**Introduction to R Studio**

*Tasks*

Tour of R Studio, Commands, Assignment Operator, readr

*Introduction*

Yesterday we installed R, R Studio, and a number of basic packages to increase functionality. Today we are going to get the lay-of-the-land in R Studio and learn how to enter some basic commands.

*Tour of R Studio*

Before you start R Studio, create a new folder named Lab10. You can place this folder wherever you like on your computer. Download the data file for today and put it in this Lab10 folder. Once you have done this, start R Studio.

As we saw yesterday, when we start R Studio, the interface window has three panels. The biggest is on the left and is called the console. If we imagined just the console by itself, we can get an idea as to what base R looks like. It is really just a command line prompt with some options in a menu bar at the top.

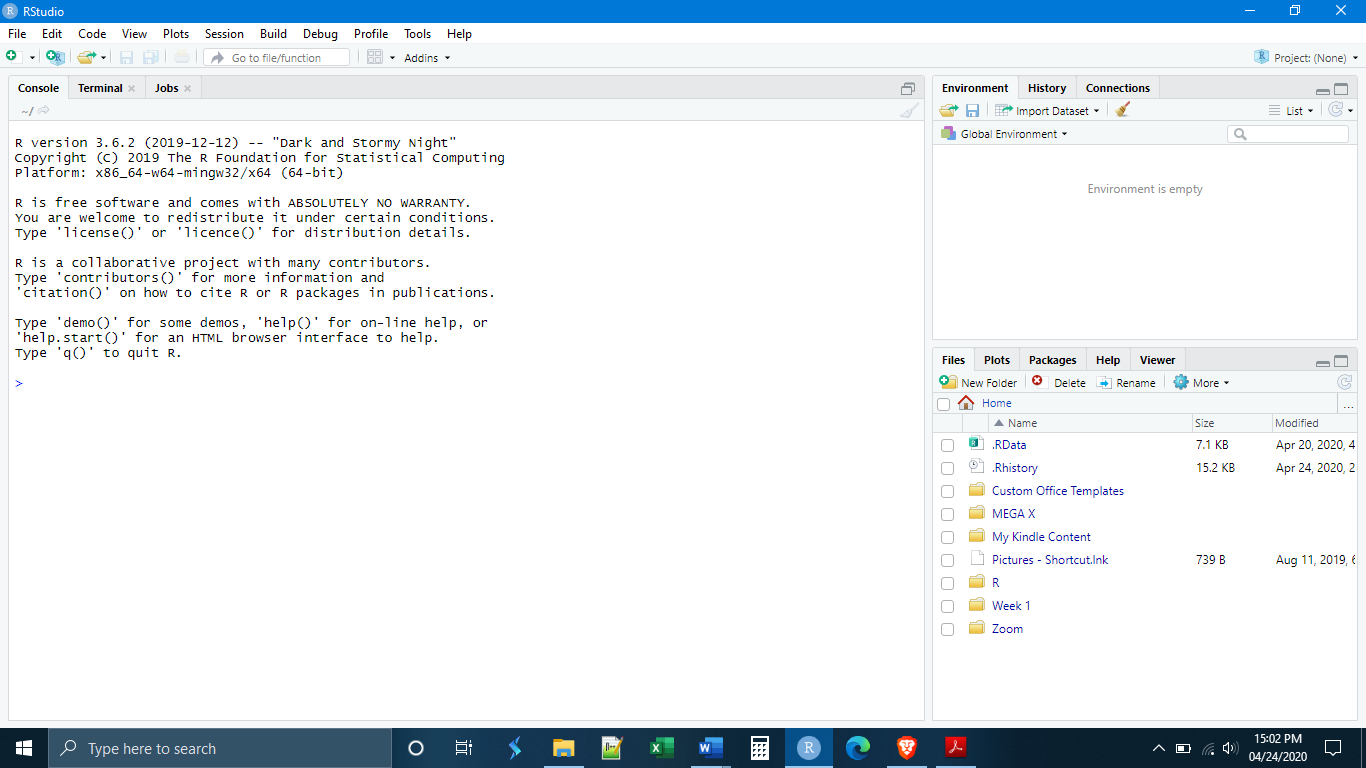
In the top right we see another panel. This panel contains information about the environment and history of your work in R. The environment includes what data have been imported, what functions are active, and what saved objects are active. The history is just what it sounds like, a history of the commands you have executed. You can toggle between the environment and the history through the tabs at the top of the panel.

The bottom right panel is a miscellaneous panel with a number of tabs granting access to a number of different things. The first tab is the “files” tab. Like other command line programs, unless told to do otherwise, R Studio will look for files in what is called the “working directory”. A “directory” is a word used in computer science to refer to a location where files are stored, in other words a folder. The working directory is the folder that you are currently “inside” of. Upon opening R Studio, the working directory will be set as the folder where R Studio has been installed. The files tab lists the files in the current working directory. We will see ways to change the working directory later. We will not worry about the “connections” tab for now.

The next tab is the “plots” tab. If we click on it, we will see that it is empty. Plots that are generated from the console will appear here.

After the plots tab there is a “packages” tab, which will show what packages are installed. Note that most of them have an unchecked box next to the name. We can understand what this means if we continue with our analogy from yesterday of comparing R packages to computer applications. In order to use a computer application, you have to open it first. To type in Word or browse the internet in Firefox, I first must start Word or Firefox. On the other hand, some applications on your computer are always on, such as the clock, and you do not need to explicitly open them in order to use them. Similarly, in R, many packages need to be explicitly loaded before you can use them (like the tidyverse). In other cases, however, the package is always active, so long as R is running (such as data.table). The unchecked boxes here indicate that the package has not been explicitly loaded during this session.

There are also “help” and “viewer” tabs that we will not worry about at this time.



**Files, Plots, Packages**

**&**

**Help**

**Environment**

**&**

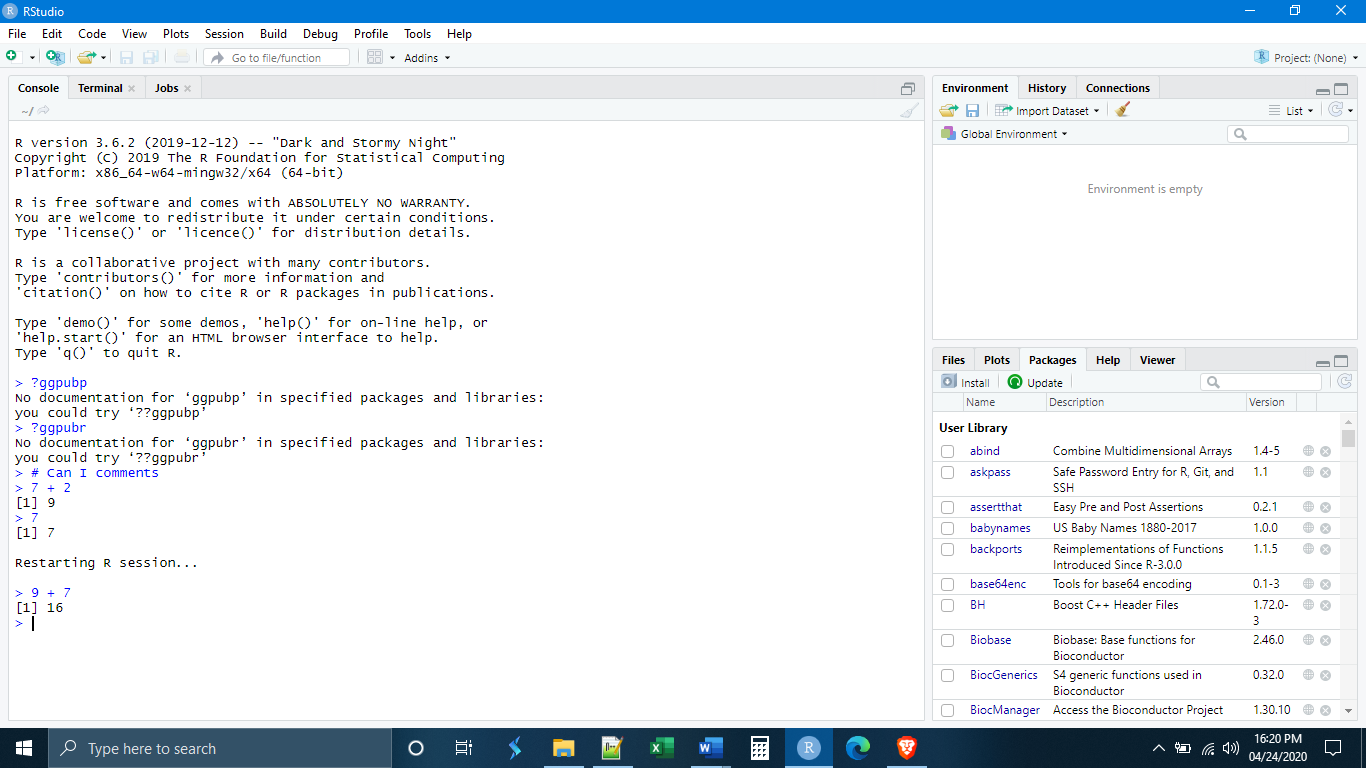
**History**

**Console**

Let’s start by entering some simple commands into the console. We can use R like a simple calculator. In the console, type

9 + 7

If we press enter, what do we see? It simply gives the sum as the output:



We can do this with all kinds of mathematical operations. Experiment with some.

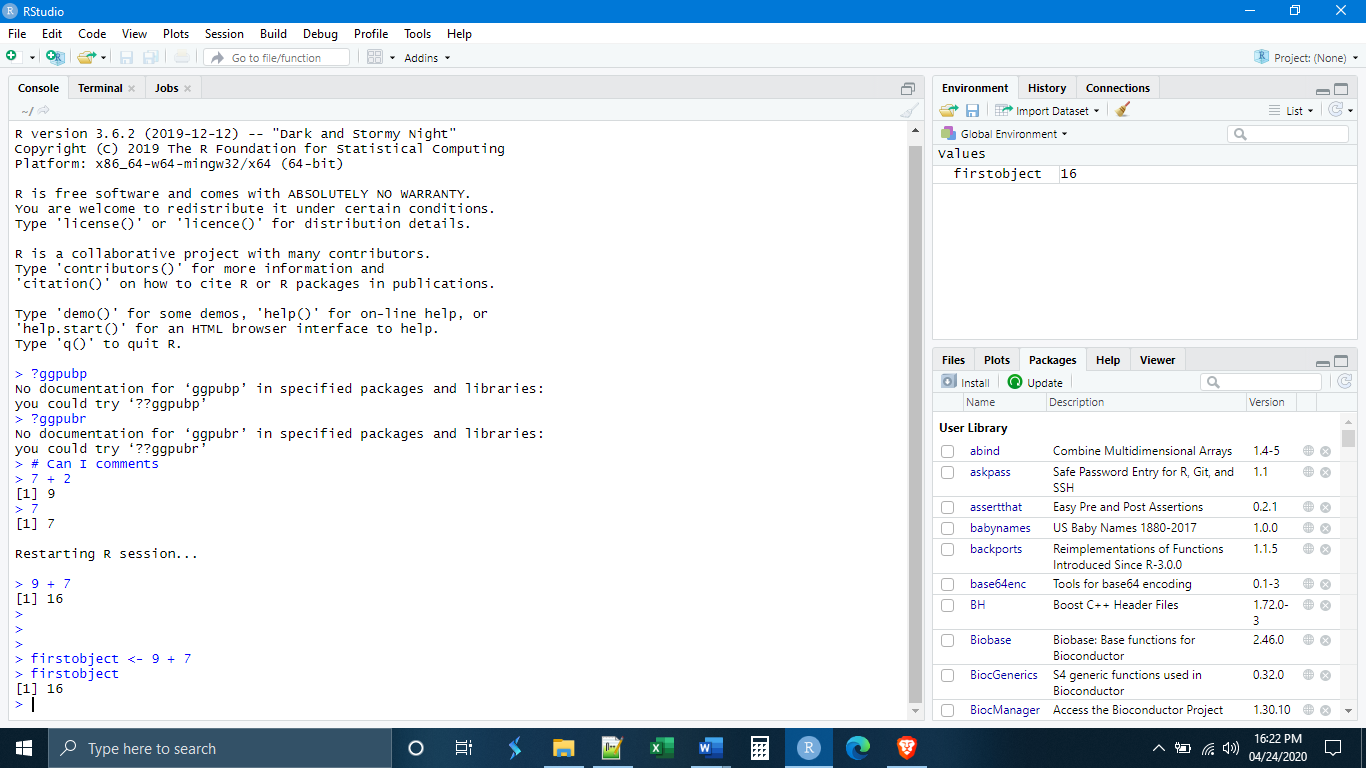
What if we wanted to save the answer of our operation for some other task later? The way things are saved in R is through what is called the “assignment operator”. This is used to create what are called “data objects” or simply “objects”. An object can be anything. It can be a number, a list of numbers, a table of numbers, a list of characters (letters), or a table of characters and numbers, a plot or graph, etc. Those different characters and numbers can also have a variety of different formats, which we will talk about later. It is important to realize that saving something and creating a data object are the same action. Every time I save something, I create a new object.

There are actually three ways to create objects, but some have advocated for a specific convention that we will follow. This convention is to only use the “left assignment operator”, which has the form <-. In other words, the “is less than” symbol followed by a dash or minus sign. It is intended to resemble a left-pointing arrow (hence “left” assignment operator). The left assignment operator uses the following syntax:

NameOfSavedObject <- ObjectBeingSaved

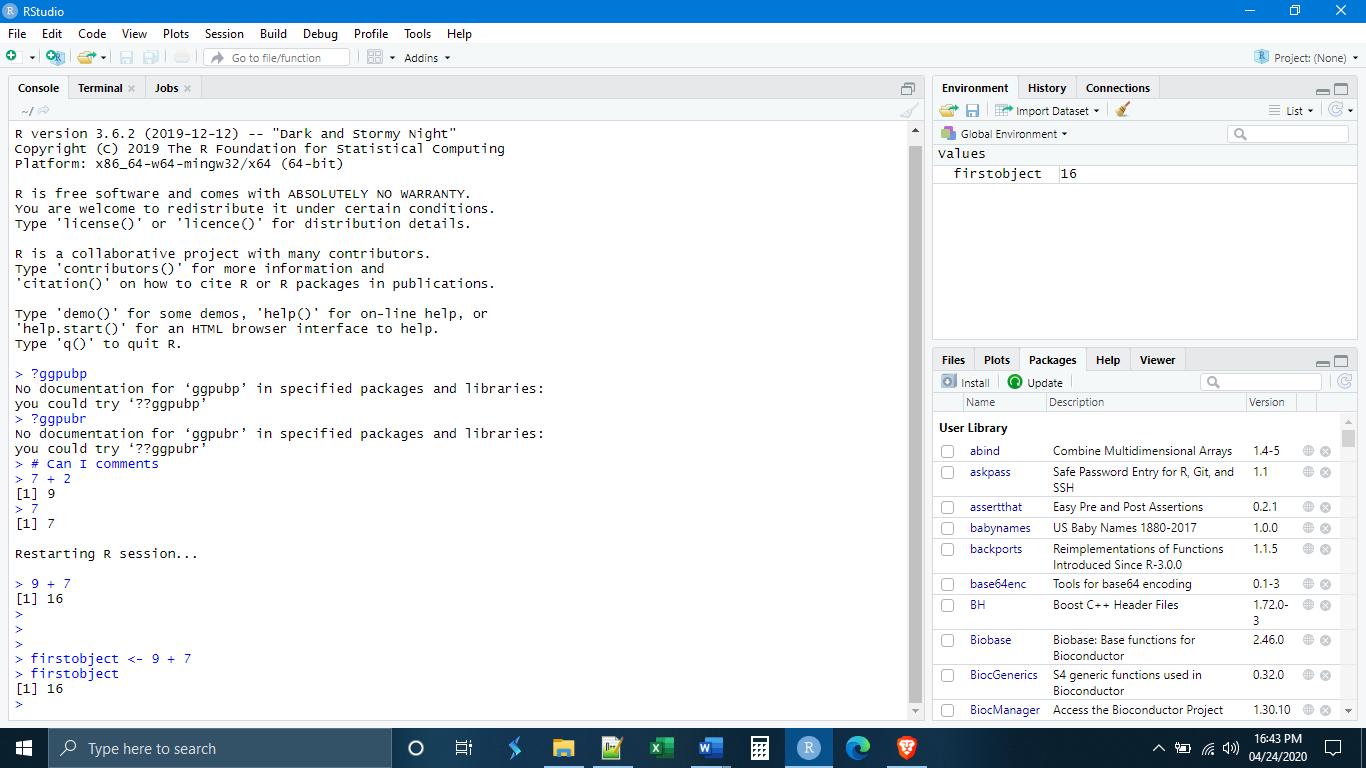
Whatever is to the left of the arrow is the new name of the object that you are creating, and whatever is to the right of the arrow is the content of what you want that object to be. Let’s make our first object:

firstobject <- 9 + 7

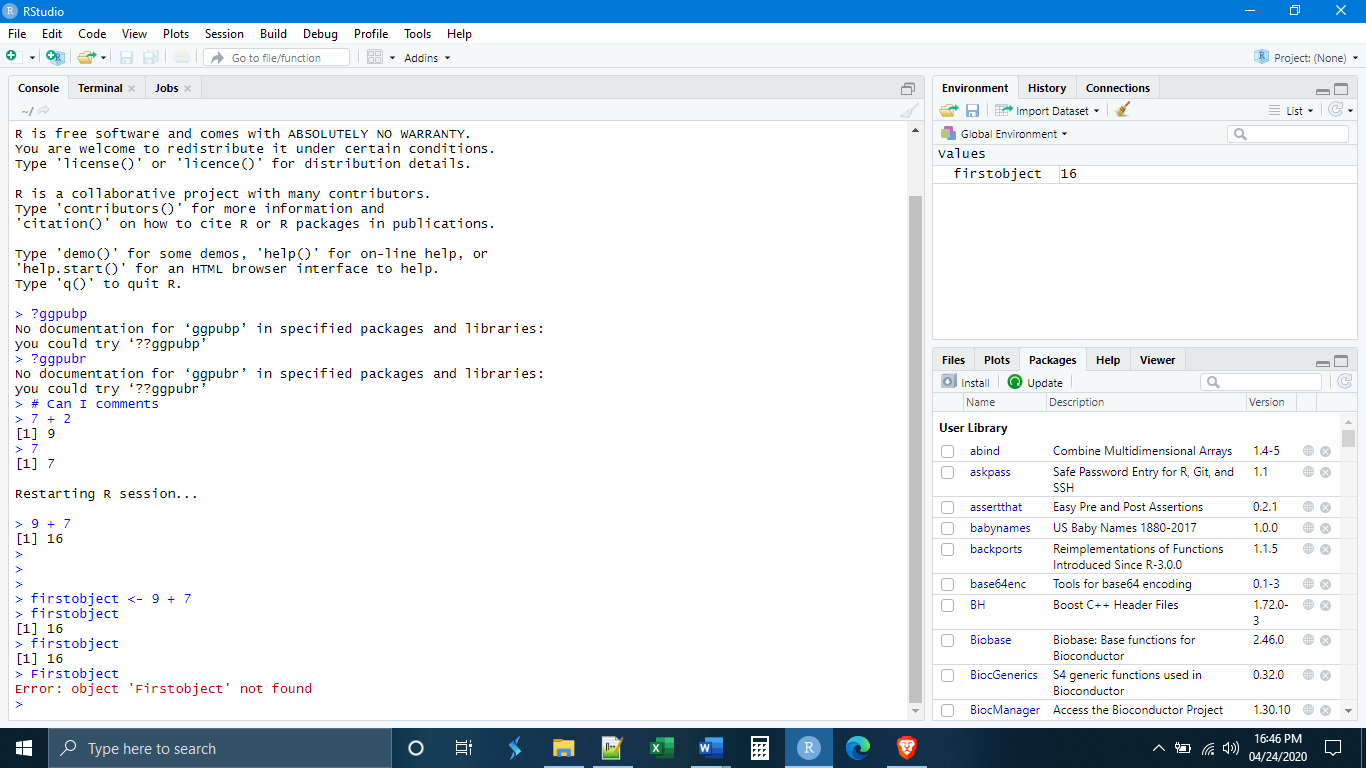


A few things need to be noted here. First, see that we now have something in our environment! We have the name of our object firstobject, followed by a description of that object, in this case just a single number, 16.

Second, note that R Studio did not give us an output in the console. This will be the case whenever we create an object. If we want, we can see our object by “calling” it, typing it in the console and pressing enter. What do we see when we do that?



