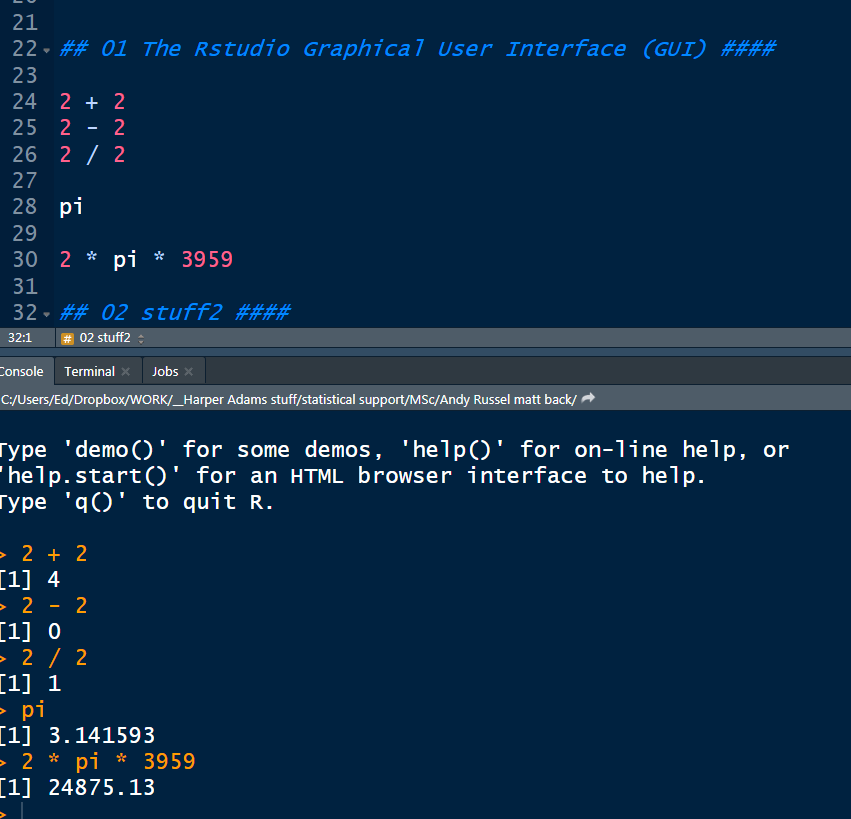
**[1] 3.141593**

The radius of the Earth is about 3,959 mi, what is the circumference (recall, circumference of a sphere is 2 times pi times the radius, 2\*pi\*r)?

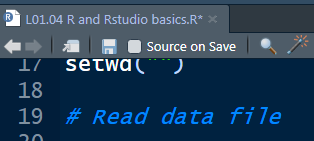
**> 2 \* pi \* 3959**

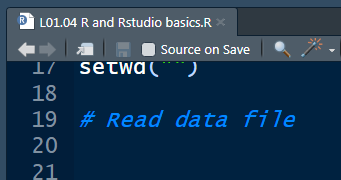
**[1] 24875.13**

Your script should look something like this now:



Make sure to regularly save your R script. Notice how the name in the script tab (upper left) has an asterisk when you make a change, and the colour of the file name changes once you save the script:





## 03) More R syntax

The R script is the main way we interact with R. Here is a bit more detail about how the rules of the R language work.

Type a command into the editor.

**(26 + 2) – (1 + 1)**

Now select it with your cursor and use the menu Edit > Run line or selection. You should see the following in the R Console:

**> (26 + 2) - (1 + 1)**

**[1] 26**

Note the round brackets affect the “order of operation”. Try moving the brackets and test this for yourself to make sure you get a feel for how it works. It works exactly like a calculator.

A few notes here.

First, the R console will ignore spaces such that

**(26 + 2) - (1 + 1)**

**(26+2)-(1 + 1)**

**(26+2)-(1+1)**

Try it out to make sure. The first way might be easier to read though!

Second, there are several ways to “run” a command you type into your script. The way I show above is only one way. Instead of putting your cursor on the same line, you could instead highlight the commands you wish to run and hit CTRL + ENTER, use the run selection button on the GUI  or use the drop down menu Code > Run Selected Line(s).

Finally, you'll want to insert comments into your script files so that you can look into them later and remember what they mean!

