Today’s lab covers a number of practical and logistical issues for working with R in the context of measurement. I know that most of you have some familiarity with R, so quite a bit of this is likely review, but sometimes it’s helpful to have a refresher and/or to learn different ways to perform similar tasks. Although we likely won’t go over all of this in detail in lab, I thought it might still be helpful for you to have my reference guide to R basics:

* R and RStudio
* The overall logic of R
* Objects in R
* Functions
* Packages – what they are, how and why to get them and use them.

As I’m not quite sure what you used in previous classes, and I want to make sure everyone has a good grasp of these tasks, we will go over:

* Getting data into R
* Working with data in R
  + Missing data
  + Subsets
  + Reverse coding
  + Calculating new values
* Getting data back out of R
* Making simple tables and crosstabs
* Easily obtaining descriptive statistics

**R**

R is a free, open-source “statistical computing environment” – somewhere between software and a programming language – that is increasingly widely used by researchers (and many others). It is quite legit and analyses using R appear in most major journals. There is a learning curve (as there is with any statistical software), but the payoff (free and very flexible) is really quite nice.

R can be very transparent – it is essentially a sophisticated calculator, so you can enter equations fairly directly and see the results as if you were calculating them by hand. However, for most common analyses, you don’t have to – there are a number of “packages” already written that perform those calculations.

More and more, there are good resources available to help you get started using R – I’ve listed several on the course website, and the Raykov & Marcoulides text often gives R examples for measurement analyses.

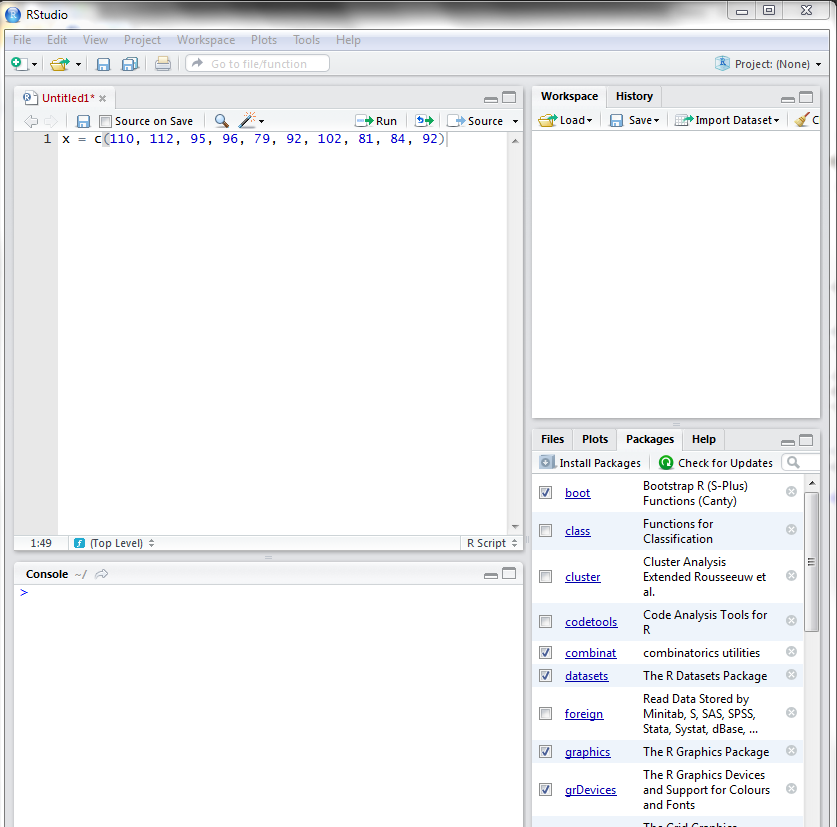
For lab today, my goals are:

* Introduce you to the logic of R
* Show you how to write scripts and view objects in Rstudio
* Show you how to get data into R
* Perform some simple calculations to illustrate matrix algebra concepts
* Use matrix algebra and the lm() package to perform a very simple linear regression analysis.

**RStudio**

RStudio is separate from the main R package; it is an add-on or extra, but I find it really, really useful. If you just install R, you will be interacting directly with the R **console**. This is like the old DOS command line, if you have any experience with life before Windows. In the console, you enter commands one line at a time, and R processes them immediately. If you need to change something, you enter the relevant line again. This can be frustrating when you are setting up a new analysis, tinkering with data, etc., so it is generally nicer to use **scripts**, where you write out all the commands in a text file and then enter them into R (like syntax in SPSS).