The third thing that we want to note is that R did not save the object as “9 + 7”. R recognized that both were numbers and simply saved the sum. What happens if we try to call our object by typing: Firstobject?

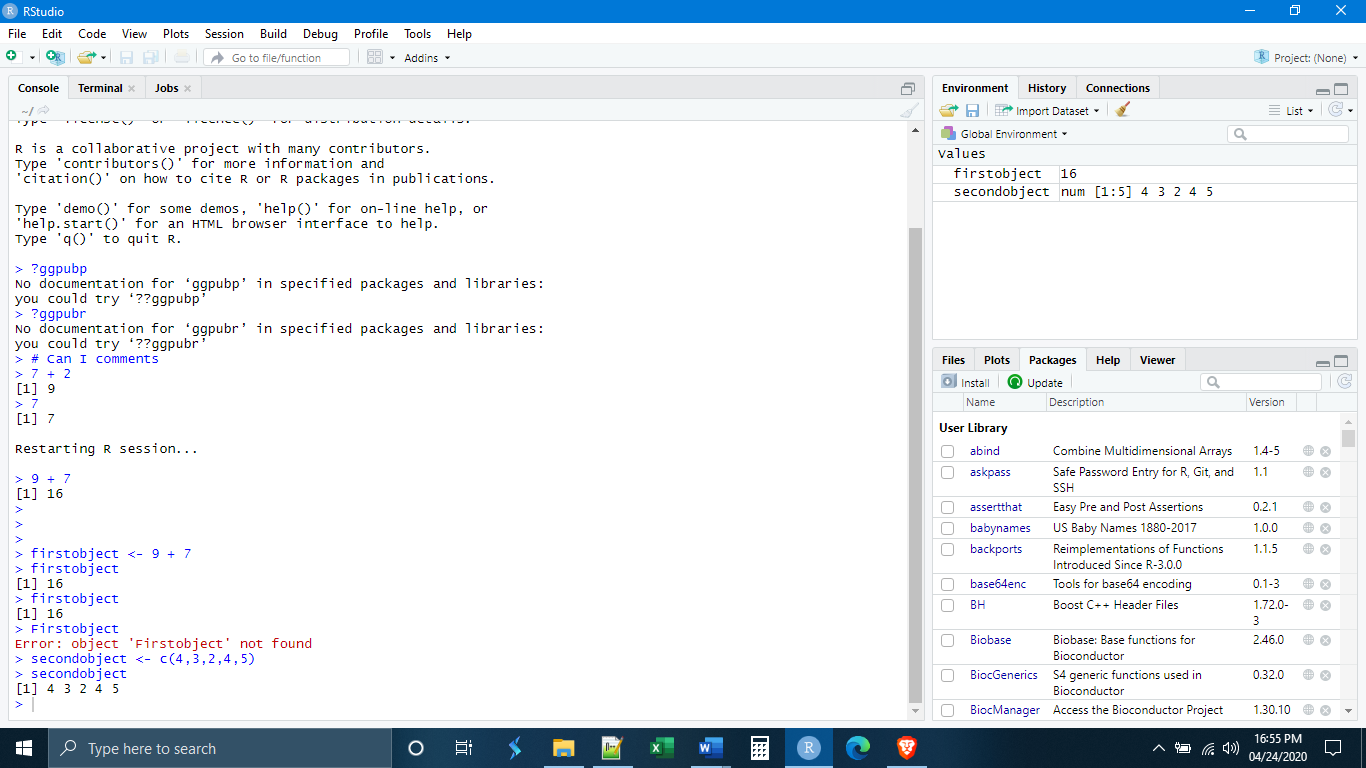


R is case-sensitive. If you try to call an object you must type it in exactly as you saved it. Thankfully, if you are working in R Studio, R Studio will suggest objects as you type by using autocomplete, helping to eliminate this problem.

Let’s make a second object, a vector.

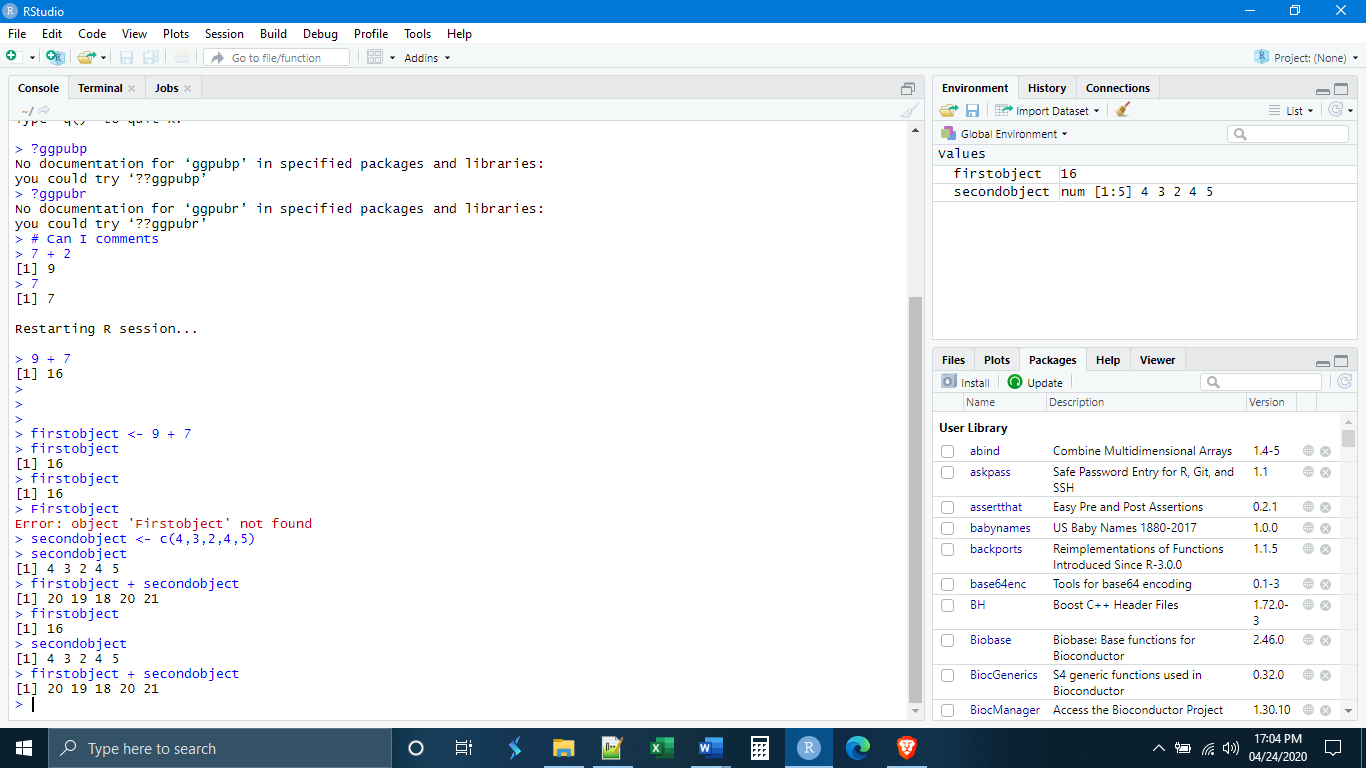
secondobject <- c(4,3,2,4,5)

What happens when we call this second object?



A vector is simply a one-dimensional collection of data points, be they numbers or letters. The way that you indicate what is inside of the vector is with the function c(), meaning “concatenate”, where what is inside of the parentheses are the contents of the vector. We might be tempted to call this a list, but a list means something specific in R, so we should be careful not to use that word. Remember, if you want to save something you always need to use the assignment operator. Simply entering the vector will not save it.

How do our objects interact? What happens if we add firstobject to secondobject?



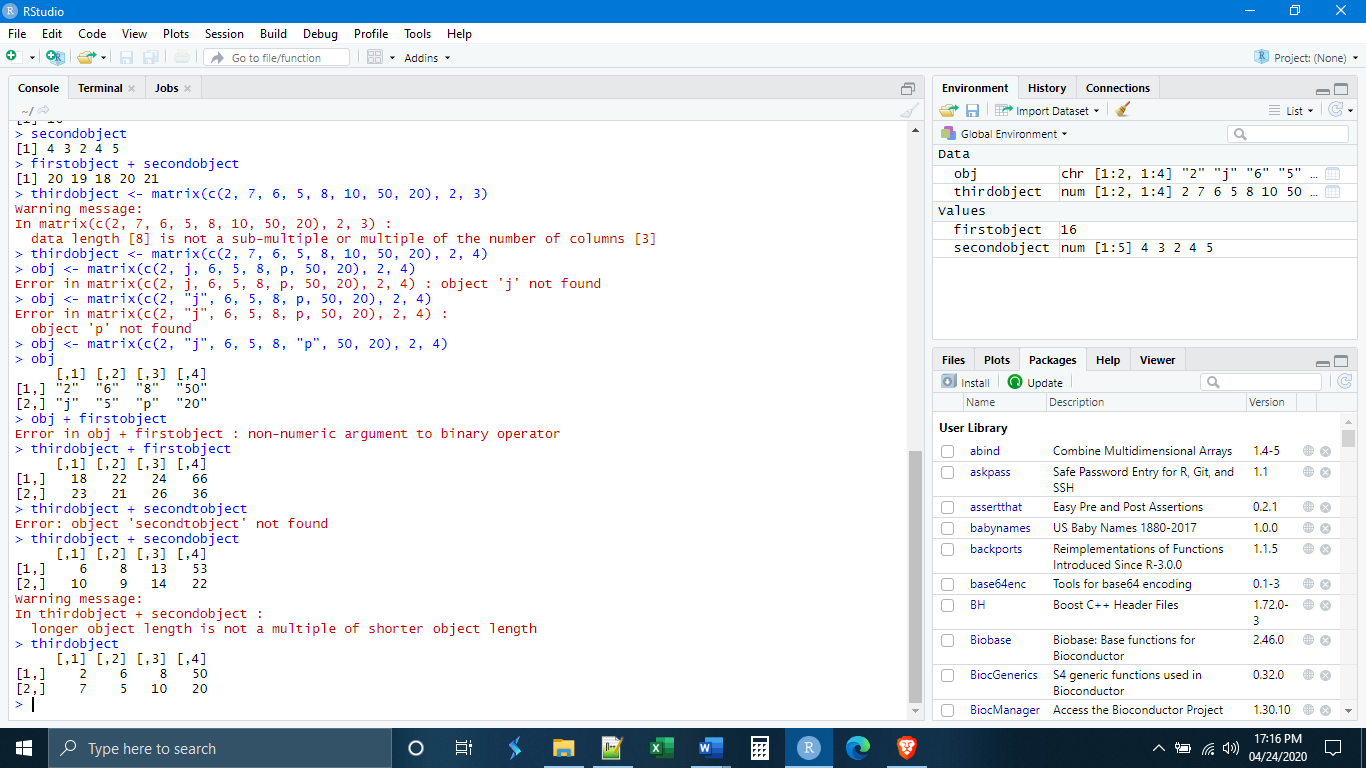
We see that firstobject was added to every element of secondobject. This is a very important point about vectors. Anything done to a vector is done to each element of that vector.

Before we move on to other ways to enter commands in R Studio, let’s make one more object, a matrix.

thirdobject <- matrix(c(2, 7, 6, 5, 8, 10, 50, 20), 2, 4)

The matrix() function will turn the input into a matrix. A matrix is a two-dimensional collection of data elements (again, be they letters or numbers). The syntax for the matrix() function is:

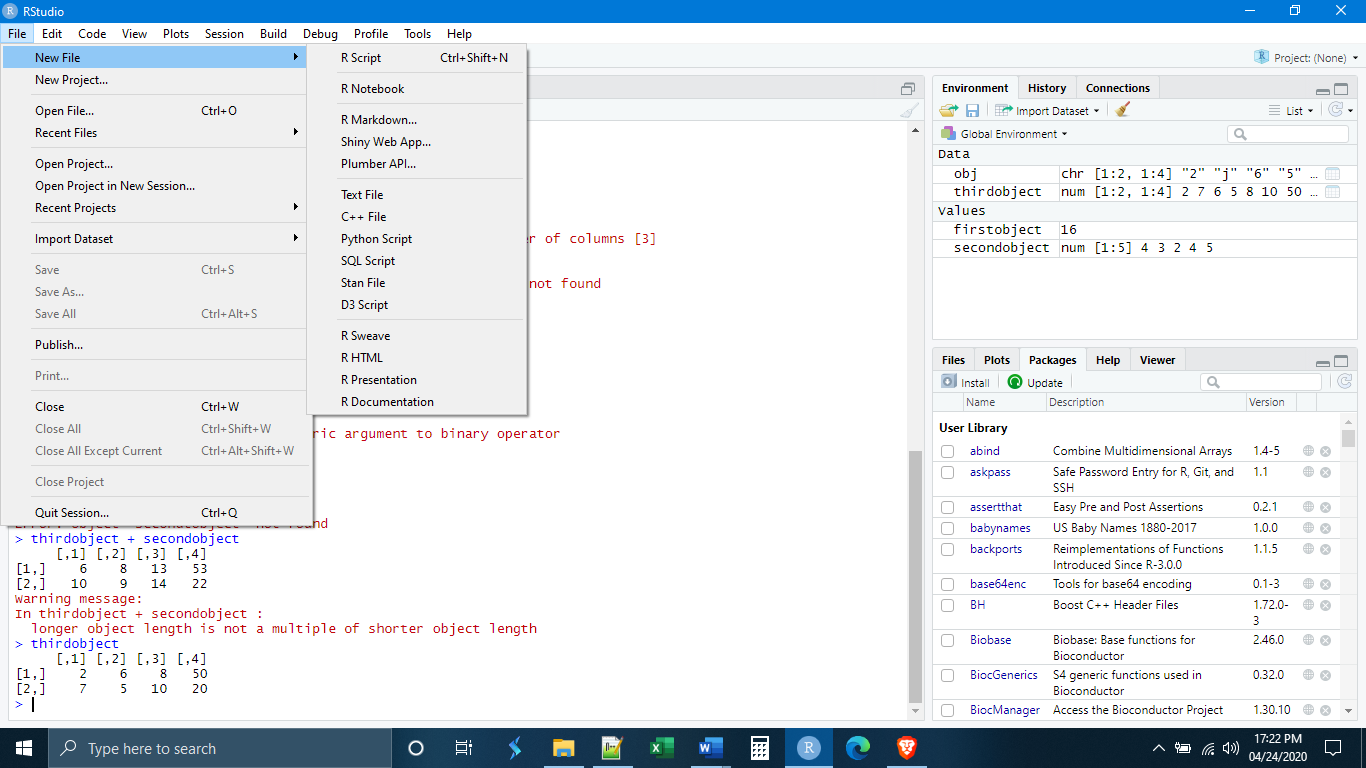
matrix(c(ListOfElementsInTheMatrix), NumberOfRows, NumberOfColumns)



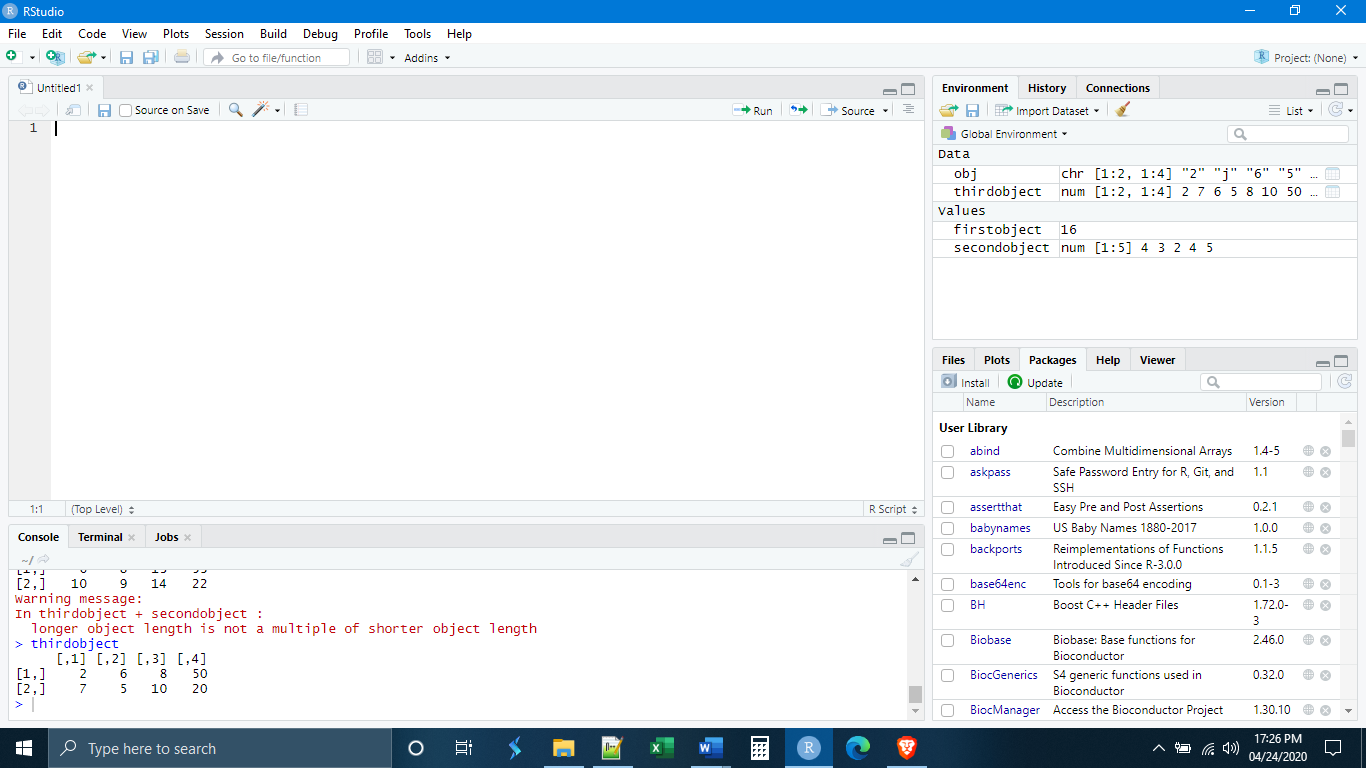
*Scripts, Notebooks, and Markdown Files*

The console is fine for doing simple calculations as we have been doing, but it has some serious drawbacks. First, you will notice that you can only feasibly enter commands one line at a time. Second, going back to previously entered commands can be a bit cumbersome. Third, there isn’t a really good way to save your work and come back to it.

All of these difficulties can be overcome by the other ways to enter commands in R Studio. If we go to file, new file. We see that there are a number of different file types that can be created in R Studio. We will focus on the first three, R Scripts, R Notebooks, and R Markdowns.

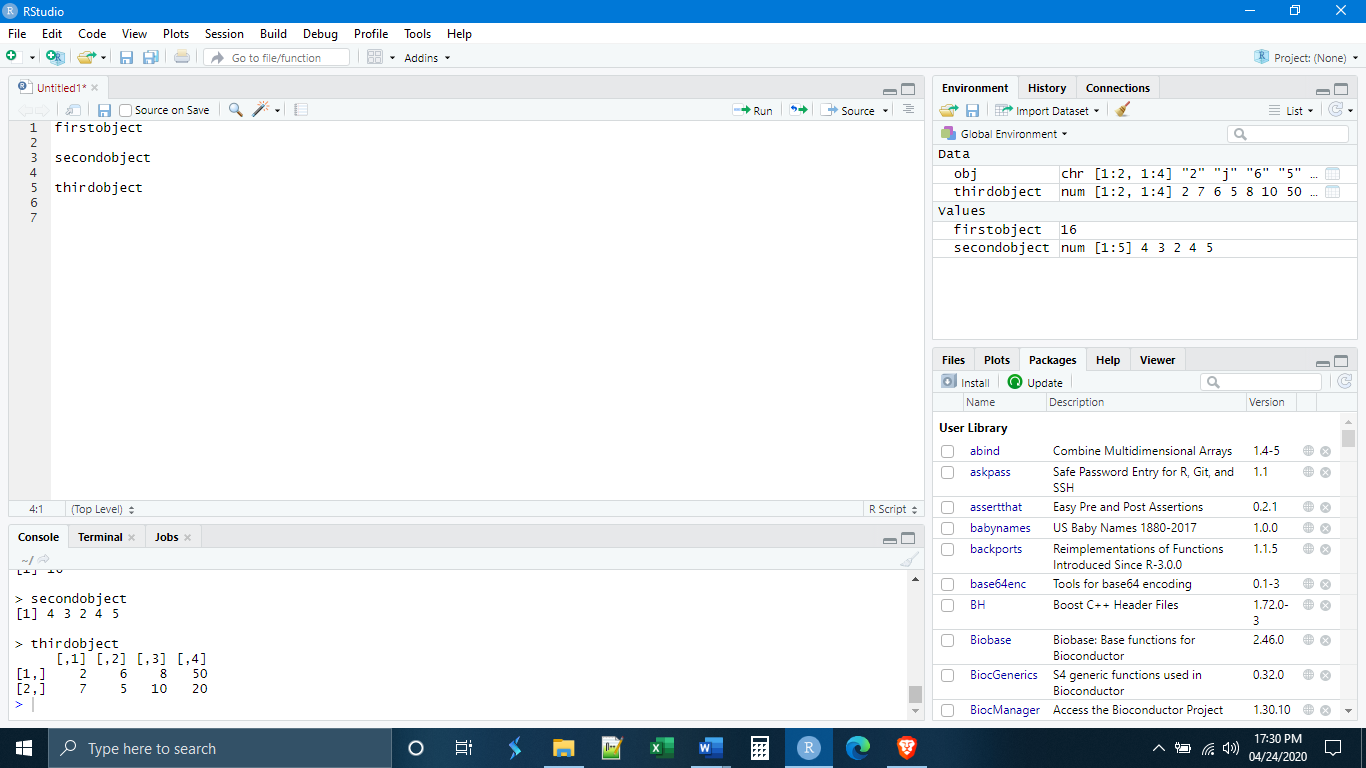


Begin by opening a new R script. Notice that immediately the layout of the panels in the R Studio window change.

