

# Easy R Introduction Tutorial

**Overview:** This document is a quick introduction to using R to write data processing scripts that uses finding soil moisture from oven dried samples as an example. These are calculations that are easy to do in excel, but that's exactly what makes it a good starting script; you can check your answers and understand the steps both ways.

There are four sections to this tutorial:

- 1) Setting up RStudio and your Data
- 2) Soil Moisture Calculations Script
- 3) Data Quality Check Script
- 4) Quick Graphing in RStudio

If you only want a basic introduction, feel free to stop after section 2; the soil moisture calculations are a good introduction to writing scripts in R on their own. Section 3 is useful if you want to learn more about how to write data quality check scripts and section 4 is a very quick introduction to graphing in R.

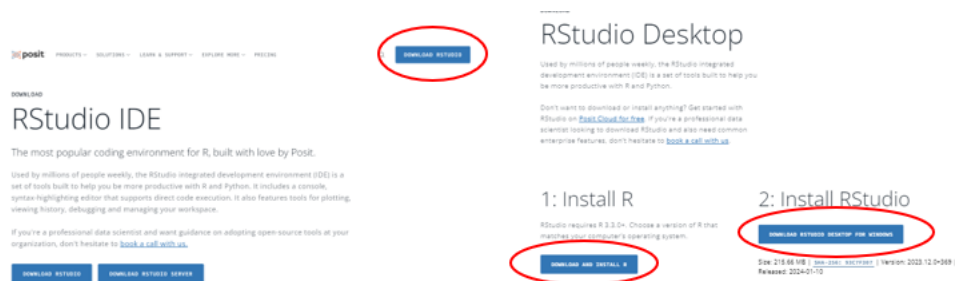
\*Note: This tutorial is meant to be a good entry point into R for completely new users, but everyone has different levels of general computer knowledge. If you are new to downloading programs or finding file paths, don't be afraid to ask a supervisor or peer for help. Those steps are some of the first to working with R but after setting up you don't need to interact with your computer's file system much.

## Part 1: Setting up RStudio

**Sample Data:** You can create a sample data file by copying and pasting the table on the next page into an excel document, naming the file "r\_soil\_moisture\_example," and making sure it is saved as an xlsx file (so, r\_soil\_moisture\_example.xlsx). Note that the excel file name and the column headers have underscores between words. This is common in R coding as strings of words need to be treated as single, differentiated entities when we tell R what to do with them. In R, you can directly import data sheets from web pages like SharePoint, but the script we will write imports data sheets from a folder on your computer.

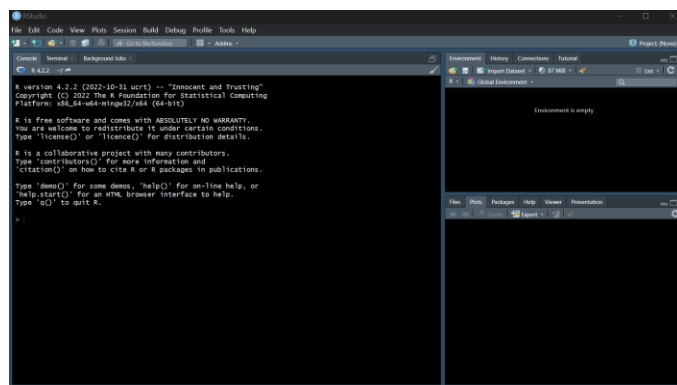
sample	rep	crop	tin_weight	tin_and_sample_weight_wet	tin_and_sample_weight_dry
1	1	corn	1.342	6.378	4.721
1	2	corn	1.567	6.566	4.902
2	1	corn	1.122	6.099	4.004
2	2	corn	1.201	6.311	4.388
3	1	corn	1.402	6.502	4.721
3	2	corn	1.333	6.335	4.56
4	1	miscanthus	1.098	6.124	3.11
4	2	miscanthus	1.175	6.201	3.256
5	1	miscanthus	1.299	6.304	3.498
5	2	miscanthus	1.326	6.402	3.561
6	1	miscanthus	1.211	6.322	3.274
6	2	miscanthus	1.169	6.083	3.087

**Downloading and Installing R & RStudio:** If you type ‘Download RStudio’ into a search engine, you will find a website to download R & RStudio (I used <https://posit.co/downloads/>). You will have to download the versions for your computer (Windows, OS, or Linux).

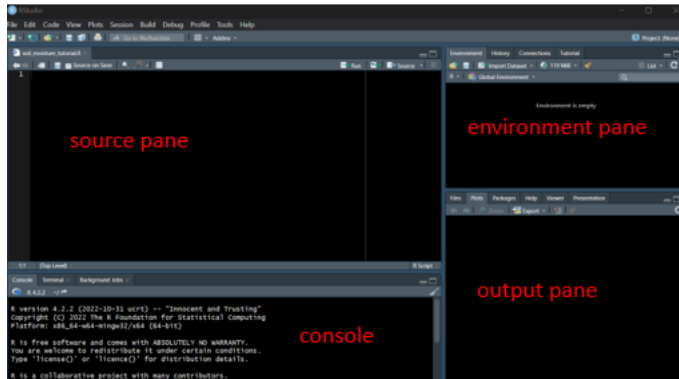


For more help downloading and installing R and RStudio:  
<https://rstudio-education.github.io/hopr/starting.html>

Here is what the RStudio interface will look like once you download, install, and open the program. Yours will likely be different colors.

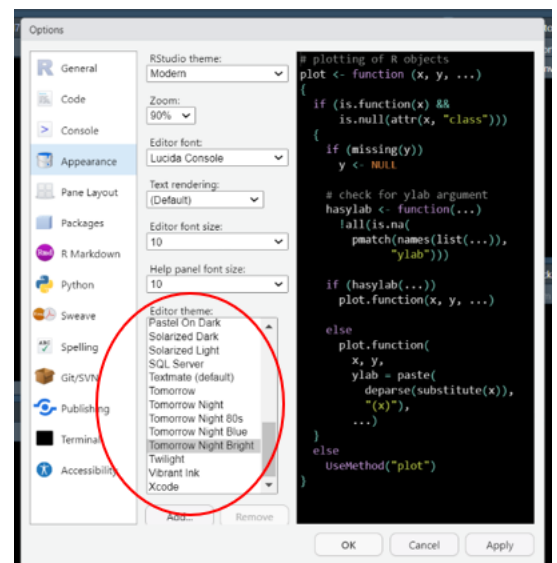
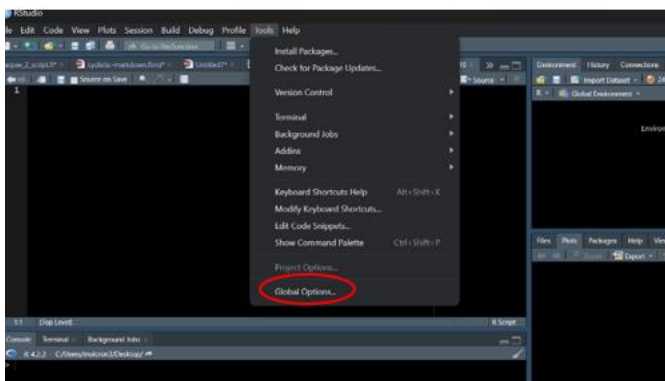