RStudio makes this process easier by integrating your script file, the R console, a list of all the objects you’ve created (more on this in a minute), and a few other things into one organized window:



This is the console. When you run a command, it will show up here, as will the results. If you create an object, it does NOT show up here unless you ask R to display it (by running just the name of the object).

This is your script file. You can set up analyses in this section and wait to run them, or rerun previous analyses, etc.

This section has tabs, just like a web browser; you can work with multiple scripts if you want to.

This section keeps track of your objects. Any time you create an object, it appears in this list. You can click on it to see it. – make sure it is what it ought to be.

This section includes lots of different information. Right now, it tells me which packages I am currently using and which other ones I have available. (I’ll explain packages later).

**The Logic of R**

R is what’s known as an “object-oriented” language. Basically, this means that you tell R to define “objects” – which can be variables, formulas, matrices, data sets, etc. – and then do things with those objects. Some important things to keep in mind:

* Objects don’t exist until you create them. R has no idea what “data” means until you define an object called “data.”
* You can call your objects whatever you want. You can call your variables x, y, and z, or Alvin, Simon, and Theodore. If you really want to, you can call a formula “dv” and a dependent variable “matrix.”
  + In other words… R is only as smart as you tell it to be. It has very few a priori assumptions about the objects you create.
  + As with SPSS, etc., it’s often helpful to give your objects descriptive names, such as full.data or mean\_perf.
  + If you are adapting an example, you can (and often should) use your own object names! R is not looking at your example. Just be sure you know what the objects in the example *mean*.
* As in most software programs, you cannot have a space in the middle of an object name. You can, however, use a period (.) or underscore (\_) to make your object names more readable (as in the examples above).
* R is **case sensitive**. Once you name an object, you need to enter it **exactly the same way** every time.
* R is **not** generally picky about amounts of white space **between** objects. You need *a* space (or R will think you’re naming a new object), but you can have 2 or 3 or 27 spaces and it will not matter.
  + The exception to this is with string (text) objects, where R is looking for the exact literal thing you entered. For example, if you are trying to select employees whose status is “Part Time”, but you type “Part Time” instead, R will tell you there aren’t any of those.
* R uses <- to *assign* or define an object.
  + If you already have an object, and you assign something different to that same object name, it will overwrite the original object. This is useful when you need to correct or change something (say, drop an item from a scale), but causes problems if you inadvertently try to name two objects the same thing, because you will lose one.

The different types of objects you can create in R include:

| **Object Type** | **What It Is** | **How You Define It in R** | **Result** |
| --- | --- | --- | --- |
| Single values | Just what it sounds like! | x <- 1 | 1 |
| Vectors | A set of values. | x <- c(1, 2, 3, 4)  *c here means “concatenate”, or you could think about it as “combine”* | 1  2  3  4 |
| Matrices | A set of vectors (columns), grouped together. | x <- matrix(c(1, 2, 3, 4), 2, 2)  *Basically, this function creates a vector (compare above) with 4 elements and then wraps it into a matrix. The first part of the function makes the vector, the second part says the matrix has 2 rows, and the third says it has 2 columns.* | 1 3  2 4 |
| Arrays | A set of matrices – data in 3 (or more) dimensions. | x <- array(c(1, 2, 3, 4, 5, 6), c(1, 3, 2)  *This time, the second part of the function gives the dimensions of the array: 1 rows x 3 columns x 2 layers. We won’t use these often but they can be handy.* | [1]  1 2 3  [2]  4 5 6 |
| Strings | Text values. | x <- “Hello!”  x <- c(“a”, “b”, “c”)  *You indicate that something is a string value by putting it in quotes. Otherwise, if you enter text, R thinks it is an object and gets upset that it can’t find it. Also, if there is text anywhere in your vector or matrix, R will treat the entire matrix as text (i.e., you can’t do calculations with it). This is not true of data frames.* | “Hello!”  “a”  “b”  “c” |
| Data frame | A matrix of data (2 dimensional), typically large, that can mix both numeric and string data. | x <- as.data.frame(matrix(c(1, 2, 3, 4), 2, 2)) **–OR– read in your data.**  *Data frames are more like Excel or SPSS. You can have both text and numeric values, and you can name your columns, which makes them much easier to work with. Note that in this simple example, R automatically numbers the rows and names the columns “v1” & “v2.”* | V1 V2  1 1 3  2 2 4 |
| List | A set of objects that you want to go together – can combine different types of objects. | x <- list(p, q, r)  *p can be a matrix, q a vector of string values, and r a single value – or any combination you want. Lists are important because many packages save their results in lists; you need to know how to get the information you want.* |  |

**Functions**

A **function** takes an object and does something with it. R includes many built-in functions, and it also allows users to develop their own**.** We won’t build functions within this course, but we will definitely use them! R is open-source, which means that many smart and kind people write functions and then share them with the rest of the world (for free). These are often combined into **packages** (more on those next week). A package is just a collection of functions that logically go together in some way, and that have been grouped together into a set that you can download all at once, typically with a help file.