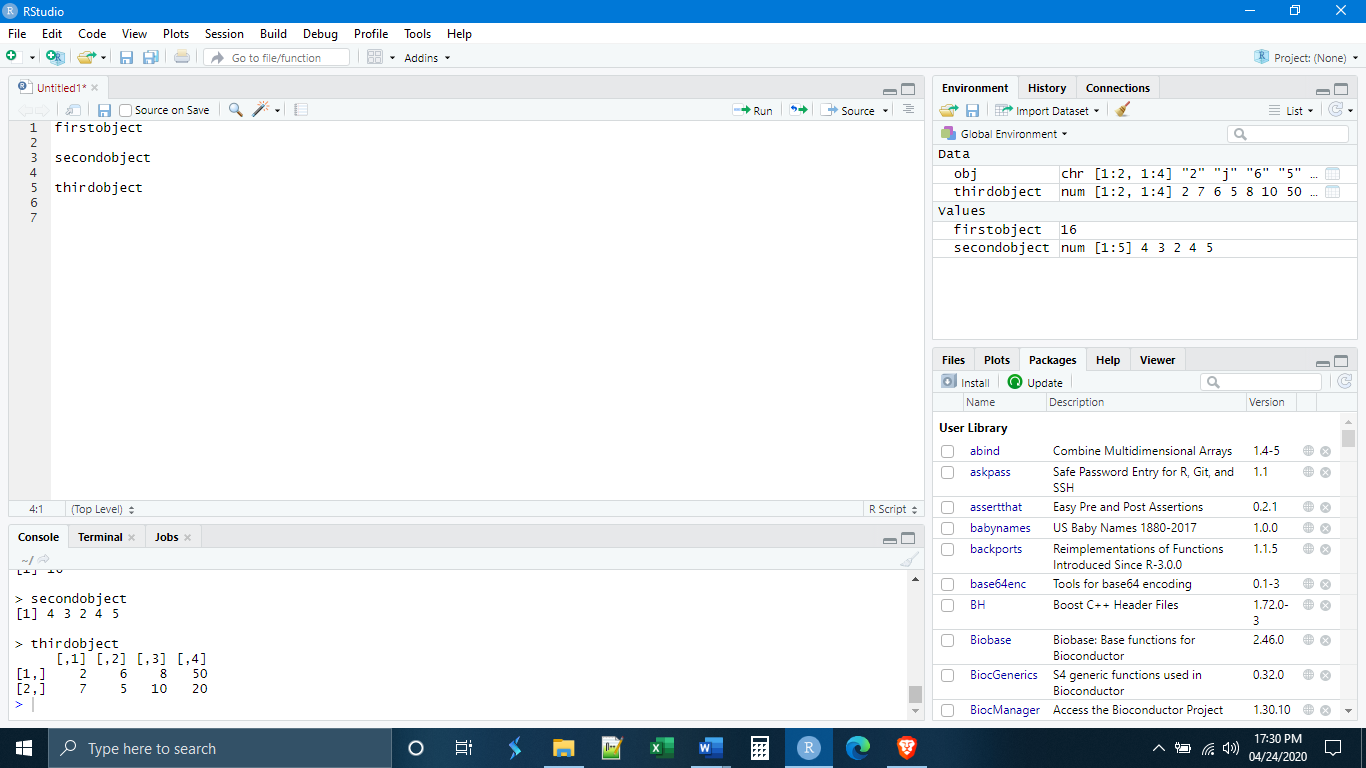


We now have a fourth panel, which is our script. The script editor works just like a simple text editor, with the difference that we can also execute R code from within it. The console panel has been made smaller and placed below our script panel.

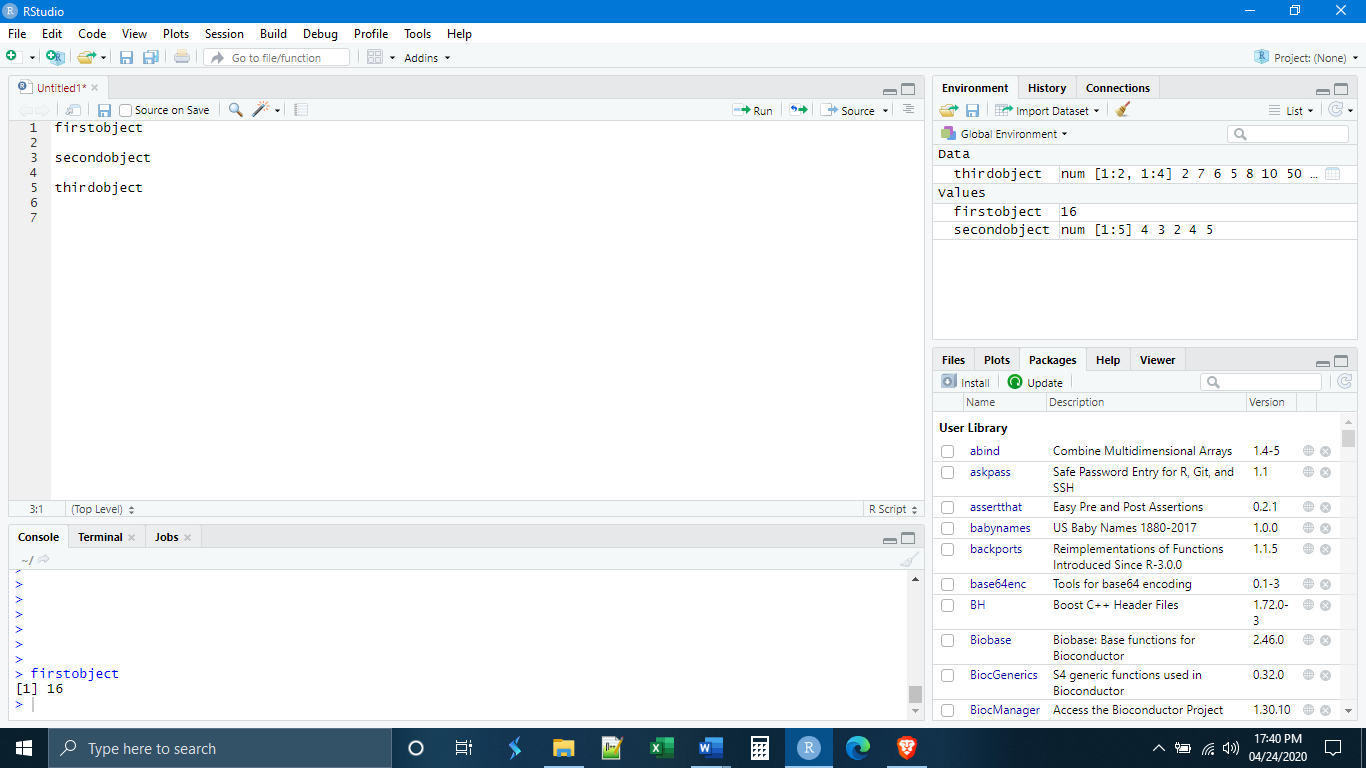
Let’s begin by entering our three objects with a space in between each of them.



In order to get the three objects on different lines, we had to use the enter key. Since this was the key we used to execute commands in the console, how do we execute commands in a script? There are two ways. The first way is to put your cursor at the beginning of the line you would like to execute and the use the keys ctrl + enter (cmd + return for Macs). The other way is to put the cursor at the beginning of the line you would like to run and then click the “run” button in the top right of the script window (see next page).

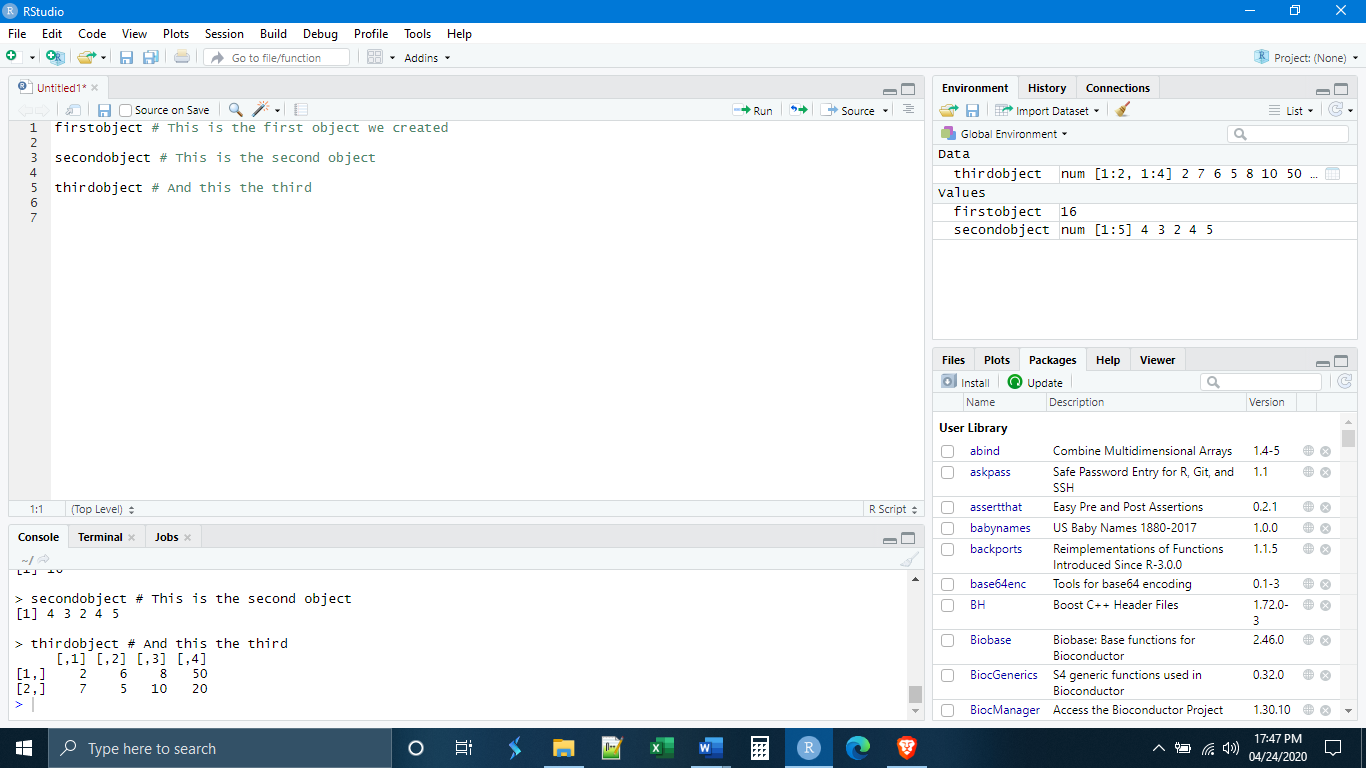


Let’s execute the first line. What do we see? The output appears in what remains of the console window below.

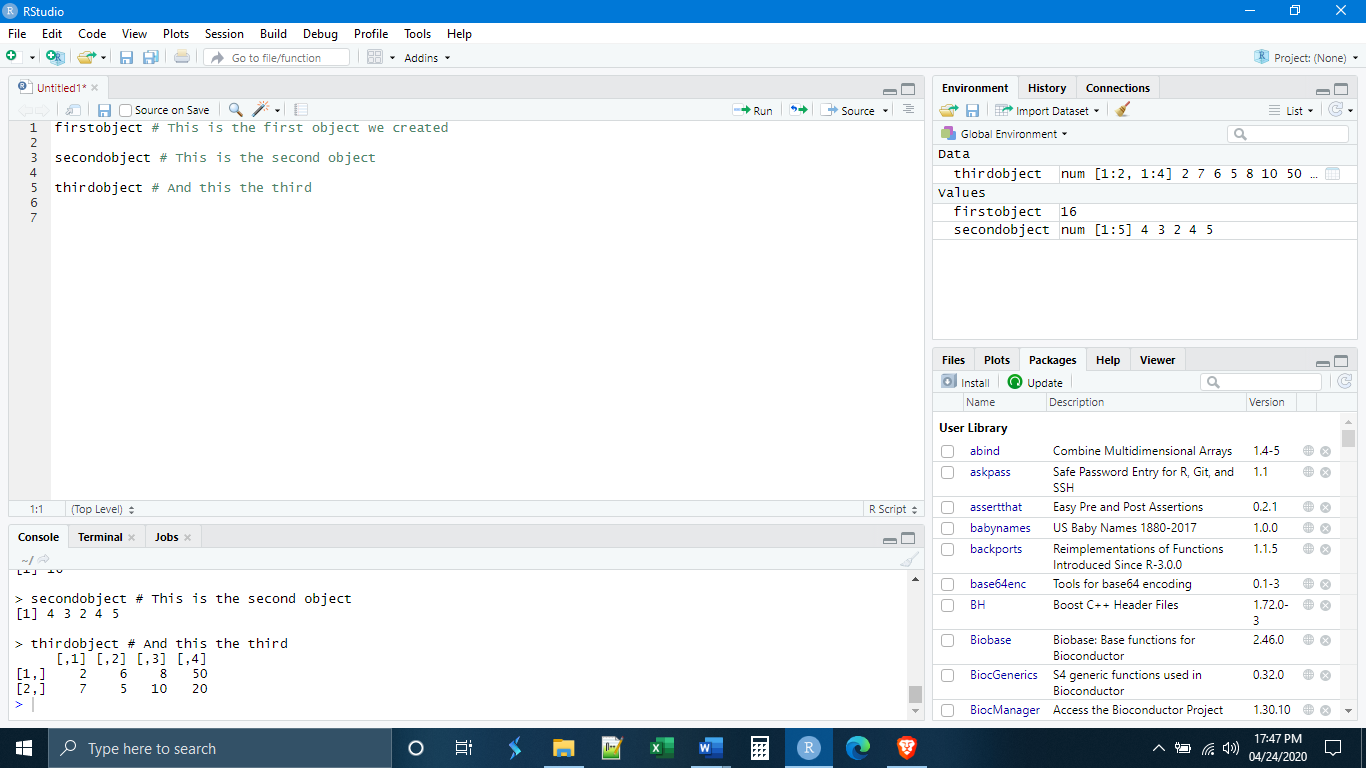


We can also execute everything in the script simultaneously by placing the cursor at the beginning of the script and using the keys ctrl+shift+enter (cmd+shift+return for Macs). Place your cursor on the first line and execute all of the code.

Using a script also allows us to make use of comments. R will ignore anything that comes after a hashtag/pound sign (#). Thus, we can put comments in our code to remind ourselves, or someone else, what we have done. When the code is executed, the comments will be ignored.

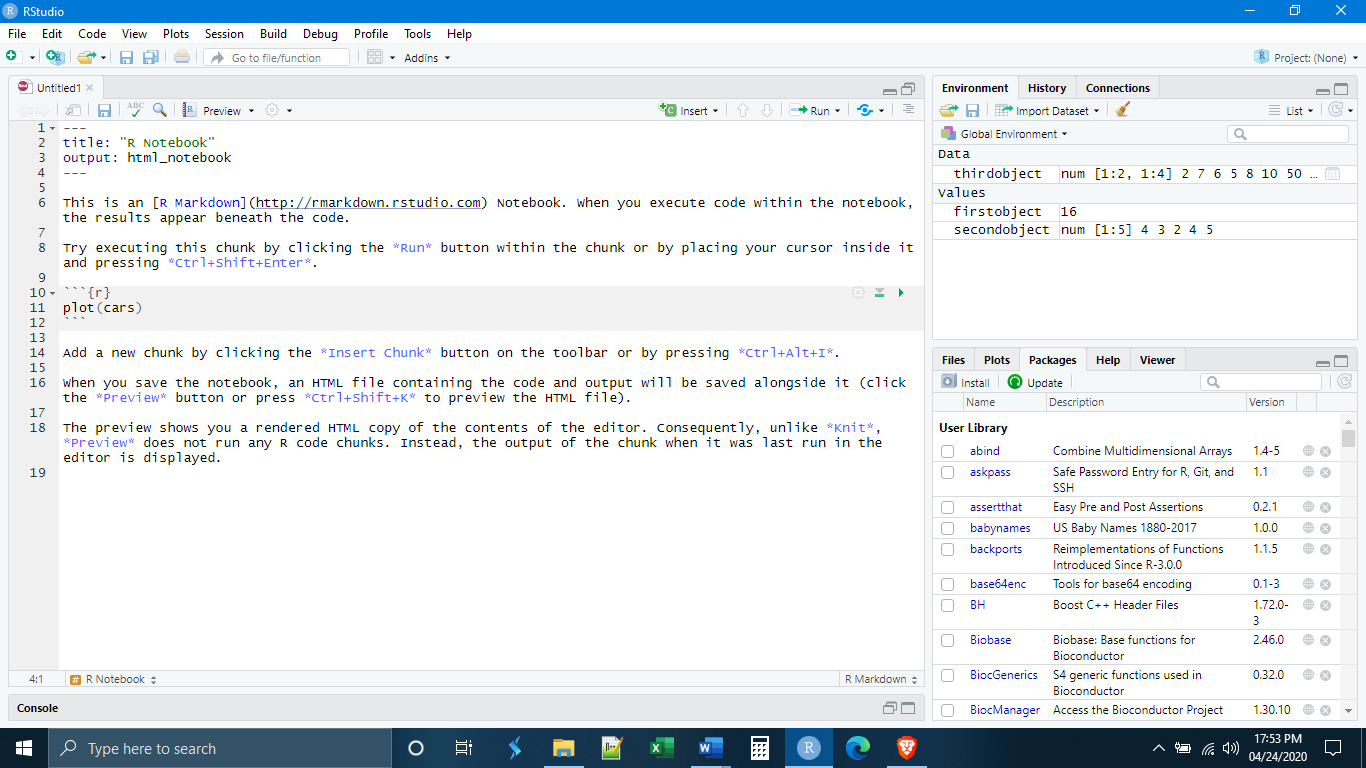


Lastly, notice that we can save our script and come back to it later using the save button at the top of the script panel. Save this script in the folder you have created.