In the upper lefthand corner of the interface, select File > New File > R Script and the source pane should open. You can immediately save your file as something like 'soil\_moisture\_tutorial' by selecting File > Save As..., inputting your file name, and selecting 'Save'. R will save the file as an R file (with a .R extension). Now your RStudio interface should look like this (without the red letters):



The upper left is your source pane, the upper right is your environment pane, the bottom left is your console, and the bottom right is your output pane.

\*\*Quick detour, I'd suggest picking a color scheme that you like and is easy on your eyes (that's why my interface is black instead of white). I used the theme "Tomorrow Night Bright," and if you'd like your screen to look exactly like mine for this exercise, you can select Tools > Global Options > Appearance and then scroll in the 'Editor theme' section until you find it. You can also pick a different theme! Color schemes can make your RStudio space fit your style, visually differentiate code types in the way that works best for your brain, and make you appear cool and good at coding to people glancing at your computer screen.



## Writing a script:

### Part 2: Create a Script to Calculate Soil Moisture

Okay, now that R and RStudio are installed and styled, we can start writing a script in the RStudio source pane. There are 6 main steps to making a script: Import packages and libraries, clear R's brain, set working directory, import your data, perform calculations, and export the results. We'll perform all of these in part 1 of the script. (You're welcome to stop after part 1 of the tutorial; it will be an introduction to basic RStudio script creation on its own).

- 1) *Import Packages Libraries*: Which packages libraries you will want to import depends on what types of calculations, data wrangling, and other functions you are planning to perform. R comes with some functions that you can use without importing any packages or libraries, but most of the time you will need additional functionality. It doesn't hurt anything (usually) to import libraries you don't use, so when you're starting out, if you want to just import the same couple of libraries that let you do most of what you want, that's totally fine. Libraries do sometimes need to be imported in the correct order to prevent the wrong one winning in the case that their functionality is at odds. In that case, the library loaded last will override the functionality of the others. You likely don't need to worry about this for a while and can get a lot of scripting done with the libraries included in this example. Packages contain libraries so which packages and libraries you install will generally go together. Packages need to be imported before libraries.

For this script, we'll import **tidyverse**, **ggplot2**, **readxl**, and **writexl**, which are in the packages **tidyverse**, **readxl**, and **writexl** (the ggplot2 library is within the tidyverse package).

Code to copy and paste into your R source pane (top left):

```
#--Import packages

install.packages("tidyverse")
install.packages("readxl")
install.packages("writexl")

library(tidyverse)
library(ggplot2)
library(readxl)
library(writexl)
```

It should look something like this (you will have different colors if you picked a different theme and that's fine!)

```
##--Enter packages

install.packages("tidyverse")
install.packages("readxl")
install.packages("writexl")

library(tidyverse)
library(ggplot2)
library(readxl)
library(writexl)
```

To run the script, highlight the chunk of code you'd like to run and hold down 'ctrl' and 'enter' on your keyboard at the same time. You can also click the 'Run' button above the source pane in place of using ctrl + enter. You might get some error messages and you can likely ignore them. Your console (bottom left pane) should look something like this:

```
> ##--Import packages
> library(tidyverse)
> library(ggplot2)
> library(readxl)
> library(writexl)
Warning message:
package 'writexl' was built under R version 4.2.3
>
```

Once a script is built, you will often run it all at once, but since we're building a script and checking it as we go, we will run ours in pieces first.

Spacing in R does not affect the way the script runs and is used as a visual organization tool. There are best practices that I follow imperfectly in this tutorial. Generally, you want a new line for each task and when a task takes up multiple lines, tab or space to the right (there will be examples in next steps). Parentheses use does affect the way R runs, and there must always be an equal number of parentheses facing each way when you run code (the parentheses must be closed). When a code chunk won't run, unclosed parentheses are often the culprit.

To install the packages, we used the function **install.package()** with the specific package in the parenthesis and quotations. To import the libraries, we used the function **library()** with the specific library in parenthesis. A **function** with more specific information in parentheses is common in R syntax. The specific information is called an **argument**. People sometimes use the terms 'command' and 'function' interchangeably when talking about R. I'm going to always use function for consistency, though if you are talking about code with someone who works with R a lot, they might use command at times.

We also used the # symbol for our first line, which tells R we are writing a comment, not code. In my color scheme this grays the comments out in the source pane (but not

the counsel) to make it clear that they are not part of the script. Including comments is super important to keep track of what is happening in your scripts.

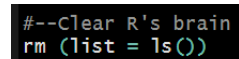
## 2) Clear R's Brain

Okay libraries are loaded. This next step is easy! You're just clearing out R's brain before you start your script so that nothing is left over from the last time you ran this or some other script. You can use this same line of code for every script you write.

Code to copy and paste into your source pane:

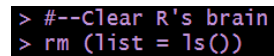
```
#--Clear R's brain  
rm(list = ls())
```

It should look something like this:



```
#--Clear R's brain  
rm(list = ls())
```

Run the script (by highlighting the code you just pasted and either hitting ctrl and enter at the same time on your keyboard or hitting the run button above the source pane. Your console (bottom left) should look something like this:



```
> #--Clear R's brain  
> rm(list = ls())
```

The **rm()** function removes objects from our R environment and **list = ls()** in the parentheses tells R we want the list of objects removed to consist of all objects. Leaving the parentheses of a command blank generally calls default settings. In this case the default selects everything because you didn't specify something specific. This is conceptually complicated, and you don't need to understand why this line of code works when you're beginning to use R (or maybe ever depending on the kind of scripts you want to write)! Just copy and paste it at the beginning of your scripts.