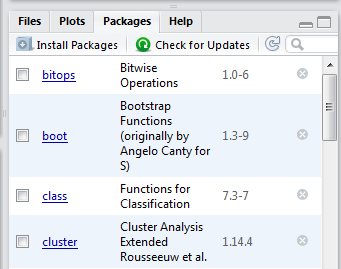
Important things to know about functions:

* Functions can be identified because they **always** end in a set of parentheses. The parentheses tell you this is a function and not just an object.
  + E,g,, mean()
  + If you just type mean, R will be confused; it will be looking for an object called “mean”.
* For most functions, you need to put something *in* the parentheses. This is called an *argument*, just as with the functions we’ve used in Excel. The argument tells R what data (object) it is to perform the function on.
  + mean(x) gives you the mean of x.
* Many functions take multiple arguments. This means that (a) the function needs multiple pieces of information from you in order to run, and/or (b) the function has options, which you can choose if you wish.
  + To understand all of the possible arguments and their values for a particular function, the best thing to do is to look at the help file for that function.
    - For example, help(mean) brings up the help file for the mean() function. The help file will appear in the lower right pane of RStudio.
  + Where this often gets confusing is that you don’t *have* to enter every argument all the time. Many arguments have default values; if you don’t enter anything for those arguments, R will just use the default. For example, the default for the cor() (correlation) function is to use all observations, which will give you a result of “NA” if you have any missing data. If you don’t have missing data, just cor(x, y) is fine. If you do have missing data, you need cor(x, y, use = “complete.obs”) to drop data listwise. If you were looking for the correlation matrix of a data frame, you could use cor(x, use = “pairwise.complete.obs”) to drop data pairwise.
  + Also, you can often omit the name of the argument, **if you enter the arguments in the correct order** (the order in which they are listed in the example). For example, in the matrix() function:
    - matrix(x, 2, 3) is the same as matrix(x, nrow=2, ncol=3)

**Packages**

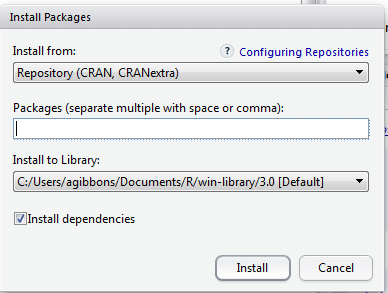
Packages are kind of like “apps” for R. They are collections of functions that all go together in some way, written by the same person or group so that they play nicely together. You may only actively use one or two functions in a package, but the package might also contain sub-functions that are necessary for the functions you care about.

The beautiful thing about R functions is that they are free. If you want a package, all you need to do is download it. RStudio allows you to do this very easily. In the lower right pane of RStudio, one of the tabs is labeled “Packages.”



The words underlined in blue are the names of the packages I currently have installed. You might have different ones; since you don’t install them until you need them, that’s quite likely and not a cause for concern. To get a new package, click on the button that says “Install Packages.”

Once you click that button, you will see a dialog box like this:

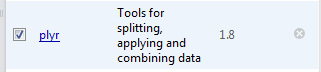


Just type the name of the package you want in that middle box; you will see it start to autocomplete with the names of existing packages. Keep that “install dependencies” box checked – this means R will also download and install any *other* packages that your package needs in order to run properly. For practice, download the package “psych.” If psych is already installed, download the package “plyr” just for the sake of practice. Don’t worry yet about what it does.

**Loading Packages**

Installing a package means it’s available on your computer for use, but it doesn’t tell R that you want to use it in this particular analysis at this particular time. In order to use a package, you have to first **load** it (theoretically, this makes R run faster, because it doesn’t have to remember a bunch of functions it isn’t going to need). It’s not uncommon to get error messages that a function can’t be found because you forgot to load the package. R will also unload all but the very essential default packages when you close it; if you start an analysis, close out, and come back to it later, you will need to reload the packages again before they will work.