For example, you might type this into your script, select it all and run it:

**# The radius of Earth is about 3959 miles**

**# I remember from school that circumference is**

**# pi \* diameter and #diameter is radius \* 2.**

**# The following expression results in the approximate**

**# circumference #of Earth in miles.**

**2 \* pi \* 3959**

Select all those lines including the comments and submit it and you should still get

**> pi \* (3959 \* 2)**

**[1] 24875.13**

**>**

After the comments are echoed in the console.

## 04) Simple R "data objects".

You'll need to think in terms of "data objects" to use R, where data objects is just a fancy way of naming the named container that contains data you are working with. There are different kinds of data objects in R, for example a single number, or a list of numbers (a vector), a matrix of numbers (a matrix) or a data frame (data arranged like a regular data set in columns with names).

So far, we've seen that pi is an R variable that “lives” in the computer memory when you start R (NOTE that R is case sensitive – Pi is NOT the same as pi).

Most of the time we'll want to make our own variables though. Let's make a variable now called x and let's put the value 26 into it and run the expression. Put the following into a script and run it now.

**x <- 26**

**x**

You should get

**> x <- 26**

**> x**

**[1] 26**

Here, the **<-** arrow, or “assignment”, operator “puts” 26 into x. When we then submit x, R returns the value we put into it. The direction of the arrow is not relevant. This works similarly to how we expect the “=” sign to work, but “=” has a special use in R syntax (inside functions – as we shall see…).

Hence, these expressions will do the same thing:

**26 -> x**

**x**

**x=26**

**x**

But by R convention we'll stick to the first way.

## 5) R functions

Most of the powerful things we want to do with R are done with functions. Functions make our life easier, as long as we understand how to use them. Think of functions as tools, or perhaps just things that do work for us in R.

#### An important point:

**There are 2 tricks to learning R:**

**1) Learning WHAT function to use to do some job you want done (of course, you must first know what you want done…). Tip: to discover WHAT function solves a particular problem, we typically ask someone for help in some form, or else use Google.**

**2) Learning HOW to use a specific function. Tip: To discover HOW to use a function, we typically use the internal R help system**

R functions exist in the form:

FunctionName(the user puts stuff into here and usually expects to get something out)

**Functions usually require you to specify the data** you want the function to act on, along with any other options or specifications that might be required by that particular function. Some of the options are set to default values, which means that if you don't specify them the defaults will be assumed. It is fairly important to understand what a function does and what options are available for it.

The best way to learn about a function quickly and get a grasp of what options are available is to use the R help function **help()**. Using help on a function name (e.g., **help(help)**) gives you a brief definition, prints it in use with assumed defaults, and a list of explanations for the optional arguments. Usually there are examples as well. Don't worry if this is a little confusing now, you'll get better with practice.

Let's try a few functions to see how they work.

Type and run the following

**attach(iris)**

**names(iris)**

**mean(Sepal.Length)**

You should get

**> attach(iris)**

**> names(iris)**

**[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"**

**> mean(Sepal.Length)**

**[1] 5.843333**

A lot is going on here. First, “iris” in an R data object that is one of the stock data sets that comes with the basic package. You can print the whole dataset by just typing its name.

**> iris**

**Sepal.Length Sepal.Width Petal.Length Petal.Width Species**

**1 5.1 3.5 1.4 0.2 setosa**

**...**

**149 6.2 3.4 5.4 2.3 virginica**

**150 5.9 3.0 5.1 1.8 virginica**

Second, “iris” is a special kind of R data object called a **data frame**, which usually has columns of variables with variable names.

Here

**> attach(iris)**

uses the **attach()** function to tell R that iris is a data frame and to make the variable names INSIDE iris available for use with commands.

Here

**> names(iris)**

**[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"**

uses the **names()** function to return the variable names from the data frame. The iris data set has 5 variables – the first 4 are measurements of different flower parts. The 5th variable name is the name of the flower species.

Here

**> mean(Sepal.Length)**

**[1] 5.843333**

returns the mean value of all the values in the variable “Sepal.Length”.

## 6) Help

Let's face it, we all need help sometimes. Luckily in R we can invoke **help()**. Try it.