## 3) Set Working Directory

This step is a little more involved because you are telling R what path to follow to retrieve **your** data for the script. We will use the **setwd()** function to tell R what folder to pull files from. What I tell you to copy and paste needs to be customized to where on your computer the excel file is. If you know the path to your folder, awesome, put it in quotation marks in the parentheses of the **setwd()** command. If you do not know the path to the folder where your soil moisture data is saved, open the folder and right click on the file. One of the options in the window that pops up should be 'copy file

as path.' Select this and then paste the path into your source pane or another document to edit. For me, this looks like:

```
"C:\Users\mulcron3\Desktop\r_soil_moisture_example.xlsx "
```

Which I change to `"/Users/mulcron3/Desktop/"` (I deleted the 'C:', flipped the direction of the slashes, and deleted my file name).

Code to copy and paste into your source pane:

```
--Set working directory  
setwd ("/Users/mulcron3/Desktop/")
```

But, with your path copy and pasted over mine.

Your source pane should look something like this (but with your folder path):

```
--Set working directory  
setwd ("/Users/mulcron3/Desktop/")
```

And your console should look something like this:

```
> --Set working directory  
> setwd ("/Users/mulcron3/Desktop/")
```

I'm going to stop including what the console (bottom left pane) should look like. For the most part, it will simply reprint what was run in the source pane. The console includes notes about issues that are encountered, so it becomes very useful when you run into problems. In future scripts, you may also ask R to display information without saving it as an object, in which case it will display in the console.

#### 4) Import Data

Since your working directory is set to the folder location of your file, you can simply use the **readxl()** function to import your data. Because we want to manipulate the data table, we will assign it as an object. An **object** is anything we want to store in R so we can access it. This can be numbers, lists, tables, and more. To assign data as an object, use an arrow made of '`<`' next to a '-' with the arrow pointing towards the object name (as if the data and on the other side of the arrow is being directed into the object name). I name tables and data frames with periods between words, and column and file names with underscores, but some people use all periods or all underscores.

The `<-` (arrow) we used in this step isn't a function; it's called an operator. **Operators** in R tell the script what to do with objects, like functions do, but aren't followed by

parenthesis. Here, the arrow is telling R to store our data as an object called data.sm.all. The names can be confusing because function, command, and operator all have meanings outside of R language that overlap. For now, you can just remember that functions have parentheses and operators don't, but both are telling R what to do with objects. It's also okay not to pay too much attention to terms right now.

Code to copy and paste into your source pane:

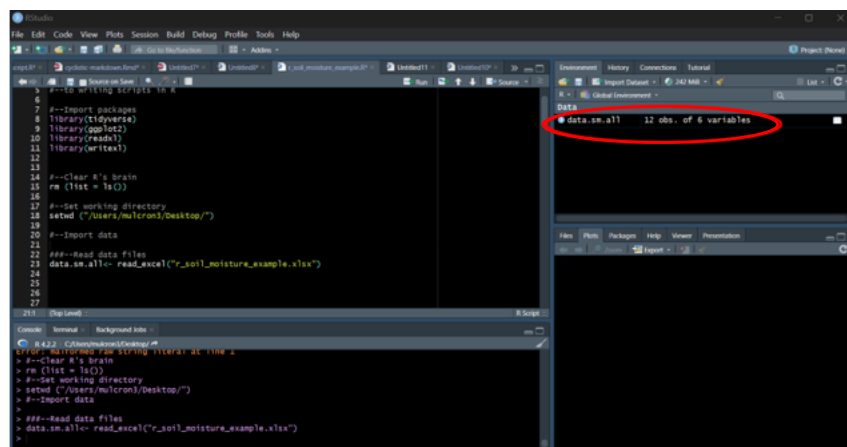
```
###--Read data files
data.sm.all<- read_excel("r_soil_moisture_example.xlsx")
```

It should look something like this:

A line called data.sm.all should appear in your environment (upper right) pane.

```
###--Read data files
data.sm.all<- read_excel("r_soil_moisture_example.xlsx")
```

A line called data.sm.all should appear in your environment (upper right) pane.



If you click on the data.sm.all in the environment pane, a new tab should open in your source pane where you can view the data. You should see all the columns from your excel sheet displayed in this tab.

## 5) Perform Calculations

That's it for set up! Your libraries are imported, and your data is stored as an object. Now we can perform calculations on the data. There are three steps we need to perform to find the soil moisture for each rep:



- i. Subtract the mass of the dried sample and tin from the mass of the wet soil and tin to find **the mass of water lost through oven drying**.
- ii. Subtract the tin mass from the mass of the wet soil and tin to get **the mass of the wet soil**.
- iii. Divide the mass of the water lost through drying by the mass of the wet soil to determine what **percentage of the wet soil weight was moisture**.

a) i. Find the **mass of water lost through drying**.

For this calculation, we will use three operators: the arrow, which you already used, the **dollar sign (\$)**, and **subtraction (-)**. The \$ operator is a very common operators. The \$ operator goes between a data table name and the existing column in the data table you want to select, calling that column, or column name you would like to create and add to the table. We do both in this step; 'water\_mass' is a new column we are telling R to create by putting it on the other side of the \$ from the data.sm.all table on the object side of the arrow operator, and we are calling the already existing 'tin\_and\_sample\_weight\_dry' and 'tin\_and\_sample\_weight\_wet' on the instructions side of the arrow operator to subtract one from the other using the subtraction (-) operator. The new column created on the object side of the operator arrow will be populated with the calculated values from the instruction side. You can use the subtraction operator in R to subtract one number from another or to subtract sequences of values like columns (what we did in this step), vectors, or matrices. Same goes for other operators like addition, multiplication, division, etc.

Code to copy and paste into your source pane:

```
#--Subtract dried soil sample & tin from wet soil sample & tin to find the
#--mass of water lost through drying, and create a column called 'water_mass'
#--to record the answer in.
```

```
data.sm.all$water_mass <- data.sm.all$tin_and_sample_weight_wet -
  data.sm.all$tin_and_sample_weight_dry
```

```
#--Subtract dried soil sample & tin from wet soil sample & tin to find the
#--mass of water lost through drying, and create a column called 'water_mass'
#--to record the answer in.

data.sm.all$water_mass <- data.sm.all$tin_and_sample_weight_wet -
  data.sm.all$tin_and_sample_weight_dry
```

The data.sm.all table in your environment (upper right) pane should now read 12 observations of 7 variables instead of 6 variables because we added water\_mass as a variable. If you click on the data.sm.all object, a new tab will open in your source

pane or the one you have open for the table will refresh. There should be a populated column called `water_mass` on the right-hand side (you may need to scroll to see the last column).