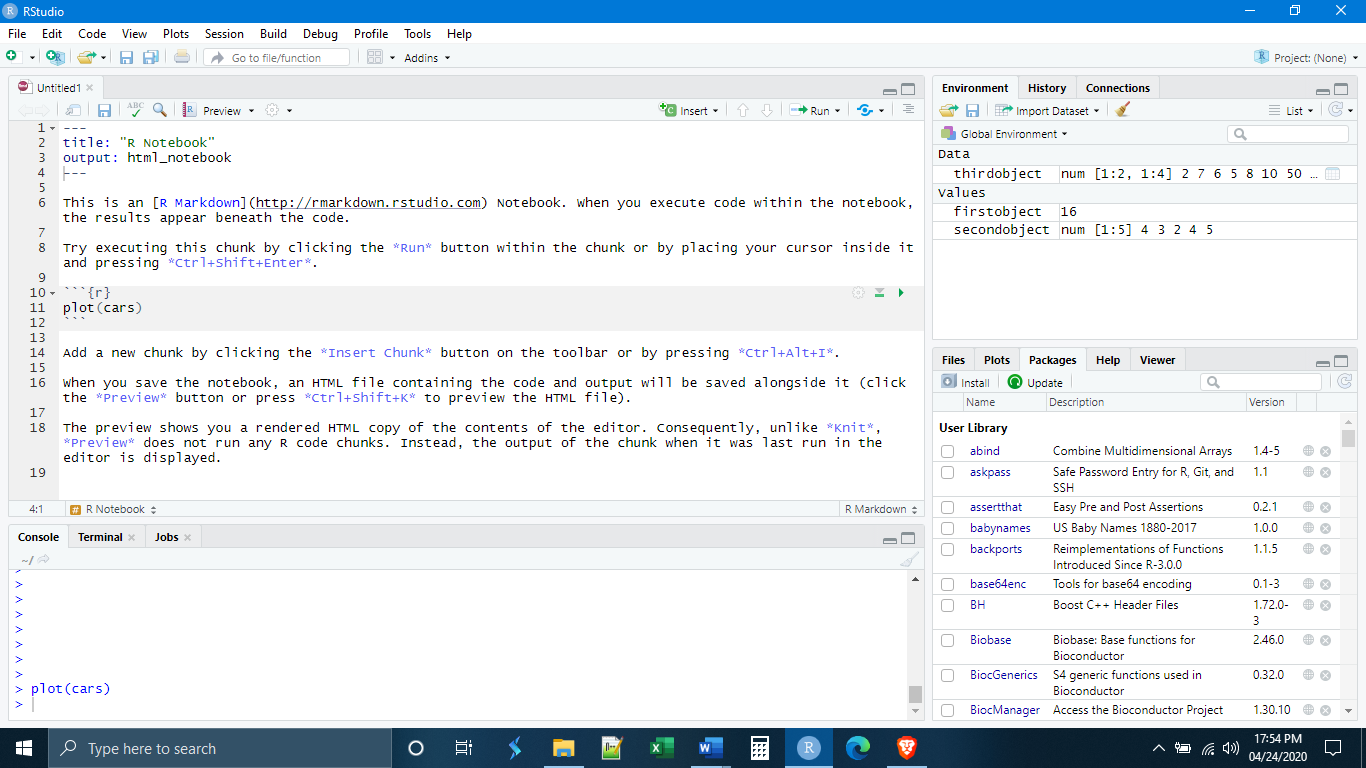


There are two other kinds of files in which you can execute R code: R Notebook and R Markdown. These are very similar. Both function as ways to organize your work in the form of a report and allow you to export your code, and the results of that code, as an .html, .docx, or .pdf.

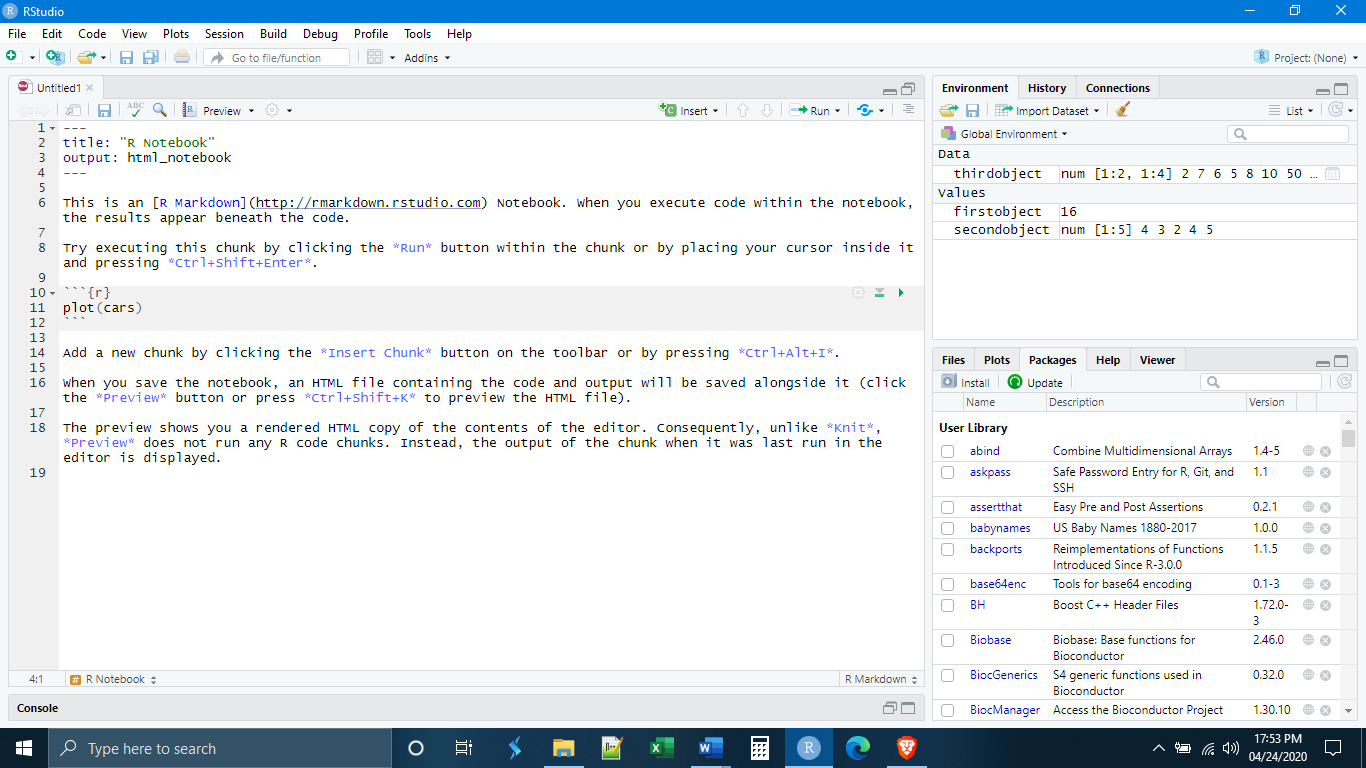
Let us begin by opening up an R Notebook. Notice that when we open the notebook, the console is completely collapsed to the bottom of the window and is no longer visible.



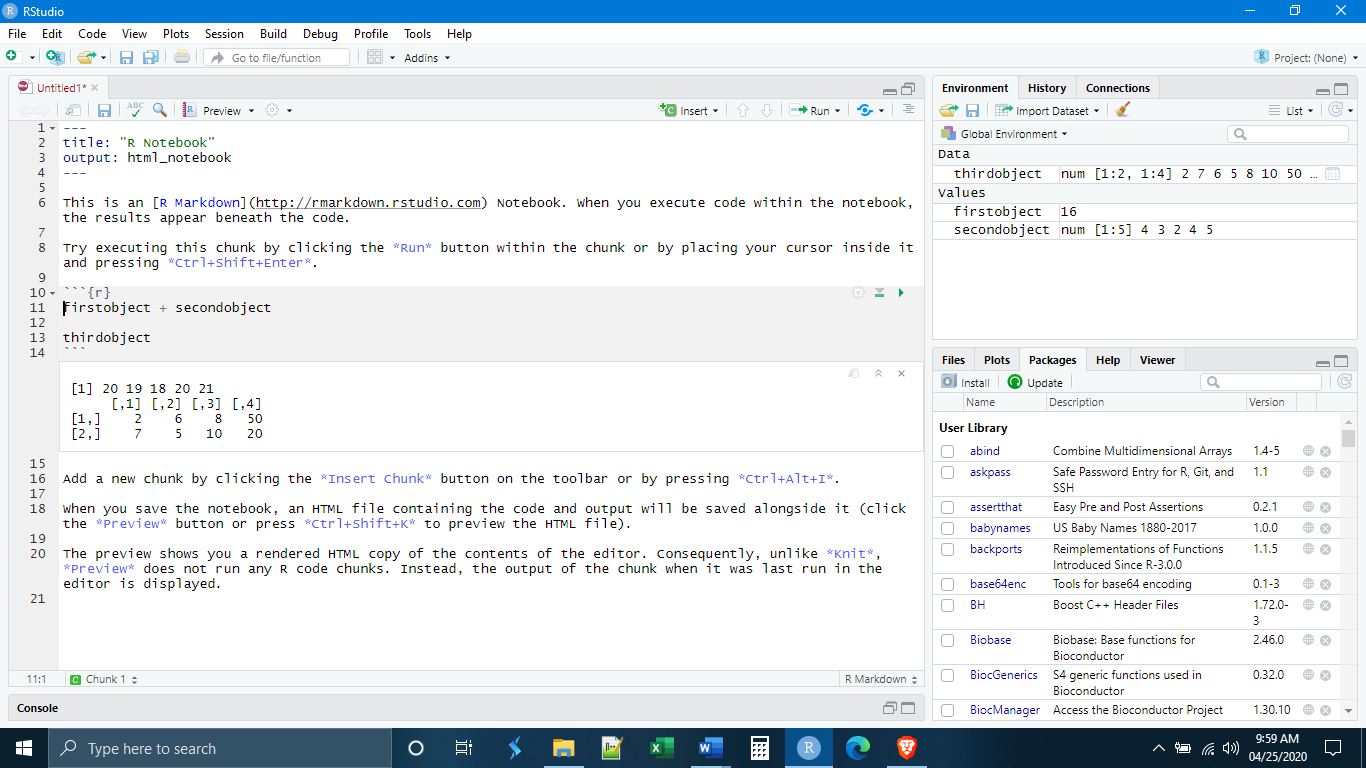
If we really want to have our console back, we can double click on it and it will pop up in the way that it did with the script. Part of the point of the notebook and markdown documents, however, is that they do not need the console, as we will see.



Unlike the console or the script, not all areas of a notebook or markdown document can execute R code. By default, the notebook or markdown document functions as a basic text editor. In order to execute R code, it must be put inside of a “chunk”. In the template that appears at the start of a new notebook or markdown document there is already one chunk present. The chunk is identified at the start by the sequence ```{r} and is ended by ```. The apostrophe-like symbol is not an apostrophe but a different character found just to the left of the number 1 key on your keyboard. Each chunk must be given a unique name at the beginning ```{r NameOfChunk}.



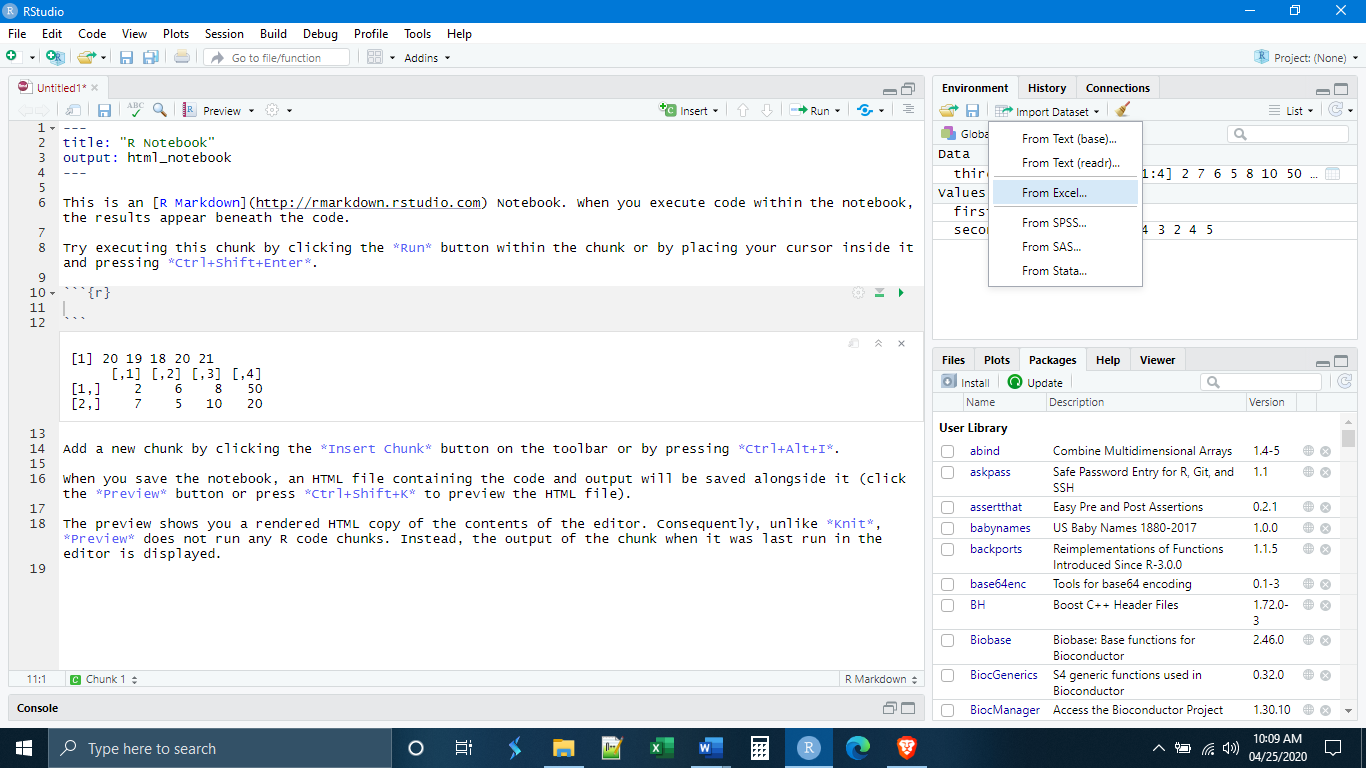
The results of any code executed inside the chunk will appear just below the chunk.



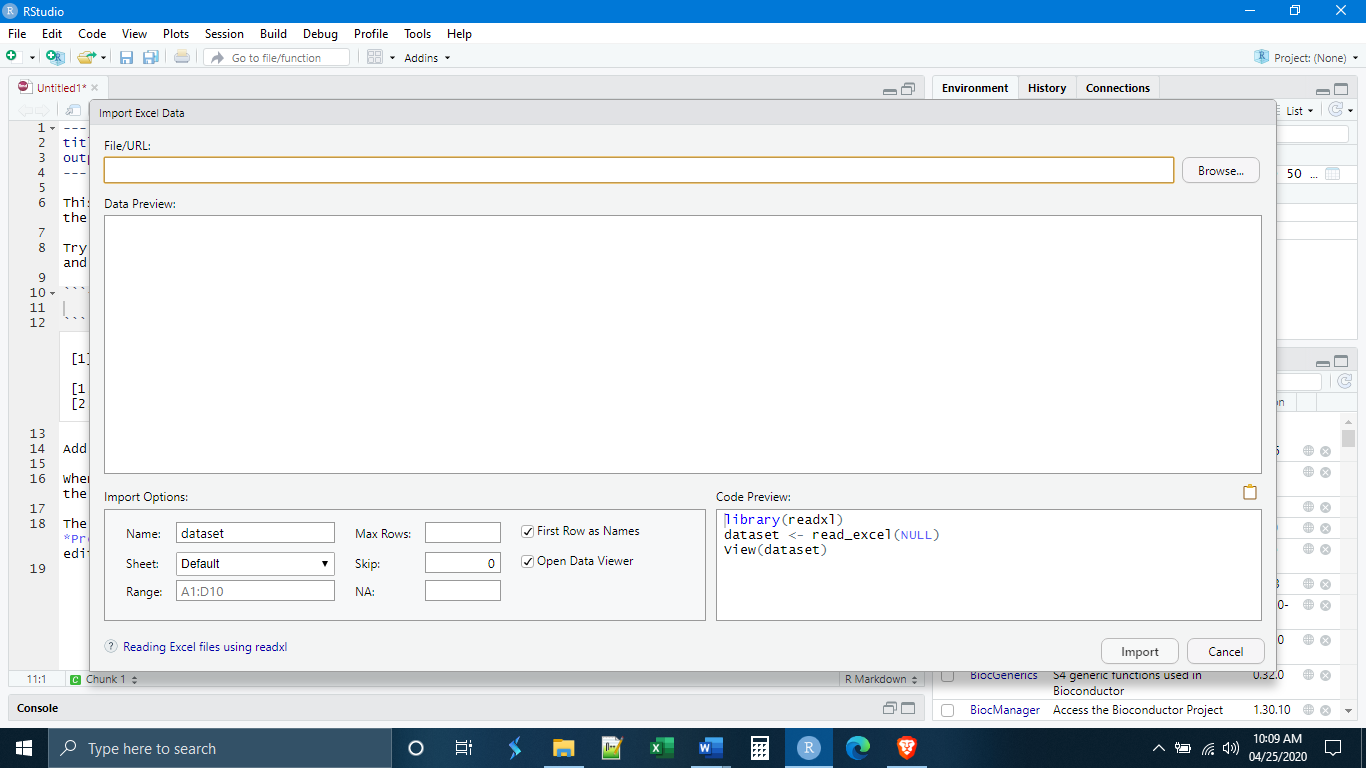
*Reading Files*

It is rare that you will enter data manually into R as we have been doing thus far. Most likely, your data will be stored in some other spreadsheet program, such as Excel, and you will want to import it, or “read” it, into R in order to work with it.

Within R Studio there are two ways that data can be read-in. The first way, which we will not be using, is through the environment panel. Click on the button that says “Import Dataset” and you will be prompted with the next steps.



Notice that this is really just a way for R to generate the code for you automatically, and R Studio shows you the code that will be entered.



The second way to read in data is by manually entering the code. Even here, there are a number of ways to do this. For now, we will focus on the read\_csv() function from the readr package in the tidyverse. I highly recommend that you take a look at the readr cheat sheet that is on blackboard. We can see that the basic syntax for read\_csv() is:

read\_csv(“filename.csv”, col\_names = TRUE/FALSE)

There are other options in read\_csv(), which we won’t worry about at the moment. Note that this will only read in files that are in comma-delimited .csv format, which is a standard format for data tables. To convert an Excel file to .csv, use Save As and change the file format to “CSV (Comma delimited)”. It is generally a bad idea to read in Excel files directly because Excel adds a lot of additional formatting that can cause problems with data importation.

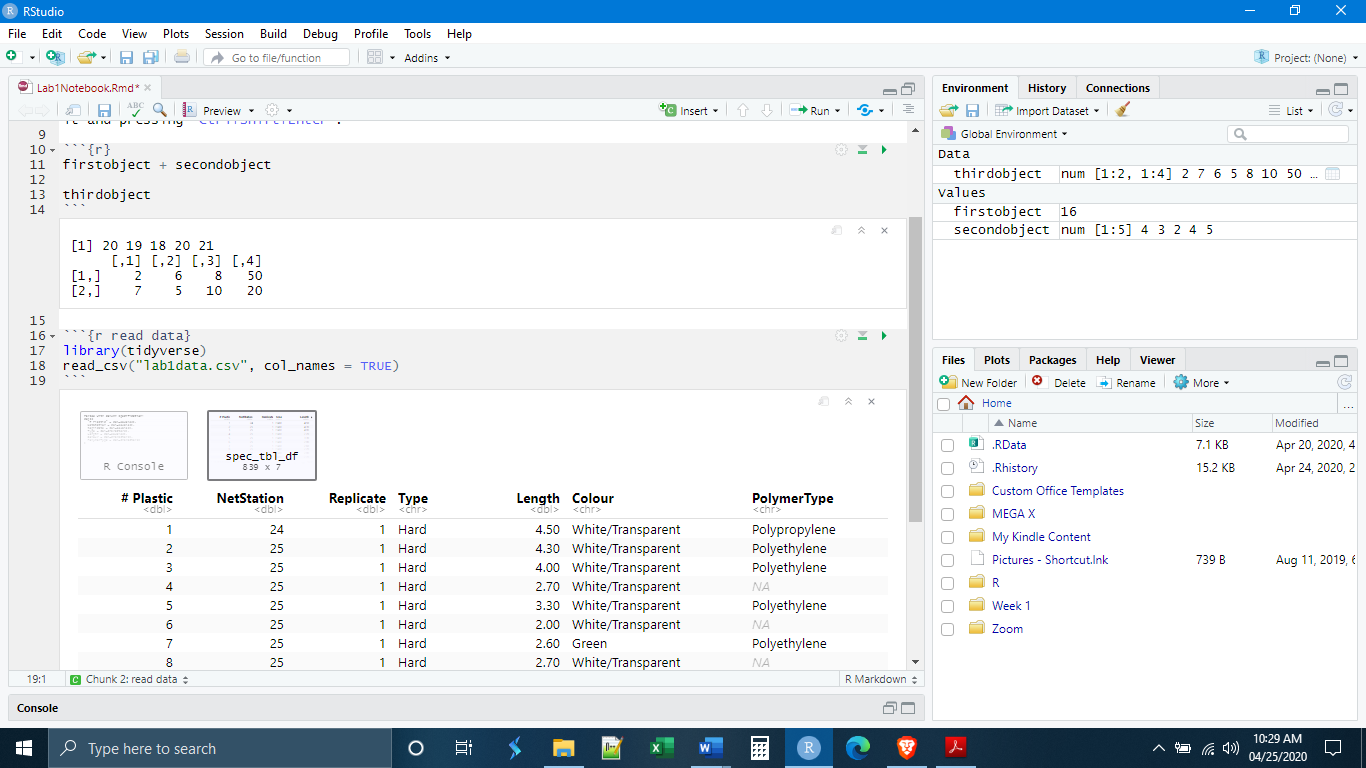
First, we want to save the notebook we are working with. Save it with whatever name you like, but it must be saved in the same Lab2 folder where you downloaded the data for today. This will be our working directory. Recall that, unless told to do otherwise, R will only look for data files within the working directory. By default, the working directory is the folder that the script/notebook/markdown is saved in.