There are two good ways to load a package in RStudio. The first is to click the checkbox next to the package name on the Packages tab:



The other is to run the code:

library(plyr)

Both do exactly the same thing. The first is useful when you just want to run the package right now. The second is useful when you’re writing a script that you intend to use again in the future, because if you include it in your script, you won’t annoy yourself by forgetting to load it (can you tell I speak from experience here?). But either does the trick.

Another handy thing is that clicking on the blue name of the package takes you straight to the official help file for the package. All the official help files have the same structure. They will start with a table of contents listing all of the functions in the package. When you find the one you are looking for, it will give you a description of what the function does, a list of the function’s arguments (remember those from last time), and at least one example of how to enter the full function. Some of these are quite detailed and helpful; some contain a lot of information about technical details that you don’t really need to worry about. But they are an excellent starting point for figuring out just what you need to do to *use* a function.

**An Incomplete List of Useful Packages**

There are tons and tons of packages. The more you use and read about R, the more you will discover packages that do things you want to do, and you can try them out as much as you want. Some of my favorites are:

|  |  |
| --- | --- |
| psych | This package is *designed* to support measurement analyses! It includes useful features like a quick function to reverse code items, and it will calculate some of the things we’ll be concerned about later on. It is sometimes a little old-school in its approach to measurement; this is a good place to read the help file carefully to determine whether it is really doing what you want it to do. |
| ggplot2 | Pretty graphics are beyond our scope in this course, but they are invaluable in consulting, presenting data in articles and conference talks, etc. This package has its own challenging learning curve (you have to learn what the author calls “the grammar of graphics”), but it is absolutely brilliant and you can do amazing things if you master it. |
| reshape2 | By the author of ggplot2, this package is excellent for reorganizing data (e.g., from short to long format for multilevel or repeated measures analyses). |
| foreign | Enables you to read in files from SPSS and other common statistical software packages – useful if you need to analyze files in that format. |
| lme4 | Outside the scope of this class, but useful for multilevel models (but with a very different logic & syntax than Mplus). |
| lavaan | The poor man’s Mplus – it doesn’t have all Mplus’ bells and whistles, but it performs many basic analyses (e.g., CFAs) in a very similar way. |

**Getting Data into R**

There are lots of ways to get data into and out of R, but that doesn’t mean it’s easy. Some strategies for converting files between R and other programs can be problematic; you can end up with more complicated data structures than you wanted, or you can convert things incorrectly and lose data. That sounds scary, but I can tell you how to alleviate the issue: **keep it simple.**  Stick to straightforward and easy methods of reading in your data.

* Convert your data to .csv format. This is easy to do if you are working with data in Excel and not much harder if your data is in SPSS.
* Use the read.csv() function. This seems to be (according to broad consensus) the simplest and most accurate way to import data into R.
* As we did in class last week, you can use:

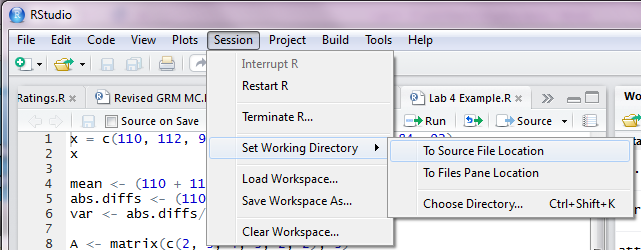
read.csv(file.choose(), header = TRUE)

to open a dialog box that will allow you to select any .csv file on your computer. This is a quick and easy way to do it.

Alternately, if you want to make sure R is referring to a **specific** data file every time, you can put the file name in that first argument instead. *If you are using R on a Mac, the “file.choose()” function often doesn’t work properly, and so you’ll need to specify your file name every time*:

read.csv(“My Raw Data.csv”, header = TRUE)

But there’s a catch. R will look for that file **in its current active folder** (called the “working directory”). If your file is in a different folder, R will tell you it can’t find it. There are two possible solutions to this. One is to specify the full file path (“C:\My Documents\Measurement\My Raw Data.csv”) instead of just the file name. This gets cumbersome if you have complicated folder structures (like me!), if you move things around, or if you are sending your script to someone else who may save their data in a different place than you did. The other solution is to remember to tell R to use the folder you want it to. In RStudio:



Click “Session,” then “Set Working Directory.” If you choose “To Source File Location,” R will change the working directory to the folder you saved your script in! If you choose “Choose Directory,” you can pick your folder manually. R will keep using that folder for the rest of this session (unless you change it again), but you will likely need to set it again next time you open R.