For example, **help(mean)**

**NOTE: when you get to this point, ask Ed to demonstrate a help page to the whole class!!**

What should come up is a help page defining and describing the **mean()** function, including a list of “arguments” that can be provided and examples. Don't forget the **help()** function...

To browse some sources of help, use help.start().

## 7) Other R variables

Above, we put a single value into the variable x. Critically, you can use R to make variables that store “vectors” (i.e., a list) of numbers or “matrices” (i.e., a table). In fact, you can and should think of any such variables in R as data “objects”, into which you can put many different kinds of data.

First, we'll use the c() function to combine several numbers into a vector or list of numbers in a single R object.

**y <- c(78, 55, 108, 63, 90, 67, 59, 81, 80, 77, 65)**

**y**

We can try some summary statistics.

**mean(y) # mean lecturer age in Crops, excluding Ed**

**sd(y) # standard deviation of y**

NB if you need to refresh yourself about mean, standard deviation, etc., let’s chat about it or Google now. We will discuss these things a bit, but it is probably review for you.

## 8) R graphics

We can graph our variable y

**boxplot(y) # plot**

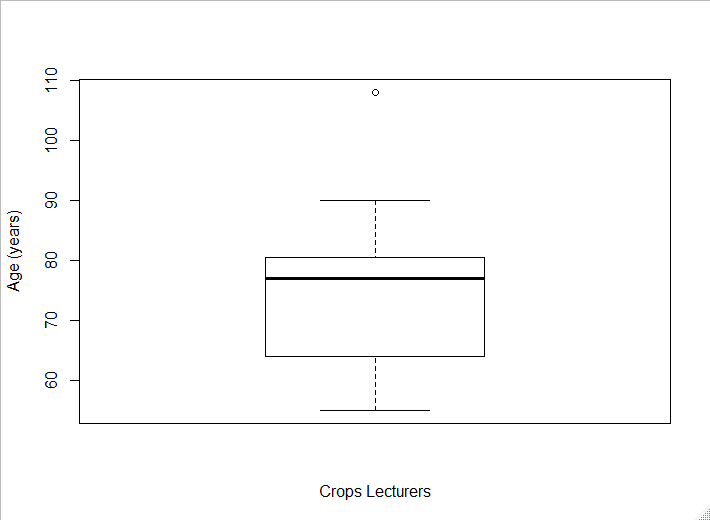
If you look at the boxplot() function definition

**help(boxplot)**

you should see that it is possible that we can add some axis labels

**boxplot(x = y, xlab = "Crops Lecturers", ylab ="Age (years)")**

NB inside the brackets we are assigning values to function “arguments using the “=” sign. This is the special use reserved for “=” in R.



If you look at the function description for **boxplot()** on the help page, you'll notice that there are several optional bits of information that we used the second time we ran it. In addition to the data, “y”, we passed a legend title for each axis using the x label command xlab and the y lab command ylab.

Note that these commands are part of the **boxplot()** function and that the labels we used were enclosed in double quotes.

## 9) Working with data

There are several ways to do most things in R, and editing data is a good example of this flexibility.

Try this

**x <- 11**

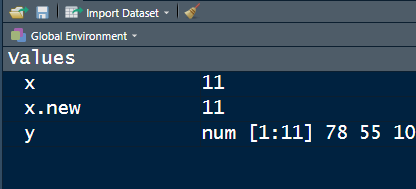
**x**

**x.new <- edit(x)**

An Editor window should pop up. Change the value, choose save. Now check that your changes have been saved to the new variable by typing **x.new**. Note here that we saved our edited value to something new. We typically do not edit data like this in R (instead using MS Excel or Libre Office Calc, the latter free and open source), but it is an option.

Also note that earlier we made a variable “x” which was probably still in memory. When we changed the value, R let us do it with no warning. No problem... as long as you understand what you are doing!

You can see all the variables in the Global Environment, a/k/a R Space:



You can check what variables you have been working with that are held in memory using the list objects function **ls()**. Try it now. The **ls()** function is handy to keep track of data objects you have floating around in memory during an R session, and once you get used to it, is quicker than scrolling through the Global Environment window (when you have a lot of variables).

Now try this input

**x<-1:10**

**x**

What do you think the “:” operator does?