When you have done this, make a new chunk and type the following code:

libraries(tidyverse) #readr is not part of base R, so we need to load it before use

read\_csv(“lab1data.csv”, col\_names = TRUE) # our .csv file includes column names

You can then execute the code line-by-line or all together.



You can see that the data appeared just below the chunk in an interactive data table. But notice that the data are not part of our environment. This is because we never turned them into an object using the assignment operator.

Let’s make a third chunk and this time save the imported data as an object. There is no need to load tidyverse from the library again. You only need to do this once per session.

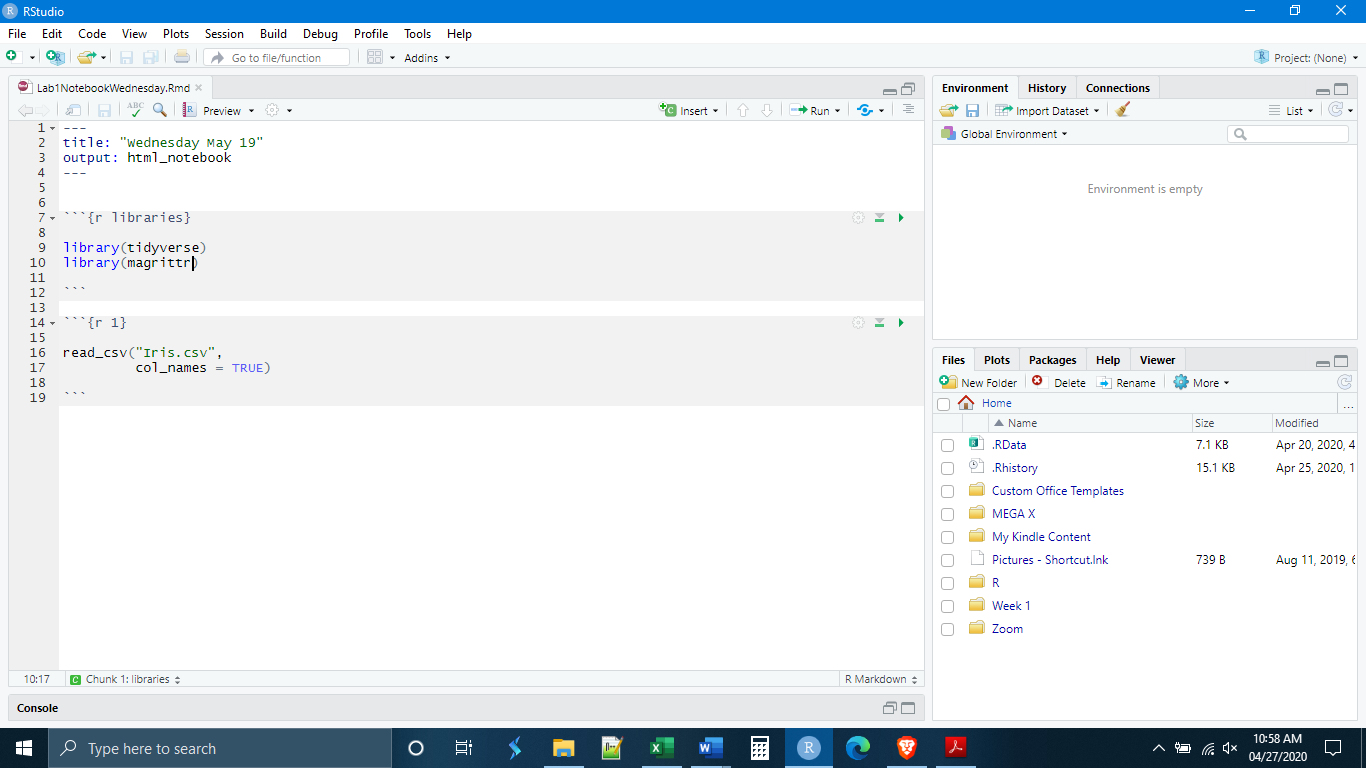
Now we can see that our imported data appears in our environment as one of the objects.



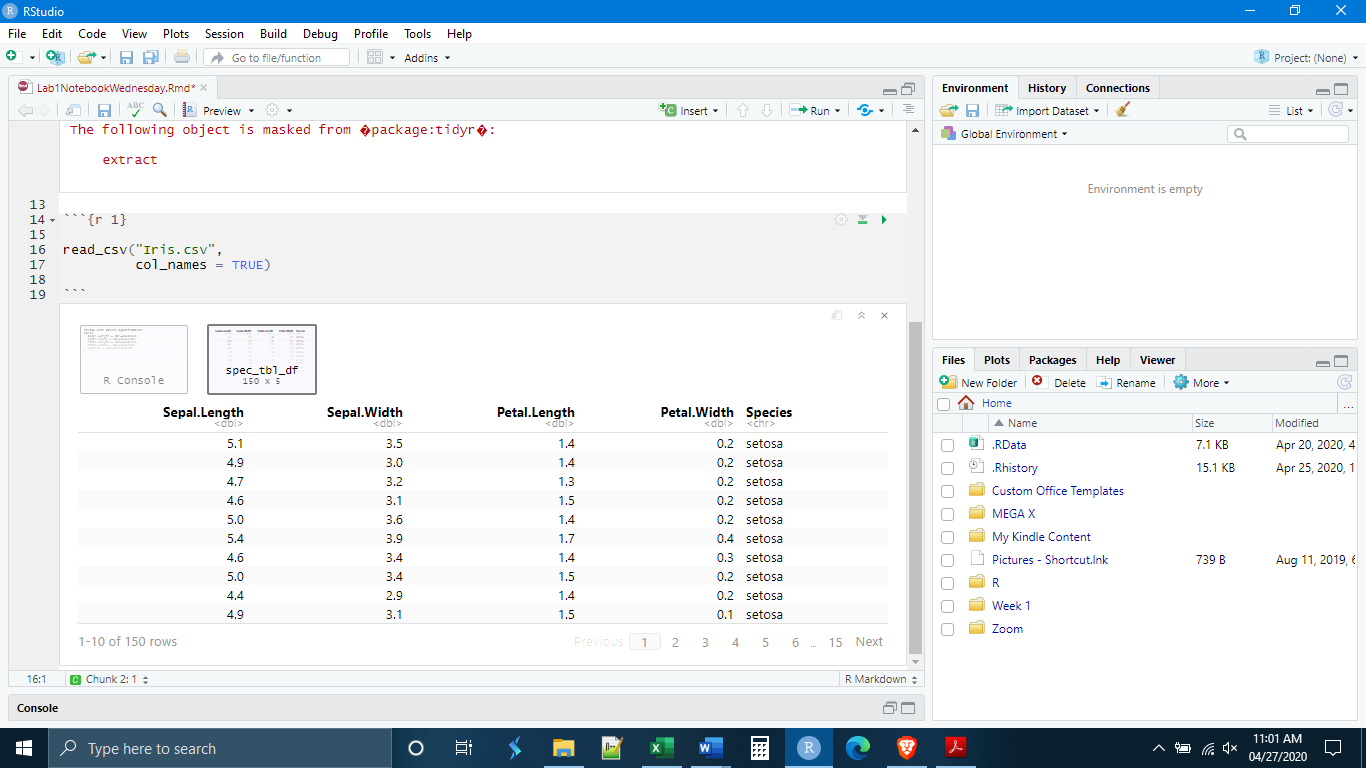
**Data Types, Conversion, and Manipulation; The Pipe, dplyr**

*Read in Data*

Let’s execute the code to read in a new dataset, Iris.csv.



Success! Let’s look at what R has given us. We see that we have five variables which are given in bold. If we look right below these column headings, however, we also see some light gray letters. These indicate the type of data in that column. In this case, we see <dbl> and <chr>.



It is important to know how R stores data because different types of data and different types of storage methods (different types of objects) are capable of different things. Some functions will not work with certain data types or data objects. R has six data types, six ways of classifying what kind of thing a particular piece of information is.

Data Types in R

Logical: TRUE, FALSE, AND, OR, etc.

Numeric: Numbers in the way that we usually understand them, including partial numbers.

Integer: Whole numbers only.

Complex: Numbers but including things like imaginary numbers.

Character: Data involving letters, or letters and numbers together.

Raw: Information stored as raw computer bytes information.

For most biological work, we are interested in Logical, Numeric, Integer, and Character data types. These different data types can be stored in different data objects. We know some things about data objects having created vectors and matrices already. A vector is a one-dimensional collection of data elements. A useful way to think about this is that a vector is a like a column in a table. In contrast to this, a matrix is a two-dimensional collection of data elements, and an array is a multi-dimensional collection of data elements.

Some Object Types in R

Vector: One-dimensional collection of data elements (think of a single column in a table).

List: Similar to a vector but can store other things like functions and other lists.

Matrix: Two-dimensional collection of data elements.

Array: Multi-dimensional collection of data elements.

Factors: These are modified vectors where the data elements represent certain classes.

Data Frames: Think of this like a table, it is a collection of vectors (columns), with each vector being composed of data elements (observations/rows) of a single data type.

Tibble: A tibble is a special king of data frame unique to R.

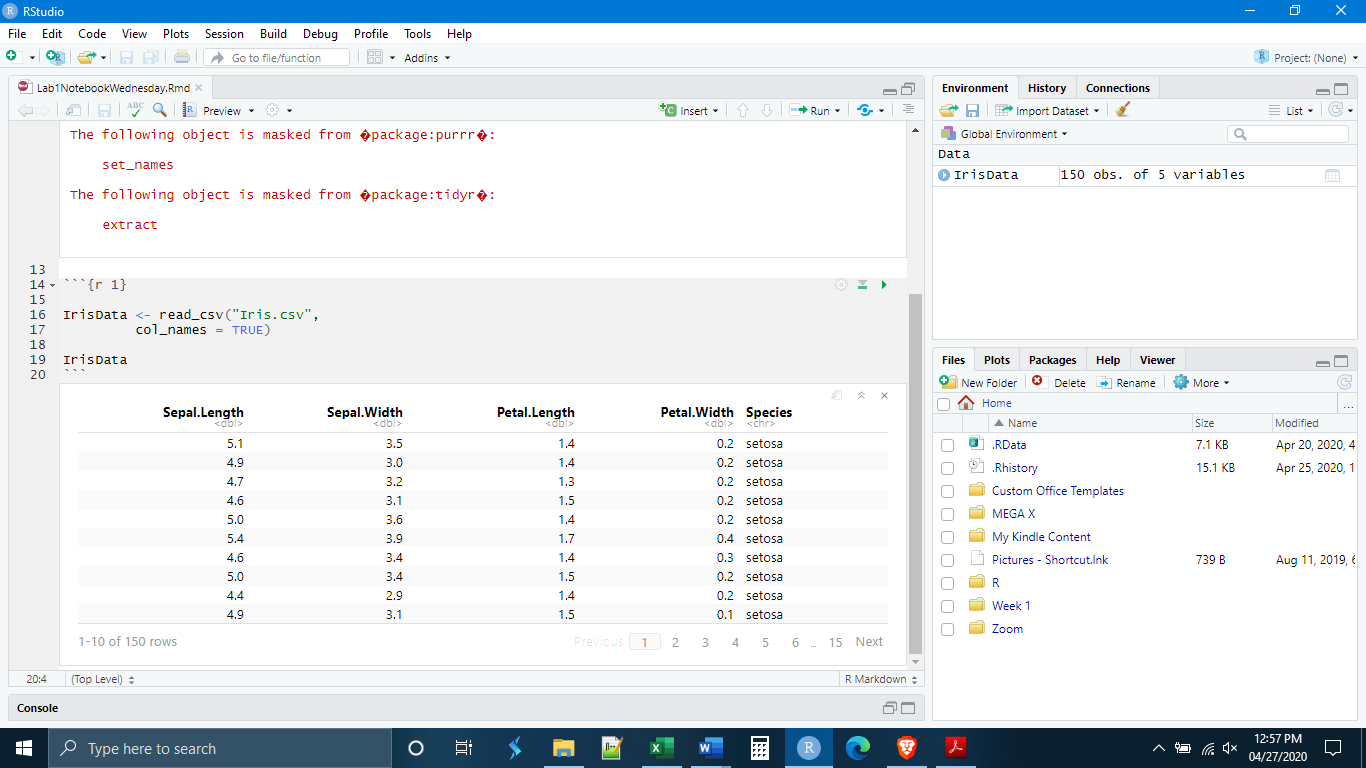
The data objects that you will need to know about are vectors, factors, data frames, and Tibbles. Of these four, the vector is the most important. The other three object types are built out of vectors. Factors are modified vectors, where each of the data elements of the vector belongs to a certain class or category. For example, our vector may be c(A, B, B, D, C, A, B), and if turned into a factor, it would have four categories or classes: A, B, C, and D.

We can think of a data frame as a table of data. Really it is a collection of vectors. Each column is a vector where all of the data elements have the same data type. All of the vectors are related by sharing the same observations (rows). The Tibble is a special kind of data frame that was invented specifically for R and follows certain rules associated with the tidyverse, which makes it work better inside of the tidyverse. The read\_csv() function creates Tibbles, so most of our work will technically be with Tibbles, but it is fine to think of them as data frames.

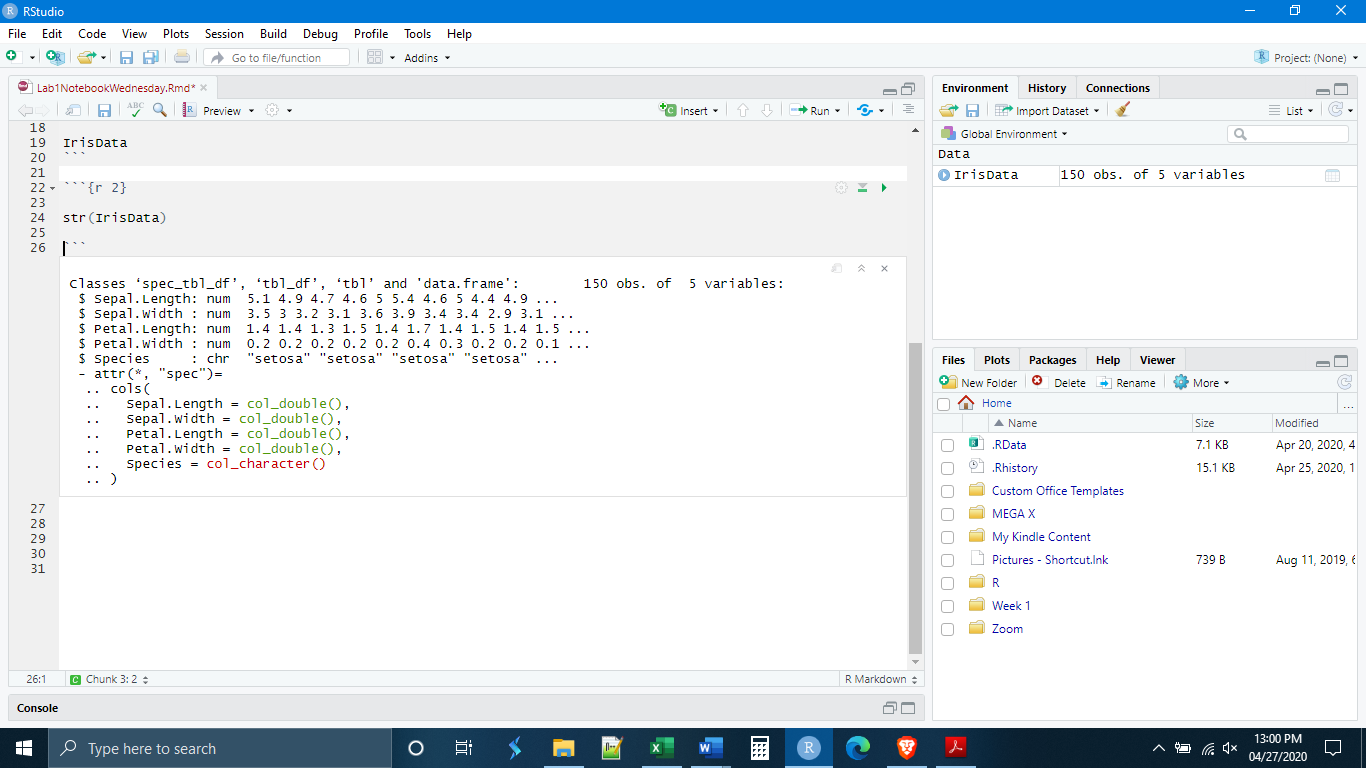
*Explore Data*

Given what was just outlined, what can we say about the Iris data that we have just read-in? Well, we know that the object type is a Tibble, which is a kind of data frame. What about each of the columns, what data type are they? We saw that they were listed under the column headings as <dbl> and <chr>. The <chr> stands for “character”. Data in that column are stored as a character data type. If we look at the data elements this makes sense. The <dbl> stands for “double”, which is the term that R uses for numeric data types. Looking at the data elements in these columns, this also makes sense. Another way to explore our data is through the commands str() and summary().

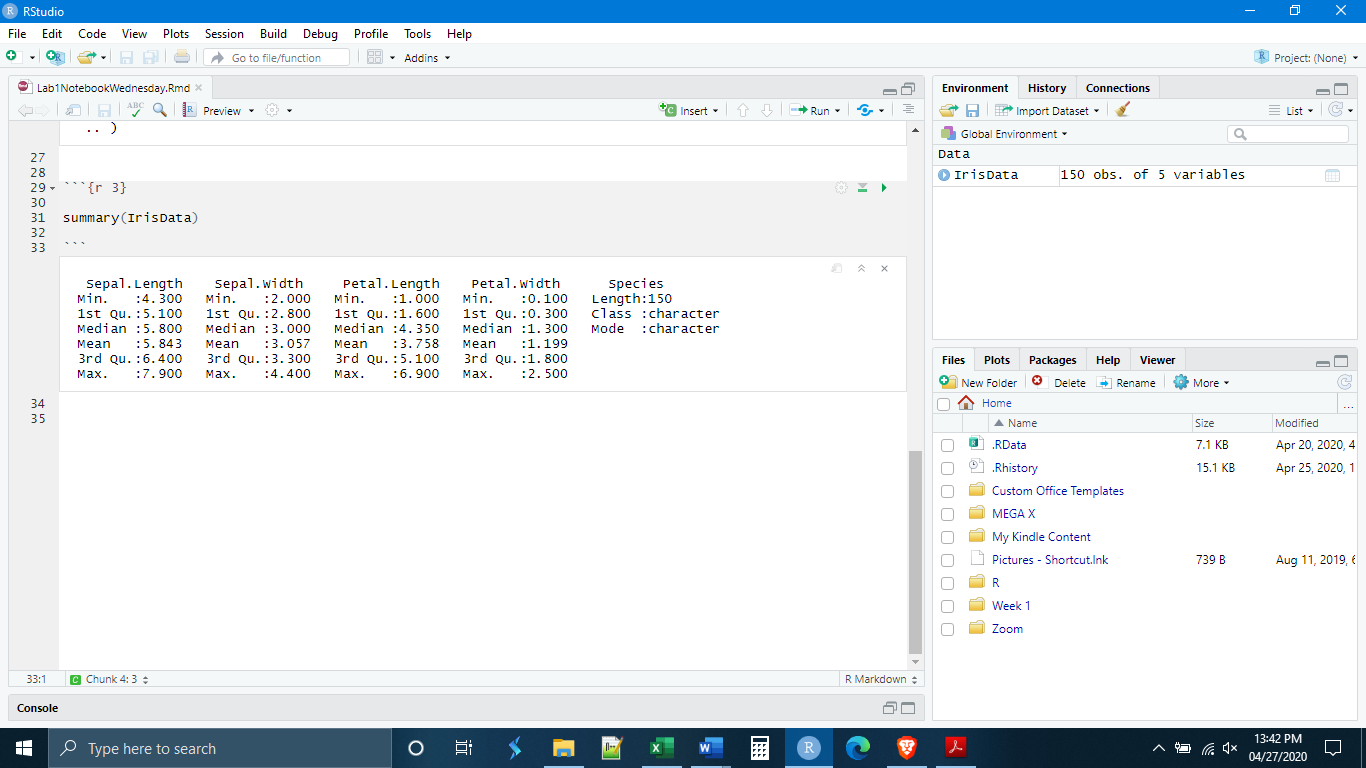
First, we need to rewrite our code to save the imported data as an object (notice it is still not in our environment).