By default, read.csv() will convert your data into a data frame. This means you can have both text and numeric values in the same data set; just like SPSS, if you have any text values in a column, it will treat the entire column as text (not numbers). Including the “header = TRUE” argument tells R to read the first row of your csv file as variable names.

**Missing Data**

If you have empty (missing) cells in your data, R will code them as “NA.” Some R functions naturally ignore NA values; some of them are not set up to handle missing data and will not run (you will get a result of NA for your calculation). You need to decide how you want to handle missing data. If you want to delete missing data listwise (meaning, you will drop any case that is missing even one value), you can use:

data2 <- na.omit(data)

to create a new version of your data file with ALL cases that had missing data dropped. If you have only a couple of missing data points, this may be fine. If you are missing a lot of data, or if a lot of participants are missing even a little data, this is a very bad idea as it will eliminate a large portion of your sample.

If you are concerned about missing data, check the help file for your function to see what your options are. One function we will use a lot is the cor() function (to get the correlation between two variables or the correlation matrix for a set of data). It requires an argument to tell it how to handle missing data:

cor(data, use = “complete.obs”)

This drops data listwise, just as na.omit() does, or you can use:

cor(data, use = “pairwise.complete.obs”)

to drop data pairwise (that is, it will include all the cases that have data for both variables in the pair you are correlating, even if they are missing data on other variables).

**Working with Data in R**

**Subsets**

We often want to work with subsets of a larger data file. For example, when we are calculating a correlation matrix, it would be much easier to say “give me all the correlations in this subset” than to have to say “give me the correlations between variables A, B, C, D, E… and Q” every time. So it is helpful to be able to slice our data up into smaller chunks. You can do this by using [brackets] to tell R which *part* of the data frame you want.

One way to do this is by telling R you want certain rows or columns, by the *row* or *column number*. Remember that R always processes this as [row, column]. So:

data[14, 27]

gives you the data point in Row 14, column 27. But you don’t usually want just one data point; you would typically want all the rows for a set of columns (say, items). If you leave one element of the brackets **blank**, R assumes you want all of it. So:

data[, 27]

gives you **all** of the rows in Column 27 and

data[14,]

would give you all of the columns for Row 14. To get all of the rows for a set of columns that are lined up next to each other, you can use:

data[, 27:30]

gives you all of the rows for Columns 27-30 (inclusive).

The downsides to this are (a) it only works if the columns you want are already next to each other, (b) you have to count to figure out exactly which columns you need, which is a bit of a pain when you have 100 variables!, and (c) if you change anything in your data set, you have to figure out column numbers all over again. So although this is quick and easy, it is probably better (and less confusing) to work with column names. If you used header = TRUE when you read in your data, you already have column names from your .csv file. If not, you can add them by creating a vector of column names, making sure they are in the right order:

colnames(data) = c(“ID”, “Item 1”, … “Outcome”)

To select just a subset of columns, you create a vector listing just the columns you want (now order doesn’t matter):

variables <- c(“Item 1”, “Item 2”, “Item 3”).

And then put that vector in the “columns” spot in your brackets:

working.data <- data[ , variables]

Again, make sure you leave that blank space before the comma to make sure you get all the rows.

Finally, sometimes we don’t want all the rows – we want to analyze only those cases that meet certain criteria. You can select cases based on the value of a certain variable by using which(), like this:

working.data <- data[which(data$gender == “Female”, variables]

That $ clearly identifies your variable: column “gender” in dataset “data”. The double equals sign means “exactly equal to” and helps keep R from being confused about string variables. You could also select, for example, all the above-average scorers:

working.data <- data[which(data$score > mean(data$score), variables]

The possibilities are infinite. ☺

**Reverse Coding**

In measurement we often deal with items that need to be reverse coded. If you have just a few of these, you can do them manually by subtracting the item from (your highest scale point – 1) and either **adding** or **overwriting** the item in your data frame. If you have a 7-point scale:

data$item1 <- (8-data$item1)

will **overwrite** the initial item with the reverse-coded item, and

data$item1\_R <- (8-data$item1)

will create a **new** variable in your data set that is the reverse of item 1. The choice is yours, but of course it will affect your later script, so just remember your choice.