## 10) Data frames

A data frame in R is what we typically think of as a data set, where there are several rows and columns of data. Typically each row contains measurements from an individual unit or subject (like a single flower) and each column is a different variable (e.g., measurement of a different part of each individual flower).

We've already encountered the data frame called “iris”. Notice that iris was not listed earlier when we used **ls()** – that is because it is part of the base package and lives on the hard drive. Still, you can edit it…

Let's make a new dataset that contains only the measurements for the first species. First, we'll get the species names in the Species variable.

**attach(iris)**

**# attach() makes the variable names**

**# INSIDE a data frame visible in R Space**

**names(iris)**

**summary(Species)**

**> attach(iris)**

**> names(iris)**

**[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"**

**> summary(Species)**

**setosa versicolor virginica**

**50 50 50**

You can see that there are 50 lines of data for each species.

**iris**

Learning your way around a matrix or data frame can be very useful in R. You can actually "reach in and grab" individual values using an index system.

For example if you had a data frame with columns and rows called x:

x[row, column] - here x is the name of the data frame or matrix, “row” is the row number and “column” is the column number.

R is very flexible with how matrices can be handled. The first 50 rows are data for the setosa species.

**iris.setosa <- iris[1:50, ]**

**edit(iris.setosa)**

What did we just do? The square brackets in the command iris[1:50, ] indicate that we want the rows 1 to 50 of the iris data frame. Because we didn't specify the column numbers, we get them all (there are 5). Thus, iris[1:50, 1:5] would return the same thing.

Try

**iris[1,3]**

**iris[1,5]**

**iris[1,6]**

**> iris[1,3]**

**[1] 1.4**

**> iris[1,5]**

**[1] setosa**

**Levels: setosa versicolor virginica**

**> iris[1,6]**

**NULL**

Do you understand what gets returned? Why does iris[1, 6] return NULL? If unclear, ask and let's discuss...

You can confirm the number of columns and number of rows in a data frame like this

**ncol(iris)**

**nrow(iris)**

**ncol(iris.setosa)**

**nrow(iris.setosa)**

Let's play with these commands a little bit to see how R handles and “thinks about” data.

**eggs.day.1 <- 5**

**eggs.day.1**

**nrow(eggs.day.1)**

**ncol(eggs.day.1)**

NOTE that variable names can be anything we want – sometimes making informative variable names is best. Also note here that single value variables aren't indexed with rows and columns

**eggs.day.1 <- c(5,6,4,7,5)**

**#note we write right over our initial variable**

**eggs.day.1**

**nrow(eggs.day.1)**

**ncol(eggs.day.1)**

Let's force the data vector into a data frame using the **as.data.frame()** function and try again.

**as.data.frame(eggs.day.1)**

**ncol(as.data.frame(eggs.day.1))**

**nrow(as.data.frame(eggs.day.1))**

**> as.data.frame(eggs.day.1)**

**eggs.day.1**

**1 5**

**2 6**

**3 4**

**4 7**

**5 5**

**> ncol(as.data.frame(eggs.day.1))**

**[1] 1**

**> nrow(as.data.frame(eggs.day.1))**

**[1] 5**

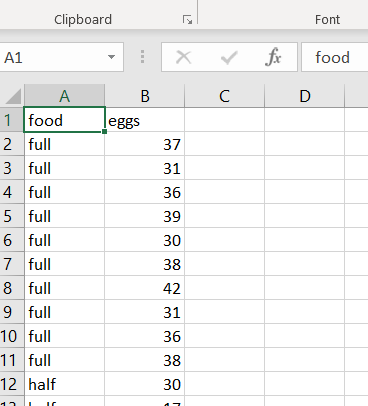
Now you have a data frame (if you save it into a new data object).

## 11) Reading and writing data - i.e., getting your data into R

Ultimately, you will need to manipulate regular data files both by reading them and by writing them. The main format we will use is a Microsoft Excel file ending in xlsx. You can read and write files of most types using R, but it makes sense to stick with Excel because it is so easy and common, and it is accessible to the vast majority of users.

You should find a data file on The Hub called butterfly.eggs.xlsx. Download it to your R working directory or another file location you can keep track of (make sure you take care to do this!). Open it with a spreadsheet program.