*(If you were performing this step in Excel, you would name the G column 'water\_mass' and write '= (E1-F1)' in the first cell of the empty G column and drag it down through all the data).*

**b) ii. Find the mass of wet soil.**

We don't need any new operators or commands for this step. We are doing the same thing as the last step, just with different columns. This time, the 'water\_mass' column we created in the last step is called on the instructions side of the arrow operator along with `tin_weight`, and a new column named 'wet\_soil\_mass' is created on the object side of the arrow operator and populated with the values calculated.

Code to copy and paste into your source pane:

```
--Subtract tin from wet soil sample & tin to get mass of wet soil sample, and  
--create a column called 'wet_soil_mass' to record the answer in.
```

```
data.sm.all$wet_soil_mass <- data.sm.all$water_mass - data.sm.all$tin_weight
```

```
--Subtract tin from wet soil sample & tin to get mass of wet soil sample, and  
--create a column called 'wet_soil_mass' to record the answer in.  
data.sm.all$wet_soil_mass <- data.sm.all$water_mass - data.sm.all$tin_weight
```

The `data.sm.all` table in your environment pane should now have 8 variables.

*(If you were performing this step in Excel, you would name the H column 'wet\_soil\_mass' and write '= (G1-D1)' in the first cell of the empty H column and drag it down through all the data).*

**c) iii. Find the percentage of wet soil weight that was moisture.**

The only new operator we use in this step is **division (/)**, which functions very similarly to subtraction (-).

Code to copy and paste into your source pane:

```
--Divide the mass of water lost through drying by the mass of the wet soil  
--sample to get the soil moisture percent for the sample, and create a column
```

#--called 'soil moisture' to record the answer in.

```
data.sm.all$soil_moisture <- data.sm.all$water_mass / data.sm.all$wet_soil_mass
```

```
#--Divide the mass of water lost through drying by the mass of the wet soil  
#--sample to get the soil moisture percent for the sample, and create a column  
#--called 'soil_moisture' to record the answer in.  
  
data.sm.all$soil_moisture <- data.sm.all$water_mass / data.sm.all$wet_soil_mass
```

The data.sm.all table in your environment pane should now have 9 variables.

*(If you were performing this step in Excel, you would name the I column 'soil\_moisture' and write '=(G1/ H1)' in the first cell of the empty I column and drag it down through all the data).*

## 6) Export Results

That's all the calculations for Part 2! You used an R script to find the percentage of wet soil that was moisture (aka the soil moisture) of each of your samples. Now we'll export our data to wrap up the section.

To do this, we'll use the function **write.csv()**. R will automatically export the data to the folder that is set as our working directory, but we need to tell R which data to export. First, we put the name of the table we want to export (data.sm.all) in the parentheses, then a comma followed by file = "name\_of\_file". A function's argument often consists of multiple pieces of information. Order within arguments matters as the command assigns information type to sequence, though you can usually override it. Don't worry about this right now, just know that multiple arguments are often included in the parenthesis of a command.

The **equals (=)** operator was 'new' in this step. We also used it when clearing R's brain in step 2 with rm (list = ls()), but it wasn't explained. Differences between the equals (=) operator and the arrow (<-) are a little complex, but generally the arrow is used when assigning instructions to objects and the equals is used within an argument (in the parenthesis of a command). Within an argument, you can often leave out the equals if you order your arguments correctly for the command. The 'file =' portion of this line of code could have been left out because R expects the name I want to assign exported file to be the second piece of information in the argument. I included the 'file =' because it helps me keep track of what my code is doing. Again, don't worry much about this now, I'm just briefly explaining so you don't get hung up on it.

Code to copy and paste into your source pane:

```
#--Export data
```

```
write.csv(data.sm.all, file = "r_sm_preliminary_example_results.csv")
```

It should look something like this:

```
#--Export data  
write.csv(data.sm.all, file = "r_sm_preliminary_example_results.csv")
```

Note: You can export as an excel document if you'd prefer by replacing the code in this step with:

```
write_xlsx(data.sm.all, "r_sm_preliminary_example_results.csv")
```

That's the end of Part 2! If you're interested, we'll do a short quality check of our data in Part 3. Keep in mind that it is totally fine if you don't fully understand the commands we used in this section and don't feel like you could write much code from scratch yet. I wrote plenty of functional scripts by Googling what I was trying to do, copying and pasting other people's code then using trial and error to get it to work for me. Just be careful to check often that code is doing what you want it to if you don't fully understand the functions you're using. You'll learn as you go.

### Part 3: Create a Script for Data Quality Checking and Processing (optional)

In Part 2, we used common R functions and operators to find the soil moisture for each of our replicates. In section two, we will check the coefficients of variation (COVs) between each sample's two replicates and have R create a column flagging issues based on the COVs. If the COV between the two reps of a sample is too high, it indicates that there was a problem with at least one of the reps and the data might be incorrect. We will also find the mean soil moisture for each sample by averaging replicates.

#### 1) Calculation: Find the Coefficient of Variation between reps

Because we are continuing with the same script, we do not need to import libraries, clear R's brain, or set our working directory. We can jump into calculations right below the code we used to export our data from Part 1.

We'll use a few new operators and a few new functions to calculate the COV between our reps. To find COV, we take the standard deviation of the soil moisture of the two reps, divide that number by the means of the soil moisture of the two reps, and multiply that number by 100. We will use two new operators to do this, **multiplication (\*)** which functions very similarly to subtraction and division, and the

**pip**ing (%>%) operator. The piping operator is very useful and feeds the results of one operation (set of instructions) into another. The piping goes in the direction of the arrowhead between the percentage signs (left to right). In this step, we create a new table on the object side of the arrow called data.sm.covs, then on the instructions side we start with our table data.sm.all, then apply the **group\_by()** function to that table to group reps by sample, then take that table that is grouped by sample and use the **summarize()** function to create a new column called 'cov' populated with the results from using the **sd()** and **mean()** functions to calculate COV.

The new commands are group\_by(), summarize(), sd(), and mean(). The sd() and mean() commands are straightforward and calculate the standard deviation or mean for the information in parenthesis. In this case, for each set of sample replicates because the piping operator fed the grouped table to the operation. The group\_by() function groups data by the variable in the parentheses, in this case 'sample'. Summarize() is a handy command that takes data input and creates and populates a new column based on the operations in the summarize() argument. We've already piped the grouped data table into the summarize() command, so we don't need to specify which data to use in the argument. We just name the new column (cov) and use an equals (=) to instruct what calculations to perform for the column values.