Now, start a new chunk and use the str() command to get information about the data set itself. We get a good amount of information. We see information on the kind of data object that it is, both a Tibble and a data frame. This makes sense because a Tibble is a special kind of data frame. We also see that the data set contains five variables and 150 observations. Then we see two sets of what amounts to the same information. Each of the variables is listed with its name (the column heading) and then what kind of variable it is. As we saw in the table, all of them are classed as numeric (double) except the last one which is a character variable.



In lecture we have been going over what can broadly be called “summary statistics”, basic information about the data such as center and spread. There is an easy way to generate this in R, that will also give us more information about the data set, this is the command summary(). Begin a new chunk and execute the summary() command on the Iris data set.



*The Pipe Operator*

Up to this time, we have written code as single lines, each line doing one or two operations or functions. But sometimes we want to do more than one function, or we want to take the output of one function and put it directly into another function. To do this in the way that we have been writing code, we would need to save each intermediary step as a new data object in order to enter it into the next function. In even a moderately complex analysis this would become tedious very quickly.

The pipe operator comes to rescue us from this problem. The function of the pipe operator is to take the output of one function and put that output directly into another function. In R, the pipe operator is written as %>%. And it is always written at the end of a line of code, indicating that the output or the result of that line of code should be fed into the next line of code. Because the pipe operator specifies what the data are going to be, you need not enter the data explicitly (there will be exceptions to this, but we will go over them as they arise).

Let’s do a simple example. How could we write the code to get the summary statistics using the pipe operator? (see Below) Notice that, when we hit the return key the next line is indented. This is a customary way of writing code which indicates that this second line is really part of the code that comes above it. R Studio knows this and automatically indents the code for you.



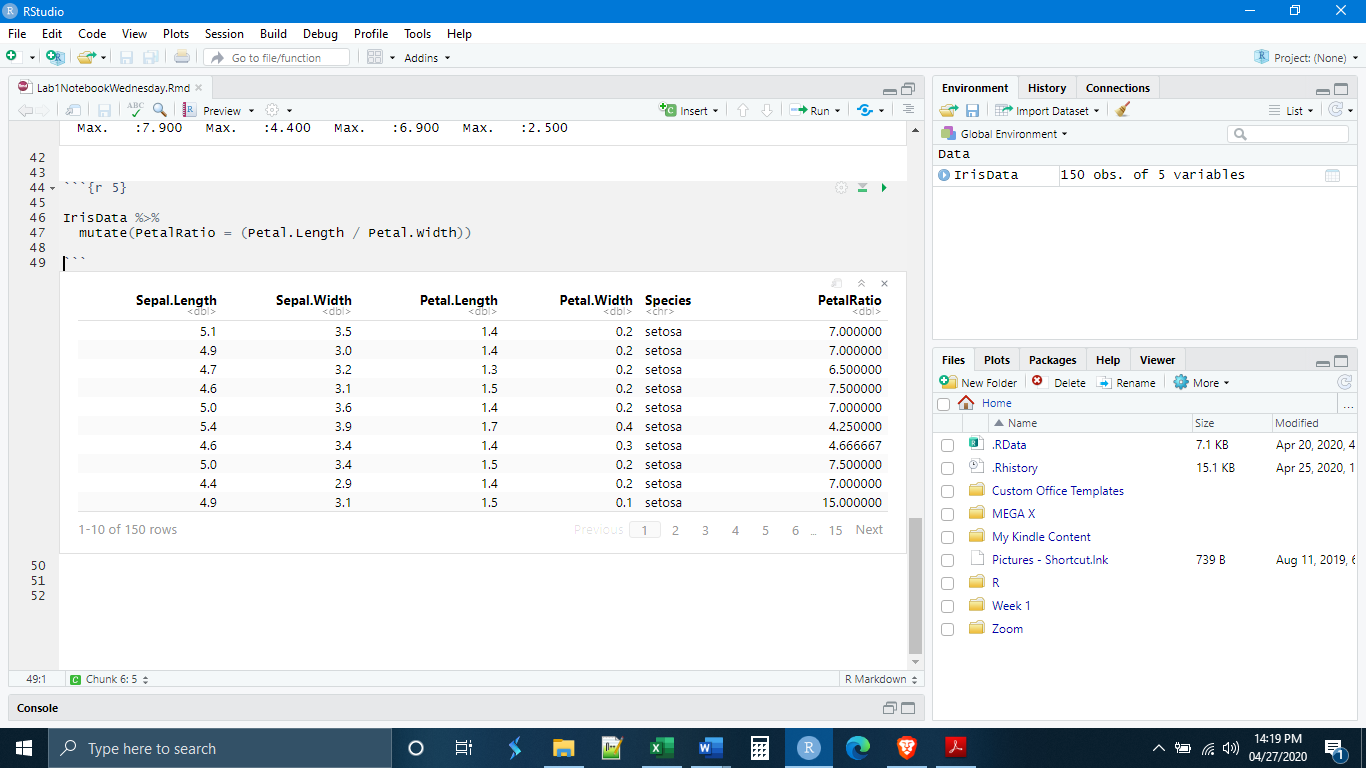
IMPORTANT: As we have seen with other functions, operations done through the pipe operator do not change the original data set unless we create a new data object.

*Manipulate Data Types*

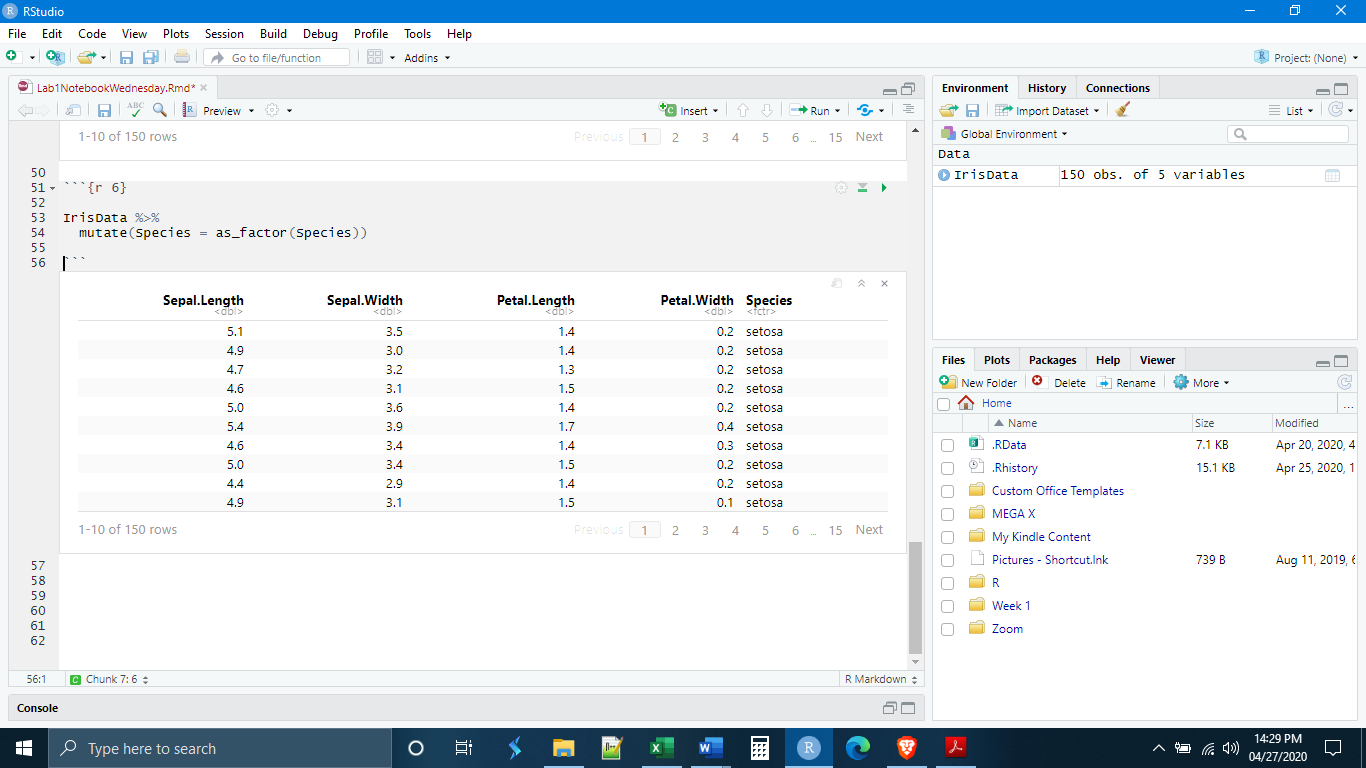
Next we need to learn some of the basics of manipulating data in R. The section of the tidyverse that deals with the manipulation of Tibbles is dplyr. We will learn more about dplyr next week, but this will serve as a very brief introduction. Today we will learn about the dplyr functions mutate() and rename(). mutate() creates new vectors (columns) in a Tibble. The syntax for mutate() is:

mutate(NewVectorName = ContentsOfNewVector)

We can do a lot with mutate and it is perhaps best to simply show some examples. Our Iris data set includes the variables flowers petal length and flower petal width. What if we wanted to make a new column that was the ratio of petal length : petal width? This could be done with the code below. Notice that, because we piped the data into the mutate function, R already knows what vectors (columns) exist in that data set, and we do not need to specify it a second time.

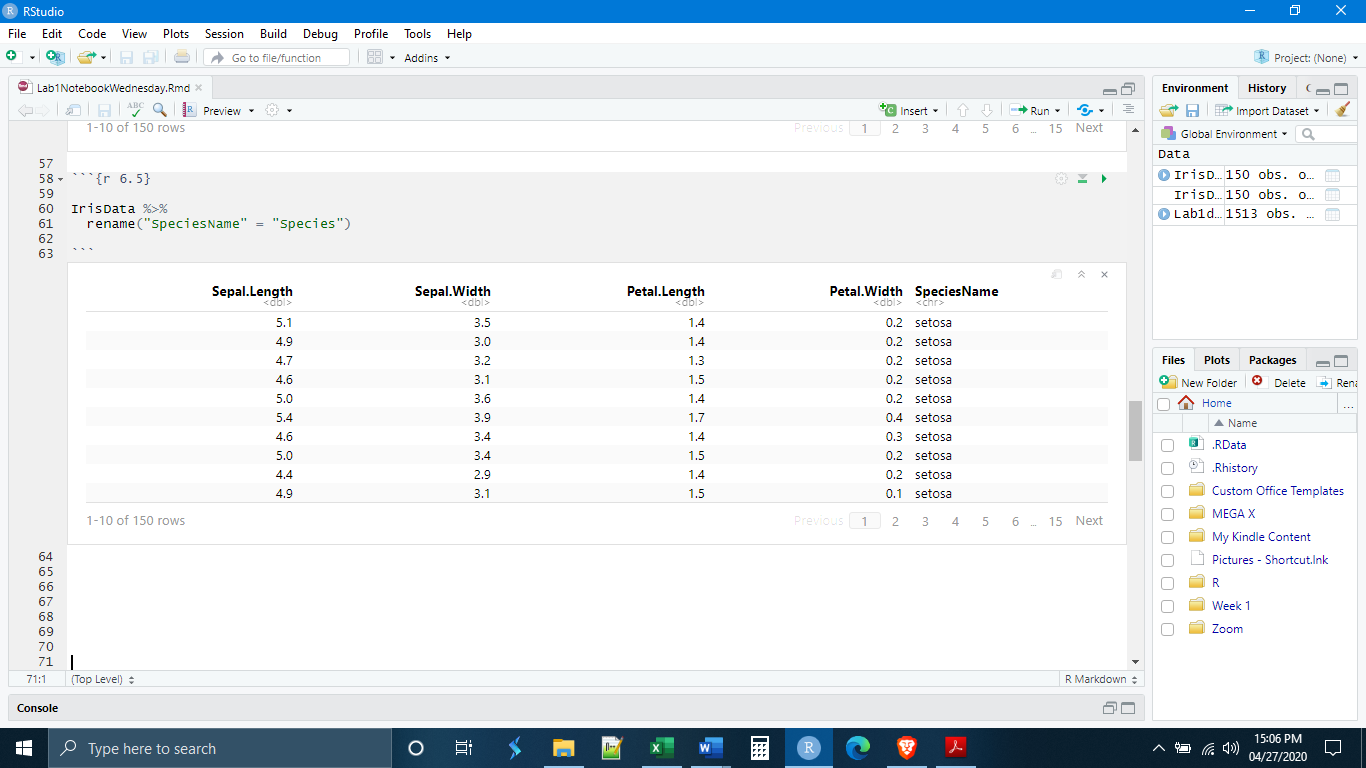


A second example. The fifth column of our Iris data set has a variable named “species”. R interpreted this variable as a character data type (also known as “string” data). But really those strings represent different categories (the different species). We can use mutate to change the data type to reflect this by using the function as\_factor(). (See Below) Now that column is labeled as <fctr> data type. Really, the Species vector in our Tibble is now a factor object. Sometimes R will not recognize numbers as numbers and will code a vector (column) of numbers as characters or factors. A function that can undo this is as\_numeric().

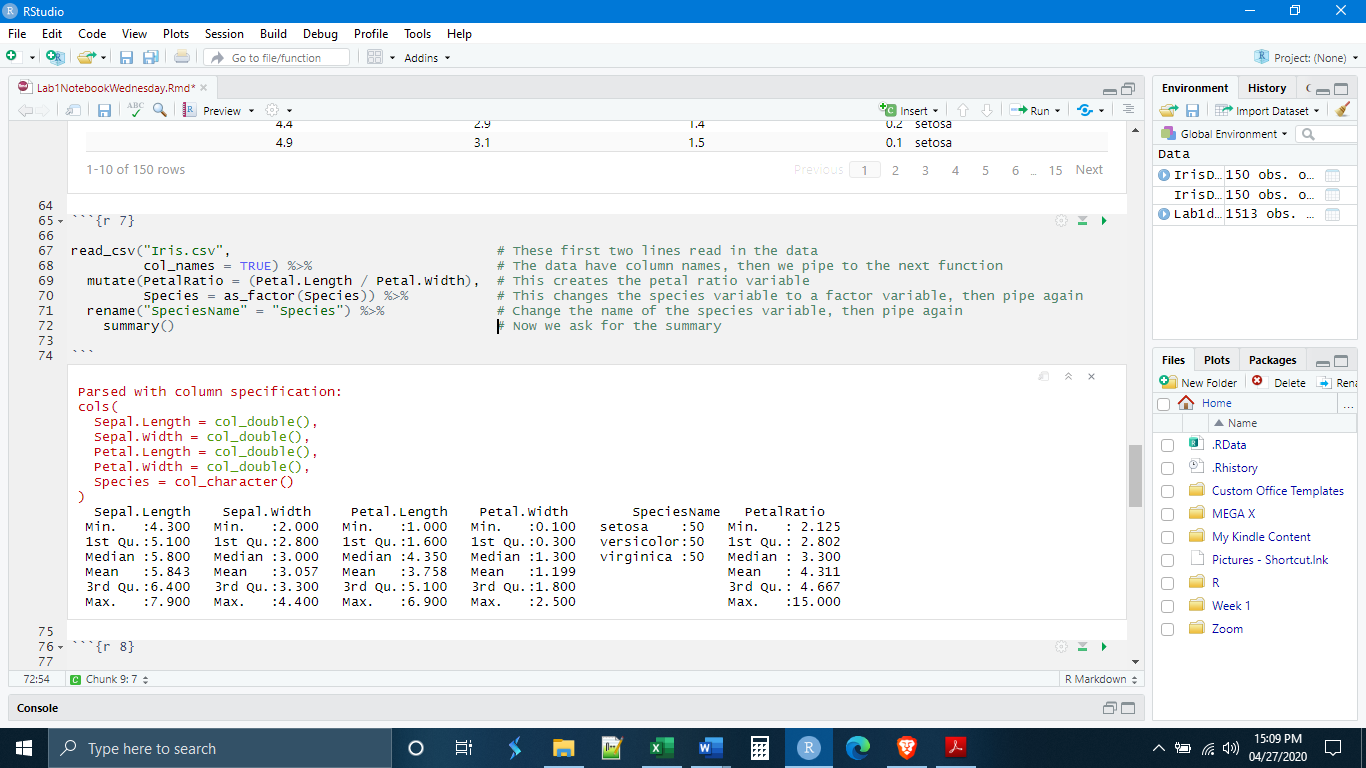


It is also possible to rename variables using the rename() function, which uses the syntax:

rename(“NewName” = “CurrentName”).

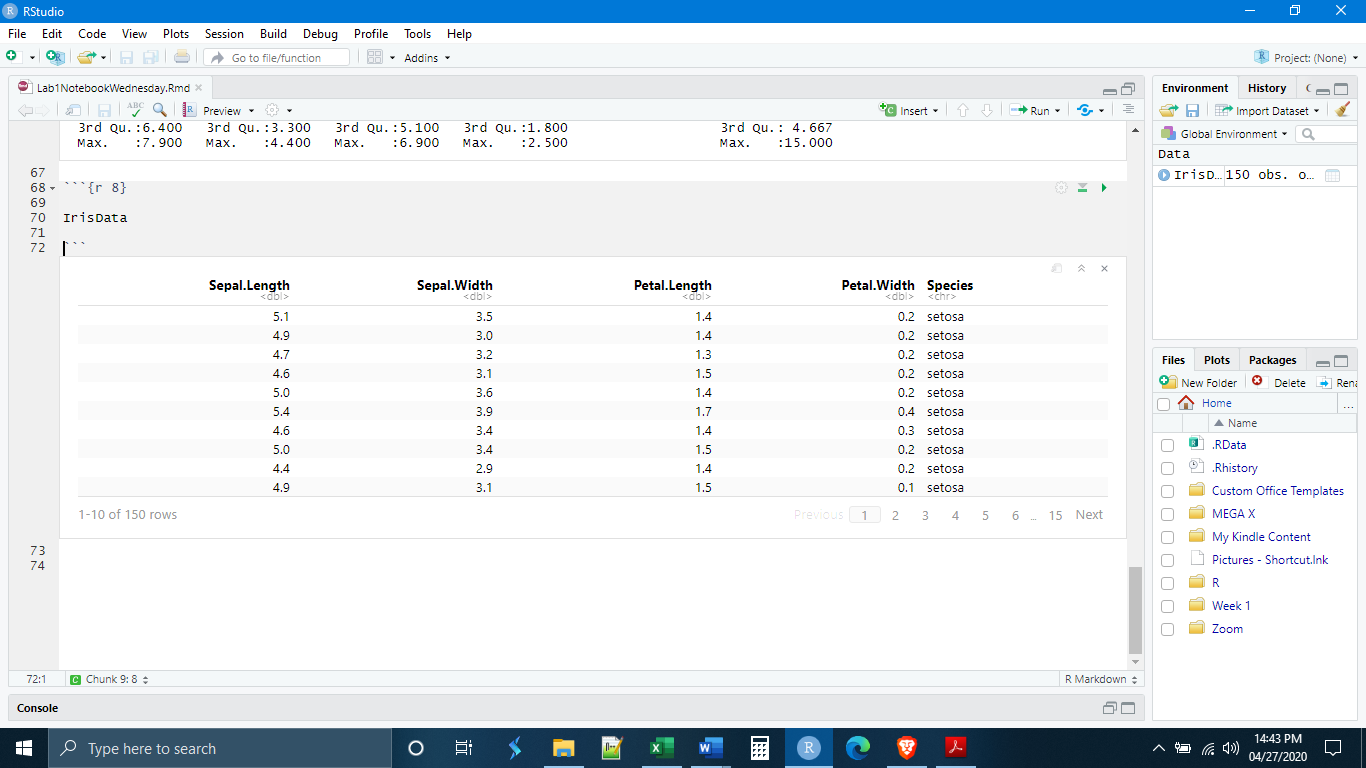


Let’s write our first dplyr pipe. All of the things that we have done can be accomplished in one piped series of commands. (See Below) Notice, we can do as many mutations as we like, as long as we separate them by a comma. I put them on separate lines because it is easier to read.