If you have a lot of items, this can get tedious. There is a function in the psych() package that will reverse code a lot of items at once. To use it, you need to do a couple of things: (1) select a *subset* of your data that contains only the items in your scale, as described above, and (2) create a vector of “keys” to tell the function which items should be reversed and which are normal. This is a vector of 1s and -1s (-1 indicates an item to reverse), in order. So if you had 10 items and wanted to reverse items 4, 6, and 7, you would use:

r.data <- reverse.code(data, keys = c(1,1,1,-1,1, -1,-1, 1, 1,1)

You might have noticed that I left an extra space between items 5 and 6. That’s not necessary, but in a long list of items, it’s easy to lose track of where you are, so I will often use that white space to visually “chunk” my items so it’s easier to see which item is #6, etc.

Note that you need to assign this function to a new object for it to do you any good; otherwise R will reverse code the items but not store them that way. Also, by default the reverse.code() function produces a matrix, not a data frame; they look a lot alike, but some functions will not work properly with a matrix. So it’s a good idea to convert it to a data frame, like this:

r.data <- as.data.frame(r.data)

**Calculating New Values**

If you want to calculate new values, you do it just as we did the manual reverse coding above:

data$scale.sum <- data$item1 + data$item2 + data$item3

or

data$t.score <- (data$z.score \*10) + 50

Getting scale scores is a little more complicated; if you ask for mean(data$item1), R will calculate the mean *across all rows*, not the mean for each participant. But there’s a function for that:

data$mean <- rowMeans(data)

as long as you want ALL the variables in data to be included in that mean – otherwise:

data$mean <- rowMeans(data[,c("pred.1", "pred.2", "pred.3")]))

**Getting Data Back Out of R**

Finally, it’s also good to know how to get data **out** of R – the output is not always pretty, and you might want to format it or manipulate it or even (gasp!) transfer data to other programs.

If the thing you want to transfer out of R is a vector, matrix or data frame, you can write it to an Excel format using:

write.csv(data, file.choose())

In the dialog box that pops up, type the name you wish to give the file (if you use an existing file name, R will overwrite that file!) and **make sure it ends in “.csv”!!**

If you are using R on a Mac, again, file.choose() does not work well here! Instead, type your file name (in quotes) directly into the write.csv() function:

write.csv(data, “My Analysis.csv”)

If you want to save something that does not lend itself nicely to a .csv format (e.g., the output from a function like alpha()), you can save it to a text file or even a Word doc by using sink(). The sink() function tells R to print whatever it would have showed you in the console into a file instead. This means you have to plan ahead a little bit:

sink(file.choose()) *or include the file name if you have a Mac!*

The file extension you type when you name the file determines what kind of file it will be (.txt = notepad, .doc = Word).

**Then** run the script you need to get the output you want, e.g.,

alpha(my.scale)

And then you need to run sink() **again** to tell it to STOP sending output to that file:

sink()

**Simple Tables**

You know by now that one of the first steps in **any** data analysis is to take a look at your descriptive statistics. The psych package has a handy function for this called describe() (remember you have to **load** the psych package to use this, if you haven’t already):

describe(data)

This will get you a table showing you the sample size, means, standard deviations, skewness, kurtosis, and many other commonly used descriptive measures.

If you want a quick frequency table of just one variable, use the table() function, like this:

table(data$variable)

Or you can get a crosstab of two (or more) variables:

table(data$variable1, data$variable2)

If you’d like to quickly calculate mean scores (or other summary statistics) on a continuous variable by group, you can use the aggregate() function:

aggregate(data$num.variable, by = list(data$group.variable), FUN = mean)

If that last argument is “FUN = mean,” you’ll get mean scores for your continuous variable at each level of your group variable. You could also use “FUN = sd” or “max” or “min” or several other common summary statistics.

**Lab Exercise:**

1. Download the file “Lab 5 Data.csv” from Canvas and save it somewhere on your computer.
2. Set your working directory, then read in the data file using the (short) file name.
3. Find the descriptive statistics for all variables in the data file, so you can see what you’re working with.
4. Reverse code the 2nd, 5th, and 6th items.
5. Find the descriptives one more time to verify that the reverse scoring worked.
6. Calculate a mean scale score for each participant for each item and add it to your data set as a new variable.
7. Find the overall descriptives for your scale score.
8. Now, find the scale score descriptives separately for males and females.
9. Write your data set, including the scale score variable, to a .csv file.
10. Use sink() and rerun steps 3-7 to print the results of your analyses to a Word file. Turn in the Word file AND your RStudio script file.