Code to copy and paste into your source pane:

```
#--Find the coefficient of variation between reps
```

```
data.sm.covs <- data.sm.all %>% group_by(sample) %>%  
  summarize(cov = sd(soil_moisture) / mean(soil_moisture) * 100)
```

It should look something like this:

```
#--Find the coefficient of variation between reps  
  
data.sm.covs <- data.sm.all %>% group_by(sample) %>%  
  summarize(cov = sd(soil_moisture) / mean(soil_moisture) * 100)
```

A new data.sm.covs table should appear in your environment pane. It will only have 2 variables, which is fine.

Note: The reason we didn't use piping in part 2 to calculate soil moisture was that we were creating new columns and then using those columns in the calculation to create the next column. We could have used piping in part 2 though. The following code would have gotten us to the same end values:

```
data.sm.all <- data.sm.all %>%  
  mutate(water_mass = tin_and_sample_weight_wet - tin_and_sample_weight_dry) %>%
```

```
mutate(wet_soil_mass = water_mass - tin_weight) %>%
  mutate(soil_moisture = water_mass / wet_soil_mass)
```

**Mutate()** functions like `summarize()` in that it creates a new column to populate with the results of a calculation but `mutate()` is used without grouping for calculations (when we used `summarize()` in this step we grouped by sample to find the mean of reps, but in part 2 we treated reps individually). Piping in part 2 would have let us use one chunk of code, but it isn't using much less code overall and makes it harder to see what's going on. Piping *is* better for this section 3 step because the `group_by()` data wrangling step isn't creating new columns but is set up for the `summarize()` step. We could have done this step in section 3 without piping with code like:

```
data.sm.cov.groups <- group_by(data.sm.all, sample)

data.sm.covs <- summarize(data.sm.cov.groups, cov = sd(soil_moisture) / mean(soil_moisture) * 100)
```

Without piping in this step, we create an intermediate table that makes it less clear what steps we are performing and takes more code. It's alright to not worry much about when to use piping and when not to right now; you'll develop a feel for it.

## 2) Data Wrangling: Join the Coefficient of Variation values to your existing data table.

Using the `group_by()` and `summarize()` functions reduced the table we created to only the `group_by` variable, `sample`, and the column generated in the `summarize` argument, `cov`. We'll add the `cov` column from the `data.sm.covs` table we created in the last step to our full table with the rest of our data columns using the **`join()`** function. In this case specifically **`left_join()`**. The 'left' tells R to join the second table to the table we list first, which means only data from the second table that matches the 'key' variable in the first table (which we put in the argument; `data.sm.covs` column in the example below) will be kept. Joining and merging tables can be a little tricky, but between Googling and some trial and error you'll be able to figure it out in future scripts. We are replacing our original complete table with the same table with the `cov` column joined, which is why the `data.sm.all` object appears twice in the code for this step.

Code to copy and paste into your source pane:

```
#--Join the cov values to the table

data.sm.all <- data.sm.all %>%
  left_join(data.sm.covs)
```

It should look something like this:

```
#--Join the cov values to the table  
data.sm.all <-data.sm.all %>%  
  left_join(data.sm.covs)
```

The data.sm.covs table in your environment pane should be updated now. If you open the table, you should see a complete data frame with 10 columns.

Note: We could have added the across() function argument to group\_by() and summarize() in step 1 to keep all columns and avoid needing the join() command and step 2 altogether, but I wanted to show how because joins are super useful, and when first learning R you'll probably find yourself doing things in more steps rather than structuring the arguments of your commands perfectly efficiently, and that's okay.

### 3) Data Wrangling: Add a “needs redo” column to your existing data table.

Now we need to add a column to our table that we will later populate to display whether the COV indicates a rerun. To do the, we'll use the \$ operator to create a new column called 'needs\_redo', and the c(), rep(), and nrow() functions (embedded within each other) to leave the column empty. Concatenate, or c() combines multiple values into a vector or a list. The rep() command replicates elements of vectors or lists a specific number of times and nrow() returns the number of rows in a data frame. The first part of the rep() argument is saying to input 0 and the second part of the rep() argument says to do that for as many rows as there are in data.sm.all.

Code to copy and paste into your source pane:

```
#--Add an empty column called "needs_redo"
```

```
data.sm.all$needs_redo = c(rep(0, nrow(data.sm.all)))
```

It should look something like this:

```
#--Add an empty column called "needs_redo"  
data.sm.all$needs_redo = c(rep(0, nrow(data.sm.all)))
```

If you click on the data.sm.all table in your environment pane, there should now be an empty column headed 'needs\_redo' on the right side of the table.

### 4) Calculation: Add a Y or N to the “needs redo” column

We will now have R tell us whether each sample's COV indicates a rerun or not. We will use a COV of 15 as the cutoff for indicating a rerun. We will tell R to put a Y in the column if the COV is greater than 15, and N if the COV was between 0 and 15.

There are a few different ways we could do this. The easiest is probably using a conditional function. The function **ifelse()** lets us set a condition and tell R what to do depending on the condition in the argument. Within the argument we also use **between()** to set the variable we are using (cov) and the two numbers starting and ending the range (0 and 15). The between function is the first argument for our ifelse() function, followed by what we want R to return if the value does fall within the range ("N") and what we want R to return if the value does not fall within the range ("Y"). We also used the function mutate() to modify our empty column. Before, we used mutate() to create and fill a column, but it can also modify existing columns.

Code to copy and paste into your source pane:

```
#--Add N for reps that don't need to be redone (cov <=15) and Y for samples that  
#--do need to be redone (cov >15)
```

```
data.sm.all <- data.sm.all %>%  
  mutate(needs_redo = ifelse(between(cov, 0, 15), "N", "Y"))
```

It should look something like this:

```
#--Add N for reps that don't need to be redone (cov <=15) and Y for samples that  
#--do need to be redone (cov >15)  
  
data.sm.all <- data.sm.all %>%  
  mutate(needs_redo = ifelse(between(cov, 0, 15), "N", "Y"))
```

If you click on data.sm.all, the needs\_redo column should now be populated with Y and N instead of zeros. You may need to scroll over to see the last column. Sample 1 replicates should be Y and all other sample replicates should be N. This makes sense as the COV for sample 1 replicates was about 75, indicating an issue with the data.

Note: Steps 3 and 4 could have been combined via piping via the code below:

```
data.sm.all <- data.sm.all %>%  
  mutate(needs_redo = ifelse(between(cov, 0, 15), "N", "Y"))
```

This takes up less space, but the way we did it works fine!