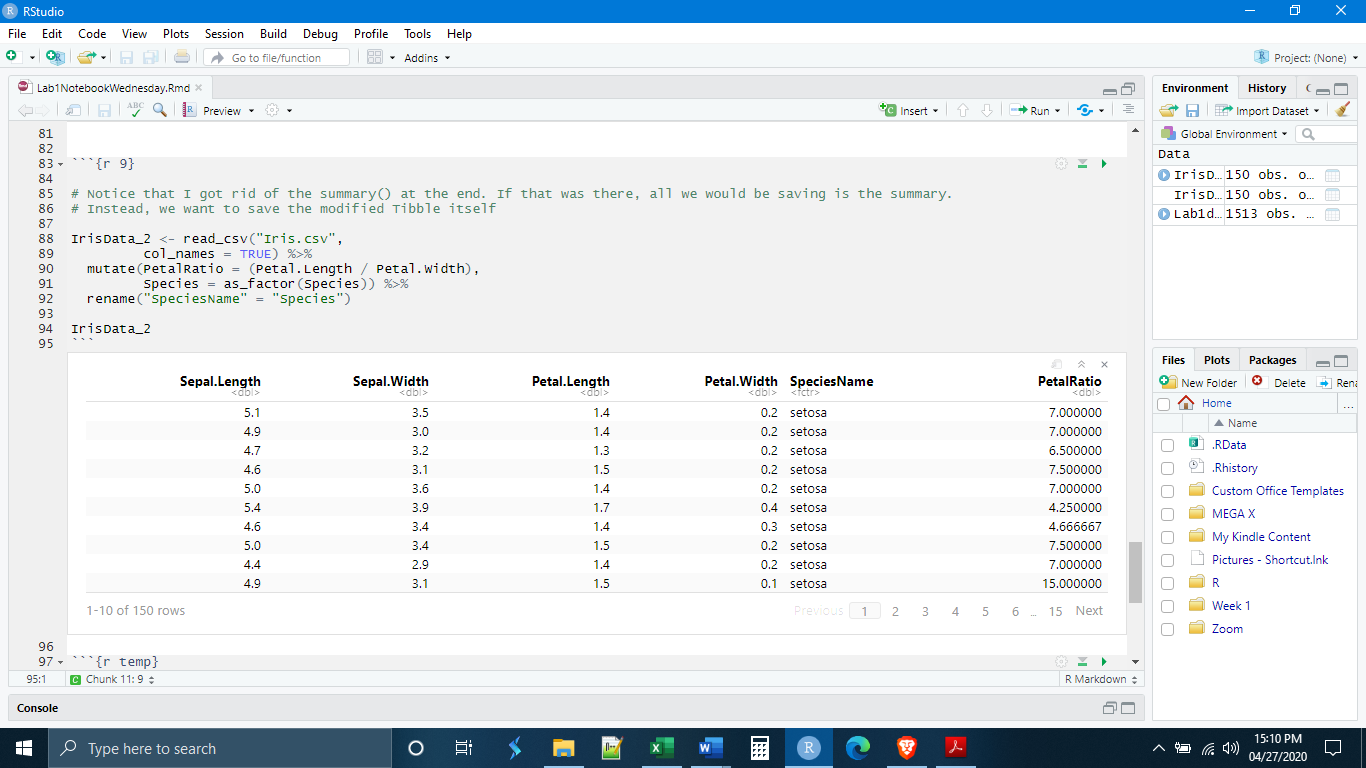


Our summary is different from the first time that we did it. Notice that it now lists all of the factor “levels” in our Species variable, in this case the three species in the data set. We also have information on the new variable that we created about petal length : width ratios.

What happens if we recall the original IrisData object? They are unchanged. Why? We did not create a new data object with the changes.



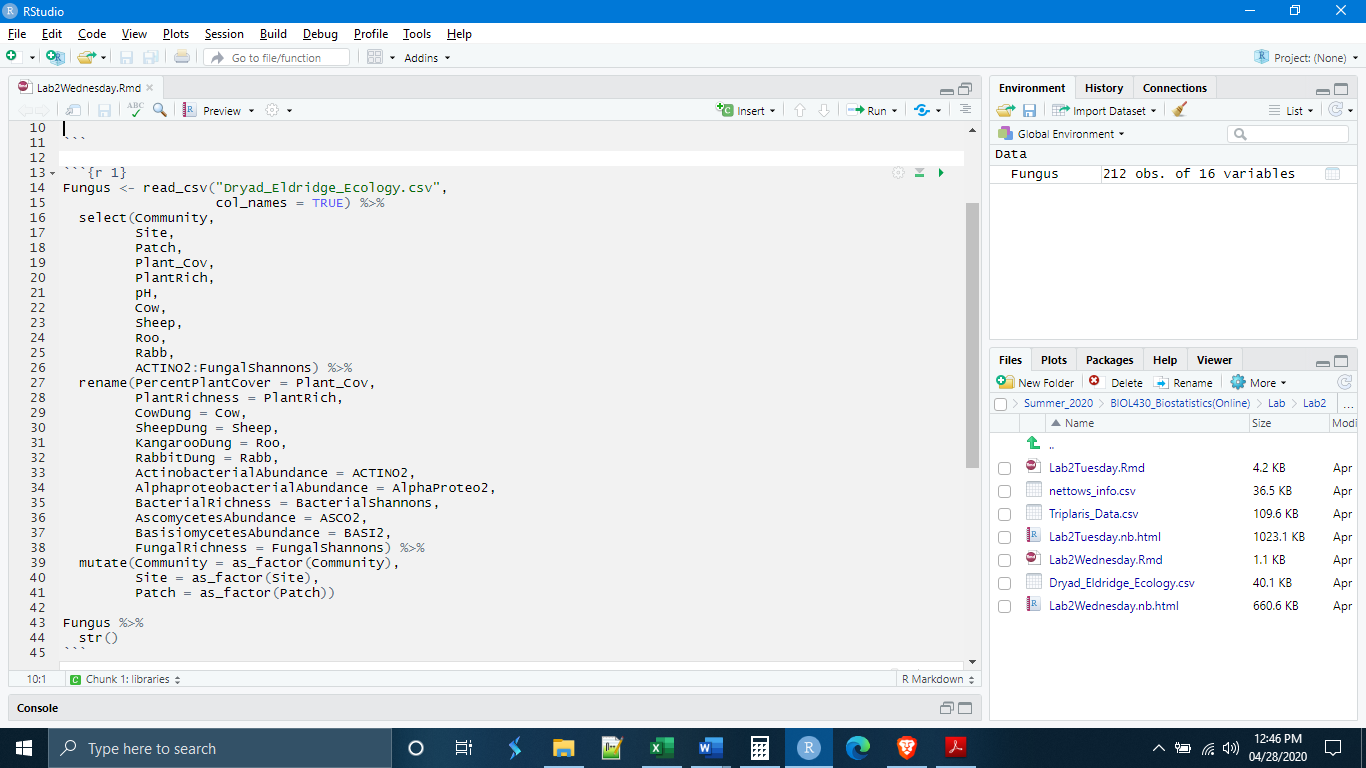
If we want the changes to stick, we need to create a new data object.



**Visualization, ggplot2 (Boxplots, Histograms), Colors**

*Data Reading and Cleaning in dplyr*

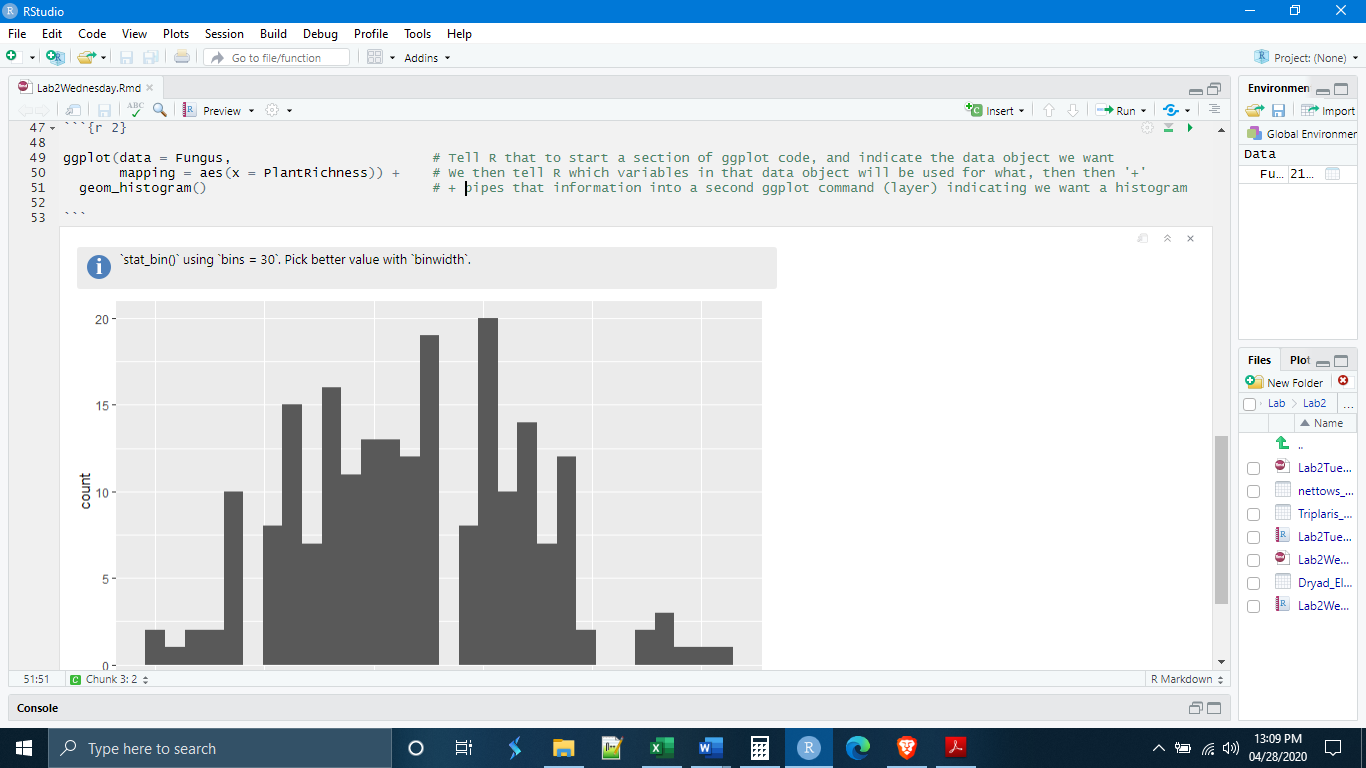
The first thing that we will need to do is read in some new data and clean it. These data come from a study conducted in Australia. Here, the researchers were looking at changes in soil microbial community composition as a result of animal grazing.



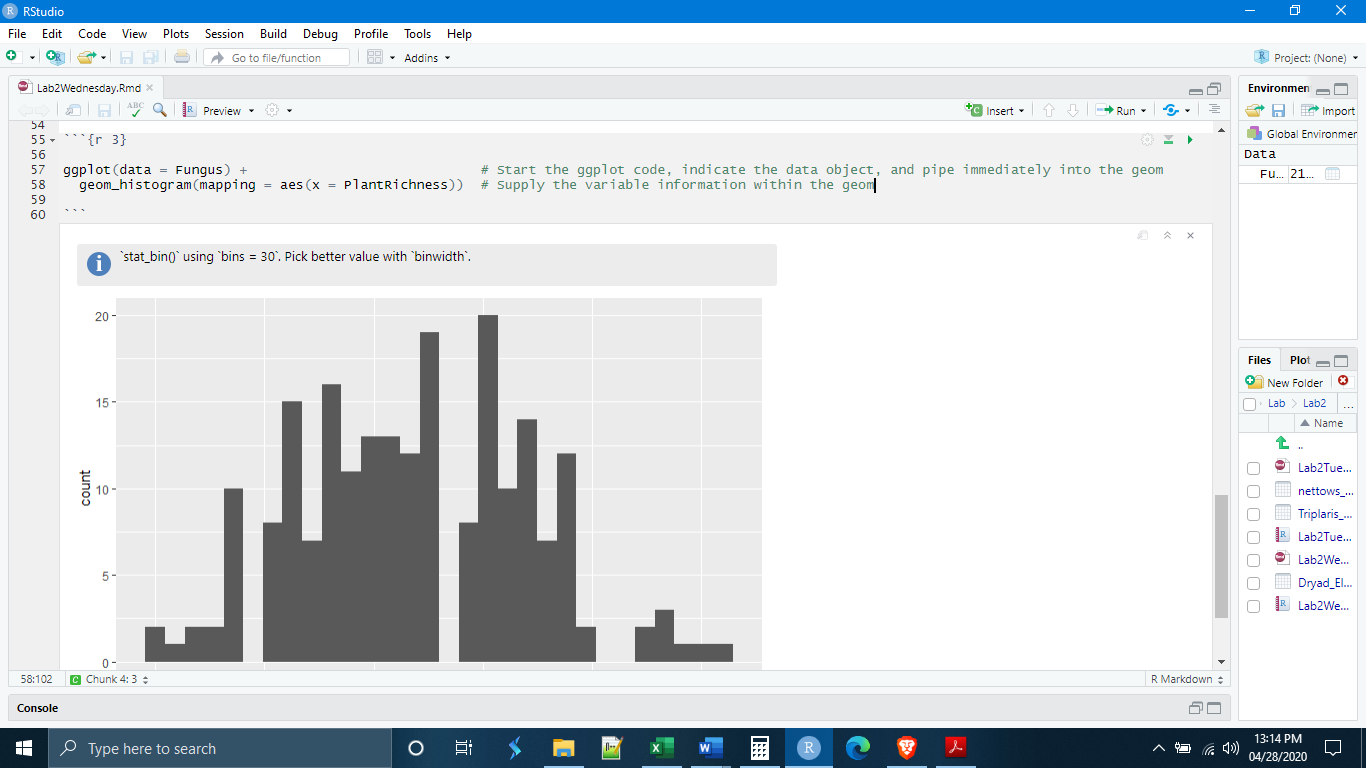
*ggplot Basics*

ggplot2 uses a slightly different syntax than dplyr. In place of the %>% that is familiar to us, ggplot uses +. Second, ggplot functions through layers. Each time you add a new term to your ggplot code, it places a new layer on top of your visualization. This will make more sense in a few minutes when we start to code, but I mention it because sometimes the order of layers matters.

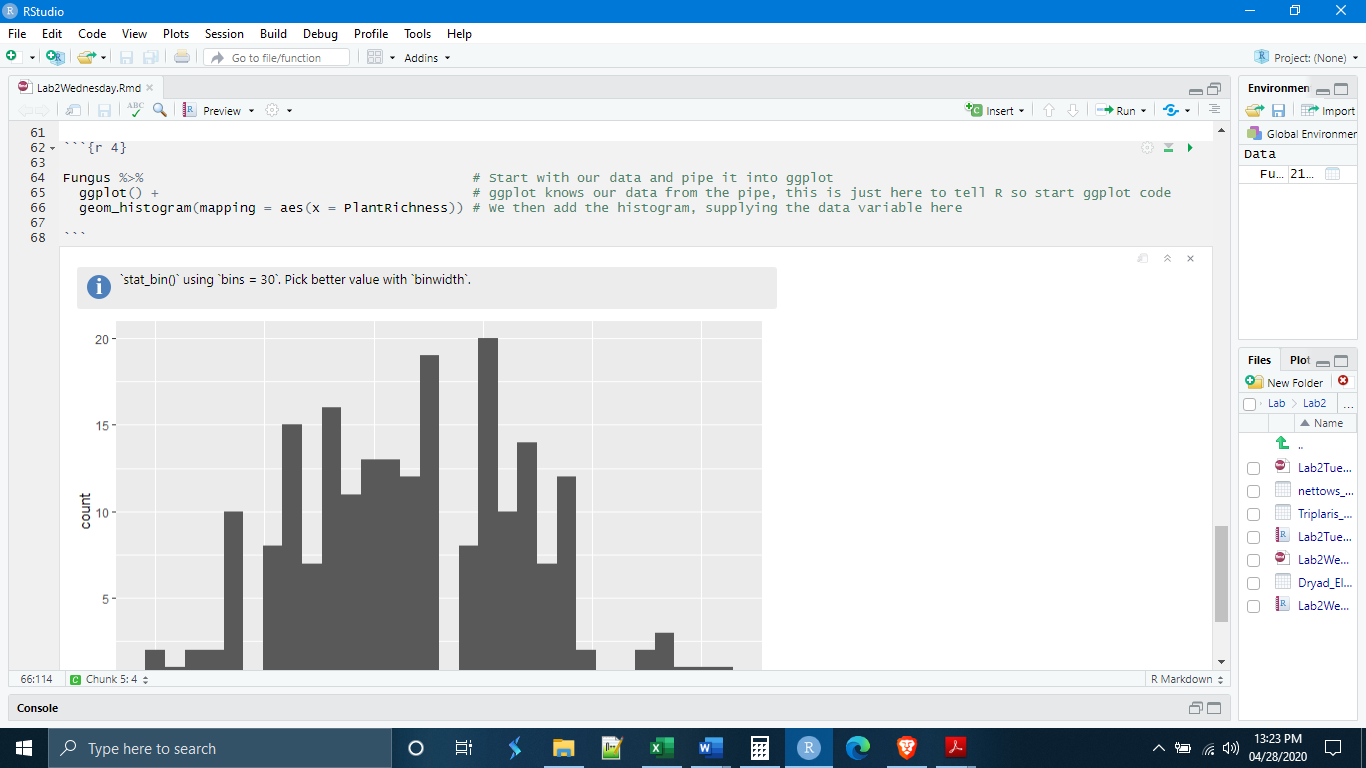
Let’s say we want to make a histogram of the species richness values. If, for a moment, we forget what we have learned from our dplyr labs, we could write the code like this



It is also possible to supply the information about variables later, within the geom\_histogram() command.

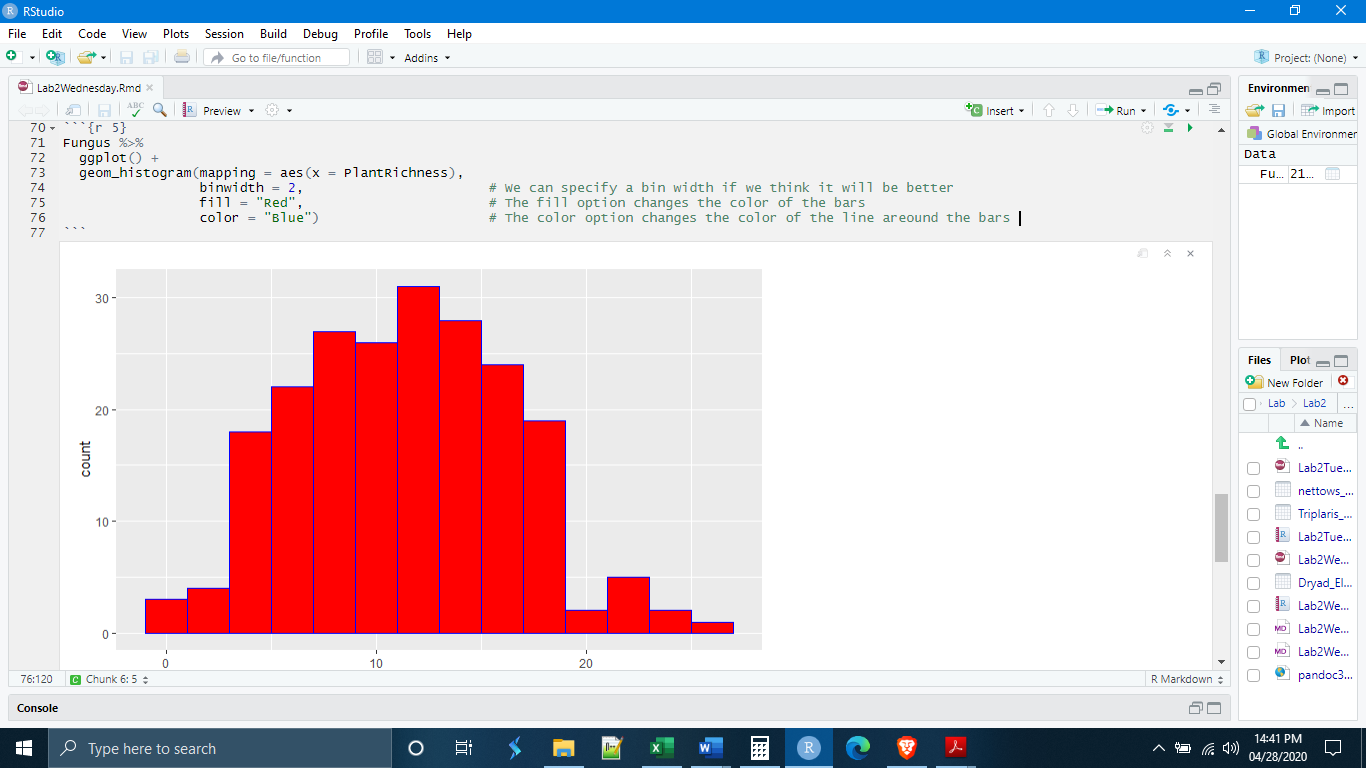


The final way is to use the dplyr pipe to pipe the data into ggplot. This is the preferred method because it allows you to make modifications to the data necessary for plotting, without needing to create a separate data object. Instead, you run a short dplyr pipe from the main data object before beginning ggplot, and pipe the modified data directly into ggplot.