You should see something similar to the below



The data set contains the number of eggs laid by butterflies (the eggs variable). This was the result of an experiment where two groups of butterflies were reared with unlimited food available as larvae. One group was reared using high quality food (the “full” group) and the other was reared on food that was nutritionally poor (the “half” group).

To read an xlsx file we need to load our first package, called “openxlsx”

Install.packages(“openxlsx”, dependencies = TRUE)

**help(install.packages)**

**install.packages(pkgs = "openxlsx", dependencies = TRUE)**

**# load the package**

**library(openxlsx)**

**butter.eggs <-read.xlsx(xlsxFile = file.choose(), header = TRUE, sep = ",")**

**butter.eggs**

Note here that there are a few different ways you can get in data – here are a few I encourage you to play with. Use the relative path to your file

**butter.eggs <-read.xlsx(xlsxFile = file.choose())**

**butter.eggs**

You read in your first data to R!

You can get and share data in a similar way between R and many different programs and data file formats. We'll mostly stick to the above.

Let's make a new data set now called butter.eggs.new. In it, let's combine the food factor variable with a log base 10 of the eggs variable. Here is one way to do that:

Note that the **$** symbol can be used to "reach inside" of a data frame object and specifiy a variable without using attach()

**> log.eggs <- log(butter.eggs$eggs,10)**

**> butter.eggs.new <- data.frame(food, log.eggs)**

**> butter.eggs.new**

**food log.eggs**

**1 full 1.568202**

**2 full 1.491362**

**...**

**20 half 1.462398**

**21 half 1.477121**

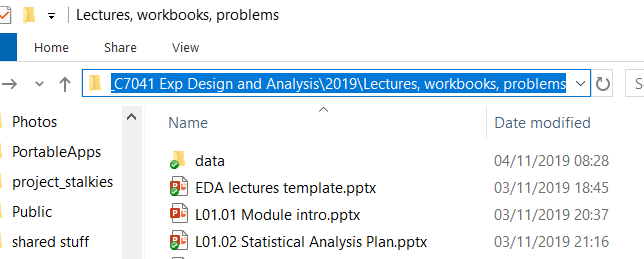
## 12) Set and get working directory

**help(setwd)**

**help(getwd)**

**getwd()**

**DEMONSTRATION here for WHOLE CLASS**



Clicking once in the bar in Windows Explorer will select your directory PATH. Note the folder names are separated by a backslash “\”.

**# step 1 paste in your path (don’t submit code yet)**

**setwd(“D:\yourpath\folder1\folder2”)**

**# step 2 in Windows, change backslashes to forward slashes and submit code**

**setwd(“D:/yourpath/folder1/folder2”)**

**getwd()**

Now, let's export the new data frame to a .csv file in our working directory.

**write.table(butter.eggs.new, file.choose(), sep=",")**

You'll have to give the file a new name. I suggest “butterfly.eggs.new.csv”. Open the file in Notepad to have a look. Don't worry about the first row of numbers in double quotes – these are the row “id names”.

Now let's read in the new file to double check ourselves – this is always prudent!

**getwd()**

**help(write.xlsx)**

**write.xlsx(x = butter.eggs, file = "new.data.xlsx")**

The new.data file should be in your working directory. Check this and open it – yell if you need assistance.

## 13) Saving your work

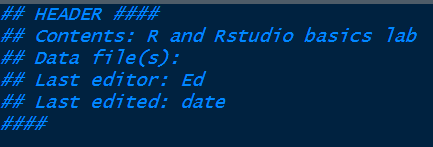
R is meant to be used as an “environment”. By default when you stop and start the program, you get a clean and standard environment. You need to make a little effort to customize your workspace. This will be very important to keep track of our lab work from week to week.

First, let's look at the default working directory using the get working directory function.

**getwd()**

You want to set it to something that will be the same every week, and where you will store your work. For example, make a directory **(i.e., do this now if you have not already done so)** called “Rwork” somewhere on the computer, network or a pen drive (I suggest the network drive).

Save your script file, making sure your HEADER info is filled in, like:



Get into the habit from now on of running all R commands from a script file and insert concise comments about things you might want to remember, refer to, etc.

## 14) Quit R session

Make sure your script is saved. Once that is done you can just close Rstudio.

Here you have an option to “save your workspace”. This saves info about your working directory and the data objects in your Global Environment. Choose “no” for now.