Another way to accomplish step 4 (this step) would have been a for loop. The for loop code in the screenshot below is commented out, meaning the script won't run it.



```

#--note: Addin 'N' and 'Y' based on cov could also be accomplished using
#--for loops. The # in front of the code block tells R that it's a comment,
#--not code, and therefore won't run it. This can be useful if you want to try
#--running a script without a part of the code but also want to easily return
#--the code to the script. Try deleting the '#' and you'll see the code block
#--become active (put the '#' back when you're done though!) I wanted to include
#--this both as an example of commenting out code with the '#' and how there are
#--often multiple ways of doing the same in the R.

#for (i in 1:nrow(data.sm.all)) {
#  if (data.sm.all$cov[i] <= 15)
#    {data.sm.all$needs_redo[i] = "N"}
#  if (data.sm.all$cov[i] > 15)
#    {data.sm.all$needs_redo[i] = "Y"}
#}

```

If you remove the #s in the for loop code, the source pane looks like this:

```

#--note: Addin 'N' and 'Y' based on cov could also be accomplished using
#--for loops. The # in front of the code block tells R that it's a comment,
#--not code, and therefore won't run it. This can be useful if you want to try
#--running a script without a part of the code but also want to easily return
#--the code to the script. Try deleting the '#' and you'll see the code block
#--become active (put the '#' back when you're done though!) I wanted to include
#--this both as an example of commenting out code with the '#' and how there are
#--often multiple ways of doing the same in the R.

for (i in 1:nrow(data.sm.all)) {
  if (data.sm.all$cov[i] <= 15)
    {data.sm.all$needs_redo[i] = "N"}
  if (data.sm.all$cov[i] > 15)
    {data.sm.all$needs_redo[i] = "Y"}
}

```

Skip the rest of this step if you aren't interested in learning about for loops right now, but I wanted to include a quick explanation for anyone who is interested. The basic syntax for using the **for()** command to create a for loop is:

```

for (variable in sequence) {
  expression
}

```

In our for loop, 'i' represents our variable. This could be a different letter like 'x,' but 'i' is common. Instead of a letter, we could even use a phrase like 'sample.' Whatever helps you keep your code strait. We are telling R that for each variable (sample rep here) from row 1 through all rows in data.sm.all (remember that the command nrow() selects for the number of rows of the argument, data.sm.all here), we want to carry out the expression after the curly bracket. We are using the **if()** command to tell R that if the cov for the sample rep is less than or equal to 15, then the needs\_redo column should be populated with an 'N,' and if the cov for the sample rep is greater than 15,

the `needs_redo` column should be populated with a 'Y'. We used **the less than or equals to** (`<=`) operator and the **greater than** (`>`) operator to do this.

Even though the for loop is longer than the original `ifelse()` code chunk we used for this step, for loops can sometimes be easier to understand and more elegantly keep track of complex trains of thought than `ifelse()` statements. Both are great to use when you're starting out creating data quality check scripts, and you'll likely learn more about when and how to use for loops if you start writing more complex scripts.

If you did copy paste the for loop into your script, either delete it or comment it out so it isn't redundant with the `ifelse()` code chunk!

## 5) Calculation: Average reps for each sample

For the last step in this section, we'll calculate our final soil moisture values by averaging the two reps for each sample. We'll use operators and functions we're already familiar with to do this.

Code to copy and paste into your source pane:

```
#--Average the two reps, and create a column called 'average_soil_moisture'

data.sm.final <- data.sm.all %>%
  group_by(sample, crop, needs_redo) %>%
  summarise(average_soil_moisture = mean(soil_moisture))
```

It should look something like this:

```
#--Average the two reps, and create a column called 'average_soil_moisture'
data.sm.final <- data.sm.all %>%
  group_by(sample, crop, needs_redo) %>%
  summarise(average_soil_moisture = mean(soil_moisture))
```

After running the code, click on your `sm.data.final` table in the environment pane. It should look like this:

	sample	crop	needs_redo	average_soil_moisture
1	1	corn	Y	11.207478
2	2	corn	N	2.408285
3	3	corn	N	4.357523
4	4	miscanthus	N	1.618455
5	5	miscanthus	N	1.868612
6	6	miscanthus	N	1.649537

We have a soil moisture value for each sample and whether that sample needs a redo. We'll export both this simplified table and one with all our data in the next step.

## 6) Export Data

This step is the same as in part 2.

Code to copy and paste into your source pane:

```
#--Export data
```

```
write.csv(data.sm.final, file = "r_sm_example_results.csv")
write.csv(data.sm.all, file = "r_sm__example_results_all_info.csv")
```

This is the end of the data quality checking part of the script! You've done all the calculations and data wrangling needed for this tutorial. There is one more (optional) section that uses a few graphing code chunks to give an idea of how quickly you can make graphs in R to get an idea of what's going on with your data.

## Part 4: Visualizations (optional)

In this section we will make a quick bar graph to get a sense of whether soil moisture differences are driven by crop type. We will also make a scatterplot by changing a few words in the bar graph code as an example of how much visualization power you already have by understanding R's (ggplot2; there are other visualization libraries to choose from but ggplot2 common) code structure. Once again, we can continue with the same script and don't need to import libraries, clear R's brain or set our working directory.

### 1) Bar Graph

The first graph we will make is a bar graph. The code is made up of functions and operators telling R what to do with objects (our data) just like in the previous

sections. I'm not going to go over what all the commands used mean for this part, but rather provide some examples of base code and tweaks to get you started with graphing. Between Googling and trial and error you can likely figure out how to make all kinds of useful graphs quickly.

There is a new structure used with ggplot where we are adding commands using a + operator rather than as arguments within a command (though the commands in added with the + operator still often have arguments).

Code to copy and paste into your source pane:

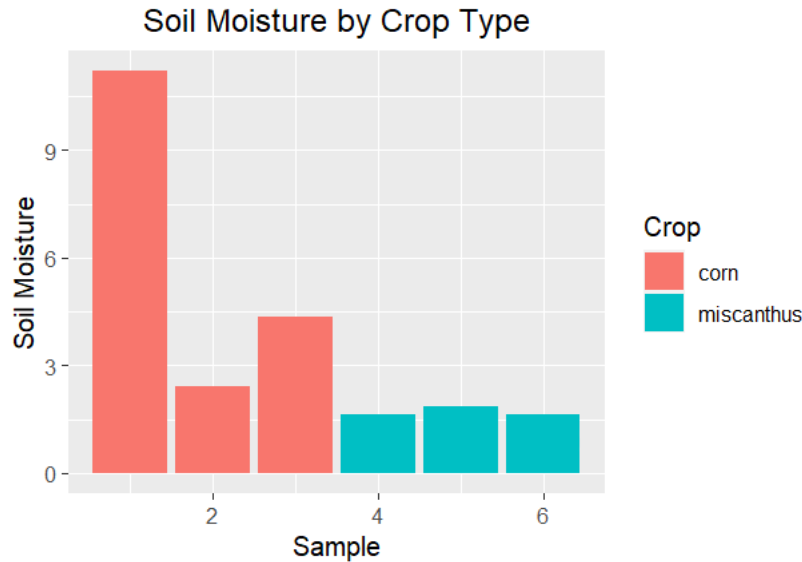
```
#--Create a bar graph to see if soil moisture differs by crop type (We use geom_col
#--instead of geom_bar because geom_bar will automatically try to aggregate data
#--and we just want straight y values for x values right now)

ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, fill = crop)) + geom_col() +
  ggtitle ("Soil Moisture by Crop Type") +
  theme(plot.title
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = 'Sample', fill ='Crop')
```

```
#--Create a bar graph to see if soil moisture differs by crop type (We use geom_col
#--instead of geom_bar because geom_bar will automatically try to aggregate data and
#--we just want straight y values for x values right now)

ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, fill = crop)) + geom_col() +
  ggtitle ("Soil Moisture by Crop Type") +
  theme(plot.title
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = 'Sample', fill ='Crop')
```

And a graph should appear in your navigation (bottom right) pane that looks like this:



Looks like soil moisture tended to be higher for corn than miscanthus!

Note: we use `geom_col()` instead of `geom_bar()`, which stands for column. The difference has to do with the way each handles data.

## 2) Scatter Plot

To create a scatterplot, we change the `geom_col()` function to `geom_point()` and the 'fill=' argument to 'col='. The 'col' in the argument stands for color (I think), not column.

Code to copy and paste into your source pane:

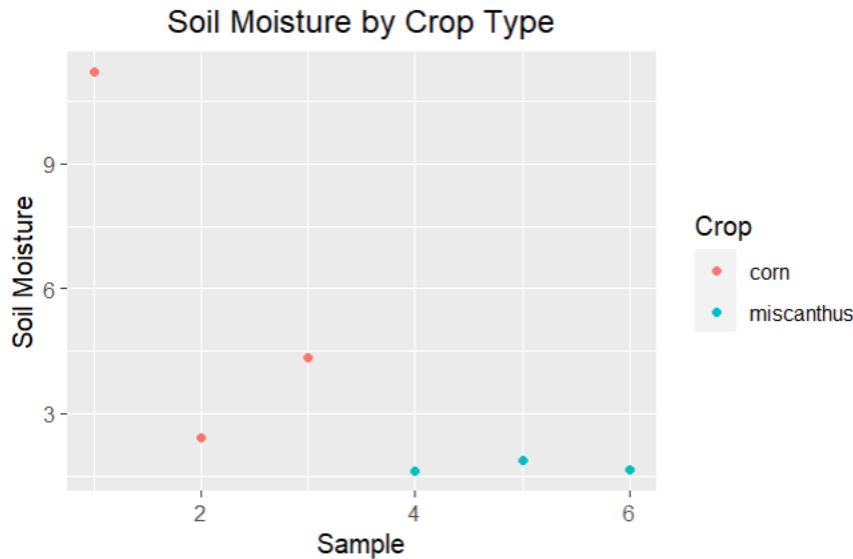
`--We can also plot a scatterplot by changing 'geom_col' to 'geom_point', and  
 --also 'fill' to 'col.' Other than those changes, the code is the same.`

```
ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, col = crop)) + geom_point() +
  ggtitle("Soil Moisture by Crop Type") +
  theme(plot.title
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = "Sample", col = "Crop")
```

```
--We can also plot a scatterplot by changing 'geom_col' to 'geom_point', and
--also 'fill' to 'col.' Other than those changes, the code is the same.

ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, col = crop)) + geom_point() +
  ggtitle("Soil Moisture by Crop Type") +
  theme(plot.title
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = "Sample", col = "Crop")
```

And a graph should appear in your navigation pane that looks like this:



### 3) Bar Graph Color Change

To change the colors of the bar graph we created in step 1, we'll add another function (via a + operator and a new line of code to keep things neat). The function `scale_fill_manual()`, along with its arguments, will allow us to change our colors. R does a fun thing where it will preview the color for you in the code so long as you type in a valid color name.

Code to copy and paste into your source pane:

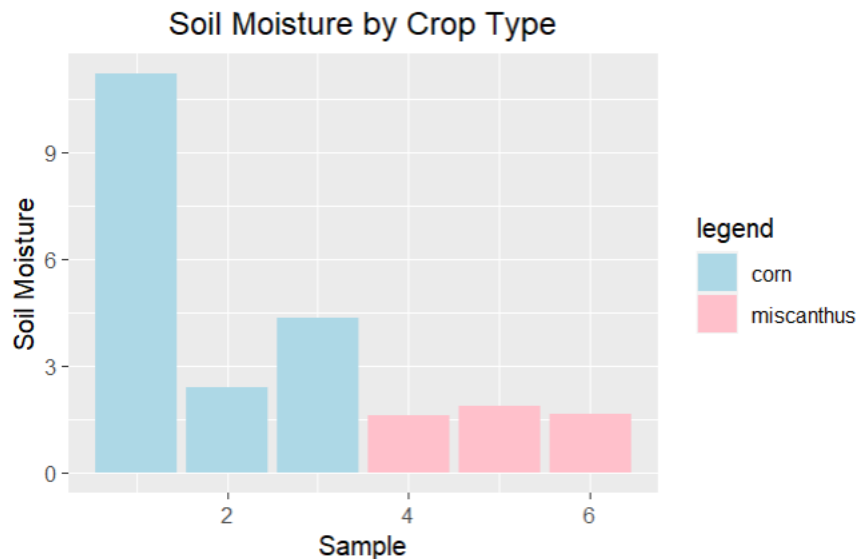
`--To change colors of the bar graph, we add a line for scale_fill_manual`

```
ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, fill = crop)) + geom_col() +  
  ggtitle("Soil Moisture by Crop Type") +  
  scale_fill_manual("legend", values = c("corn" = "lightblue", "miscanthus" = "pink")) +  
  theme(plot.title  
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = 'Sample', fill = 'Crop')
```

It should look something like this:

```
--To change colors of the bar graph, we add a line for scale_fill_manual  
ggplot(data.sm.final, aes(x = sample, y = average_soil_moisture, fill = crop)) + geom_col() +  
  ggtitle("Soil Moisture by Crop Type") +  
  scale_fill_manual("legend", values = c("corn" = "lightblue", "miscanthus" = "pink")) +  
  theme(plot.title  
    = element_text(hjust = 0.5)) + labs(y = "Soil Moisture", x = 'Sample', fill = 'Crop')
```

And a graph should appear in your navigation pane that looks like this:



You can do a lot of graphing with the ggplot command. Many data processing scripts might involve combining graphing with conditions to data quality check and correct, which isn't as hard as it might sound. Feel free to use these graphing chunks as a base in future scripts. If these code chunks are overwhelming to you, I'd Google ggplot basics and it might be easier to wrap your head around single lines of code that still produce useful graphs.

That's the end of the Introduction to R Script Tutorial! Now that we've finished, you should be able to run your code all in one go by highlighting the entire thing and hitting the ctrl and enter keys or run button. You've already run the entire script in pieces as we went, but when you finish a script, you should check that it works as a whole. All the data should be exported to the folder you set as your working directory and all three graphs should appear in your output pane (click the arrow buttons in the top left of the output pane to navigate between graphs). If it doesn't work to run the whole script at once this time, feel free to trouble shoot but also don't sweat it, there's likely a mistake somewhere but you still accomplished the point of the tutorial, which was to get enough of an introduction to R to attempt other simple scripts yourself.

### Conclusion and Quick R tips:

In this soil moisture example, R wasn't much faster than performing calculations in Excel. For more involved calculation sets though, R can be incredibly speedy relative to Excel formula columns, and, importantly, can greatly reduce the likelihood of mistakes and create scalable processing pipelines. Many scripts are simply a longer set of calculations just like what you did here (meaning knowing how to do what you did in this exercise is already super useful, and you are closer than you think to creating useful research scripts). To make a script, you just need to import libraries, clear R's brain, import your data, run your calculations, and export your results. Even if a script is long, the calculations aren't always complicated.

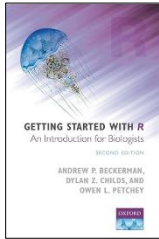
It's useful to remember when creating scripts that there are often multiple ways to perform the same command and those options can be equally useful. Sometimes, it's just a matter of what you came across first or of taste. Other times the choice between functions is about efficiency, but that becomes more important with scripts or data sets that are computationally expensive. Often, operations like piping and loops make scripts cleaner and more efficient, but when you start making scripts there's nothing wrong with doing it in more steps than strictly necessary. I learned most of what I know about R coding from Googling things like 'how do I rename a column in a spreadsheet in R' and trying out the options that came up. There is a learning curve, but you can make a lot of progress this way if you're interested in script creation. Some people try to learn R by reading about how the language works first, but that didn't work well for me. It was better to have a task I wanted to perform, search for how to do it, and learn by example. Focusing on understanding the language rules was more interesting to me after I already had a working knowledge of how to use R to do useful things.

I really suggest trying to work with R to perform some tasks, even simple ones you could do in excel, as it's a great skill to have on a resume and opens a lot of career options (at least that's true when I'm writing this in 2024). Using scripts to quality check data is a good entry point to having technicians and undergrads move from bench and field work to being involved in the data processing part of projects. R also is really very fun, at least to me, once you get the hang of it.



## Resources, R reminders, and glossary:

### Resources:



The book *Getting Started with R, An Introduction for Biologists* by Beckerman, Childs, and Petchey is a great resource. It's short and when I was first learning R I went through it in a weekend. The tone is accessible, the examples are often biology based, and I still reference it when I can't find an answer to an R question quickly with an internet search.

There are loads of good guides though and most of what comes up in an internet search for R basics will tell you useful things. Here is a list of some starter tutorials that I like:

- [Welcome | The tidyverse style guide](#)
- [Best Practices for R Programming. Writing clean code is a great skill to... | by Ivo Bernardo | Towards Data Science](#)
- <https://www.datacamp.com/tutorial/r-studio-tutorial>

### R Reminders:

- There are often many ways to perform the same task.
- Spacing is for organization but doesn't affect how a script runs.
- Googling examples is one of the best ways to learn.
- You don't need to understand everything about the commands you are using at first or be able to write code from scratch. It is okay to use other people's code examples to get the script running and learn as you go. Just keep checking that your code is working the way you intended after using functions you don't fully understand yet.
- Commenting on your code is **very important**. It will save you a lot of headaches later and let others use and appreciate the scripts you create.

### Glossary of commands we used:

Function	Use
<code>library()</code>	Installs libraries
<code>rm()</code>	Removes elements
<code>ls()</code>	Creates a list
<code>setwd()</code>	Sets working directory
<code>read_excel()</code>	Reads an excel file
<code>write.csv()</code>	Creates and exports a csv file
<code>left_join()</code>	Adds a column or table to an existing column or table
<code>c()</code>	Creates a vector or list (often to make a column)

rep()	Replicates elements of a vector or list a certain number of times
nrow()	Returns the number of rows in a data frame
group_by()	Groups by a variable, often before being fed to summarize()
summarize()	Creates a new data table with the results of a calculation
ifelse()	Returns one result if a condition is met, and another result if the condition isn't met
between()	Shortcut for setting a between range
mutate()	Adds new variables while retaining old variables in a data frame
for()	Begins a for loop
ggplot()	Creates a plot
aes()	Initializes aesthetic mapping of plot (assigns x and y variables, fill variables, etc.)
geom_point()	Adds a layer of points to a plot
geom_col()	Adds columns to a plot
gg_title()	Adds a title to a plot
theme()	Adds and adjusts details to a plot such as axis labels and label heights
element_text()	Adjusts text
labs()	Adjusts labels
scale_fill_manual()	Customizes plot colors

### Glossary of operators used:

Operator	Use
<-	Assigning an operation (instructions) to an object
=	Assigning instructions within an argument
\$	Creating or selecting a column from data table or data frame
%>%	'Piping' one operation (set of instructions) into another
+	Addition of numbers or a sequence of numbers (like columns)
-	Subtraction of numbers or a sequence of numbers
*	Multiplication of numbers or a sequence of numbers
/	Division of numbers or a sequence of numbers
>	Greater than
<=	Less than or equal to