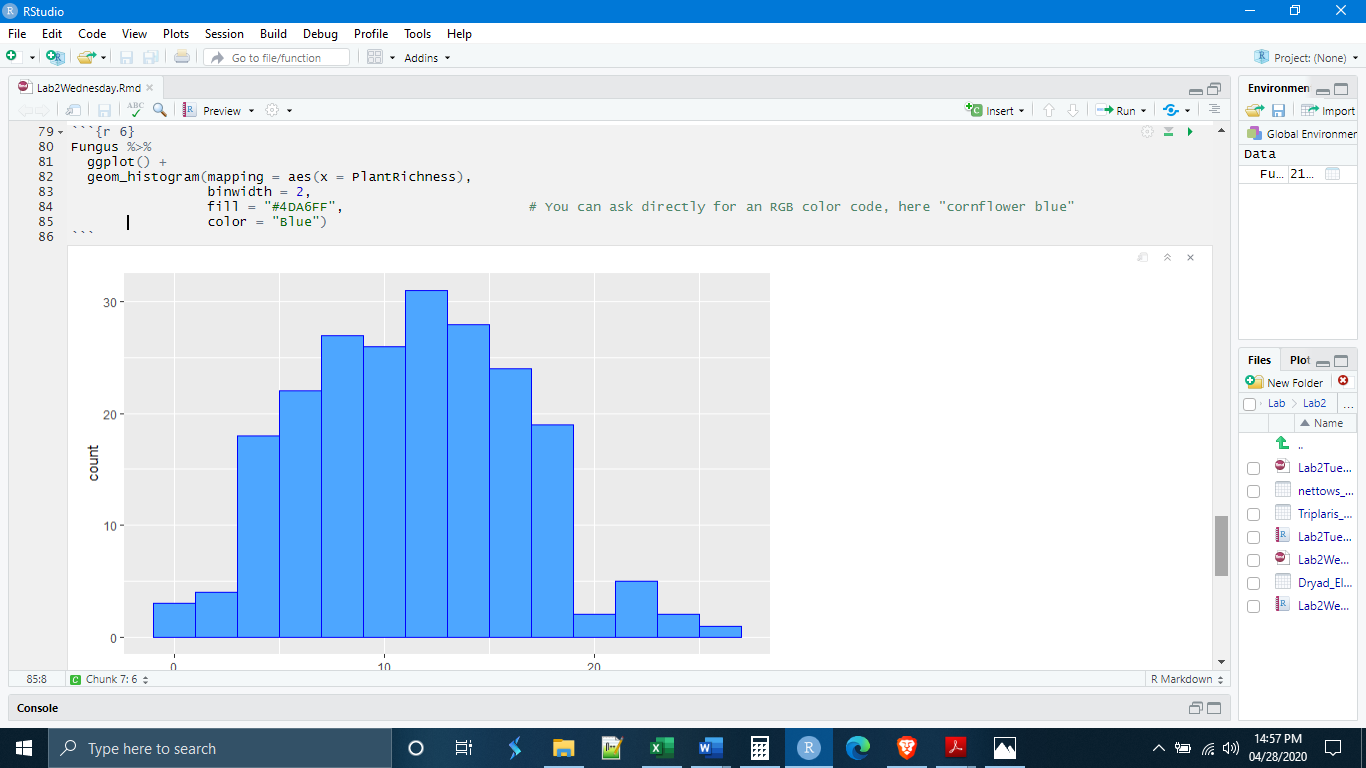


*Histograms*

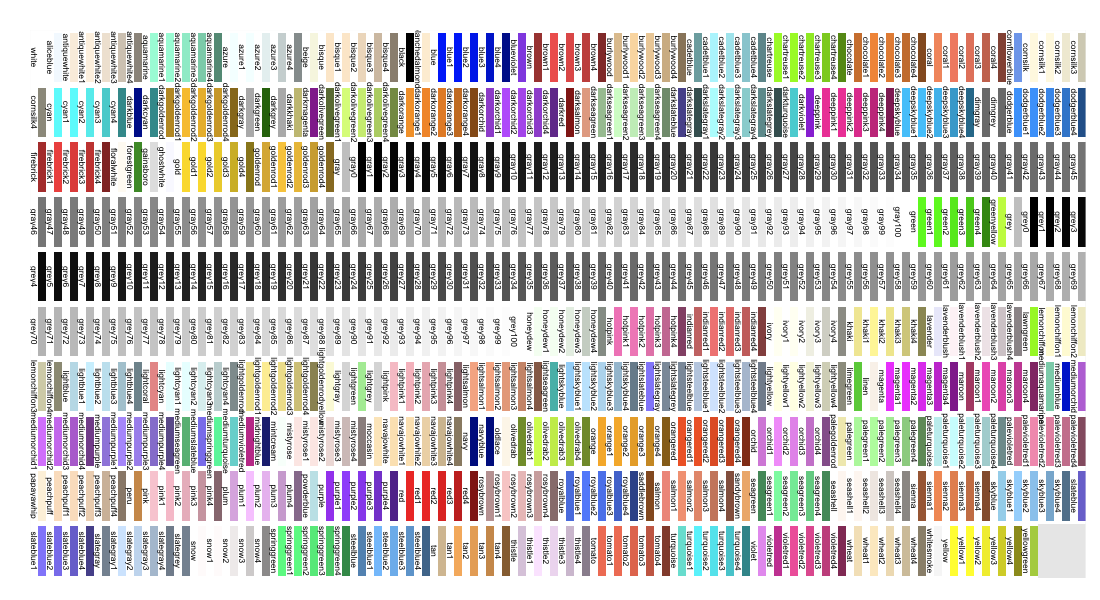
The histogram takes a single variable and plots the observation values on the x axis and the frequency of observation values on the y axis. The histogram helps to visualize the data and determine its distribution. The histogram ‘bins’ observation values, particularly if they do not fall into natural categories. As a default, geom\_histogram() will automatically choose the bin size. We can manually select the bin size, as well as the colors, with the code below.



ggplot has a good number of pre-loaded colors in its palette. It will accept any RGB color code (ex. cornflower blue = #4DA6FF).



It is also possible to call certain colors by name so that you don’t need to look up the code number. We have already done this with basic colors like “red” or “blue”, but there are quite a few options.



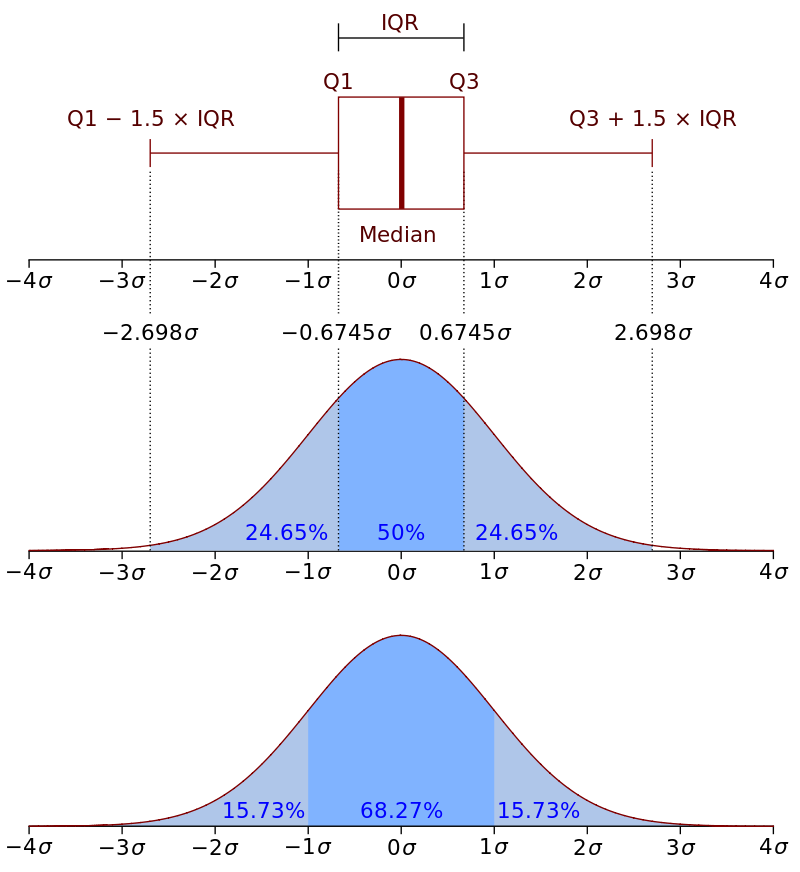
(http://sape.inf.usi.ch/quick-reference/ggplot2/colour)

*The geoms*

The ggplot command for a histogram is geom\_histogram(). The geom\_...() is a standard vocabulary for the different kinds of coordinate grid objects and plots. There are many of them. The ggplot cheat sheet will be your friend. It lists a large number of the geoms and their uses. There are such things as geom\_histogram(), geom\_bar(), geom\_point(), geom\_boxplot(), as well as things like line objects such as geom\_abline() and geographic map functions such as geom\_map().

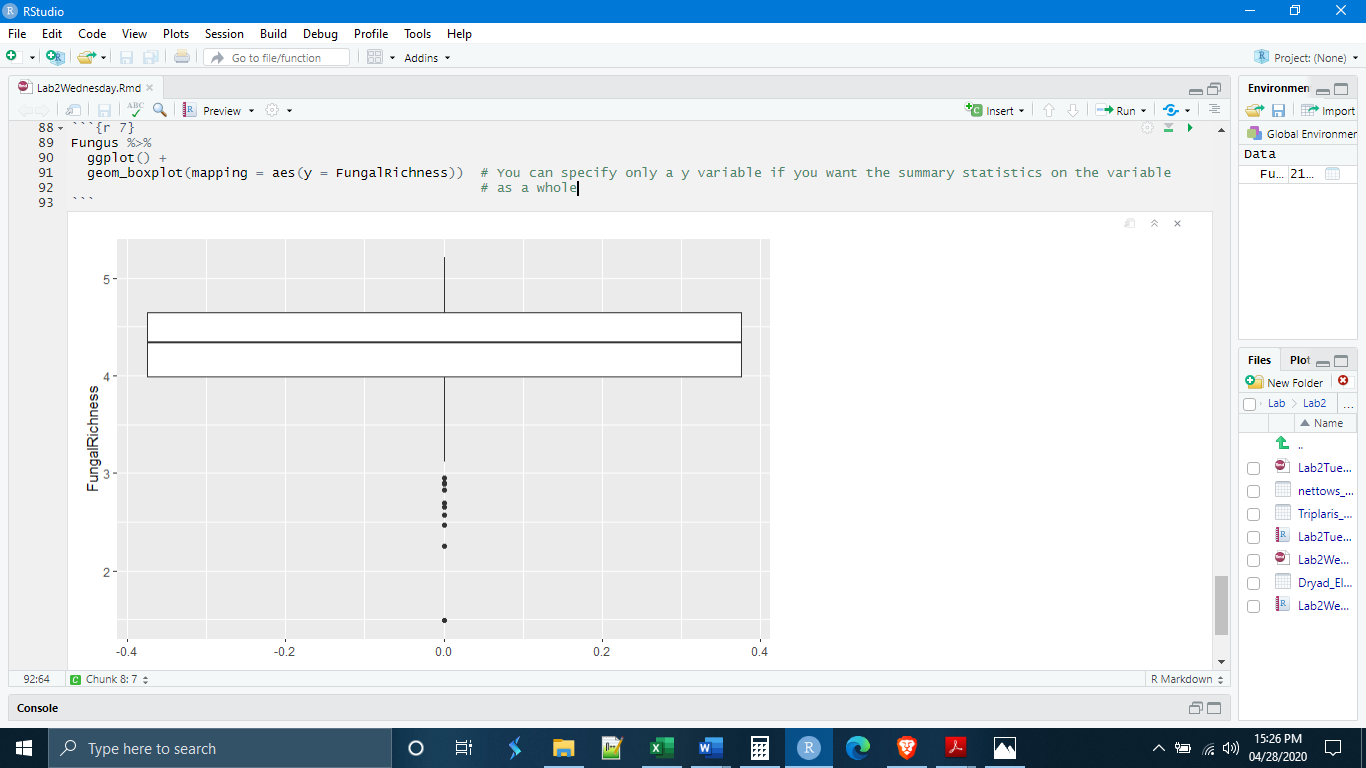
*Boxplots*

A boxplot (also sometimes called a box and whisker plot) is another plot designed to investigate the distribution of a data set. The boxplot provides quantitative information about the center and spread of the data that the histogram does not. The boxplot shows the median, the middle quartiles, and “whiskers” that represent the major part of the tails. If the distribution of observations is normal, fifty percent of the observations will fall inside of the “box”, most of the remaining fifty percent of the observations in the “whiskers”, and only observations in percentiles less than 1 and greater than 99 will fall outside of this range.

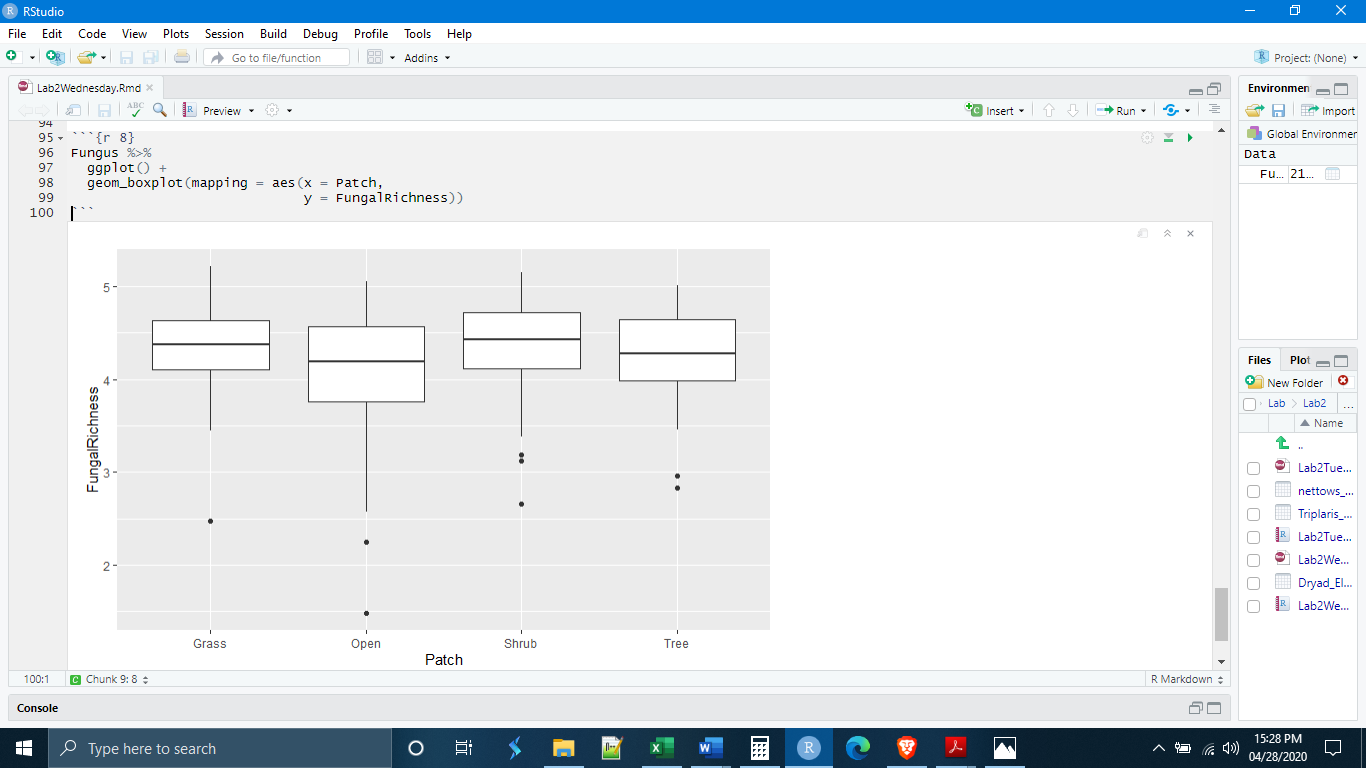


(https://upload.wikimedia.org/wikipedia/commons/1/1a/Boxplot\_vs\_PDF.svg)

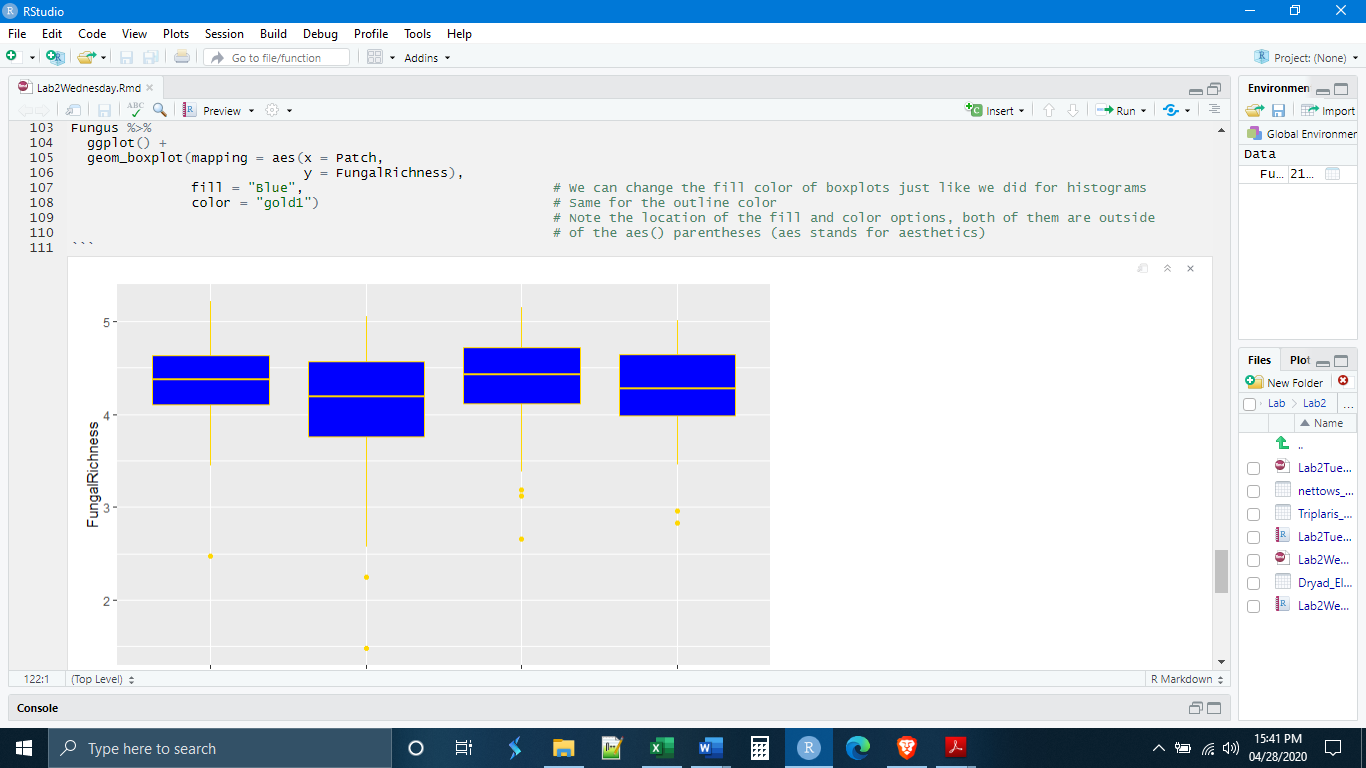
For a basic boxplot, the code in ggplot will look very similar to that of a histogram. A boxplot can be made for a single variable to investigate the center and spread of that variable or group of observations. This can be accomplished in ggplot by specifying only a y variable. ggplot will fill in the x axis with cartesian coordinates.



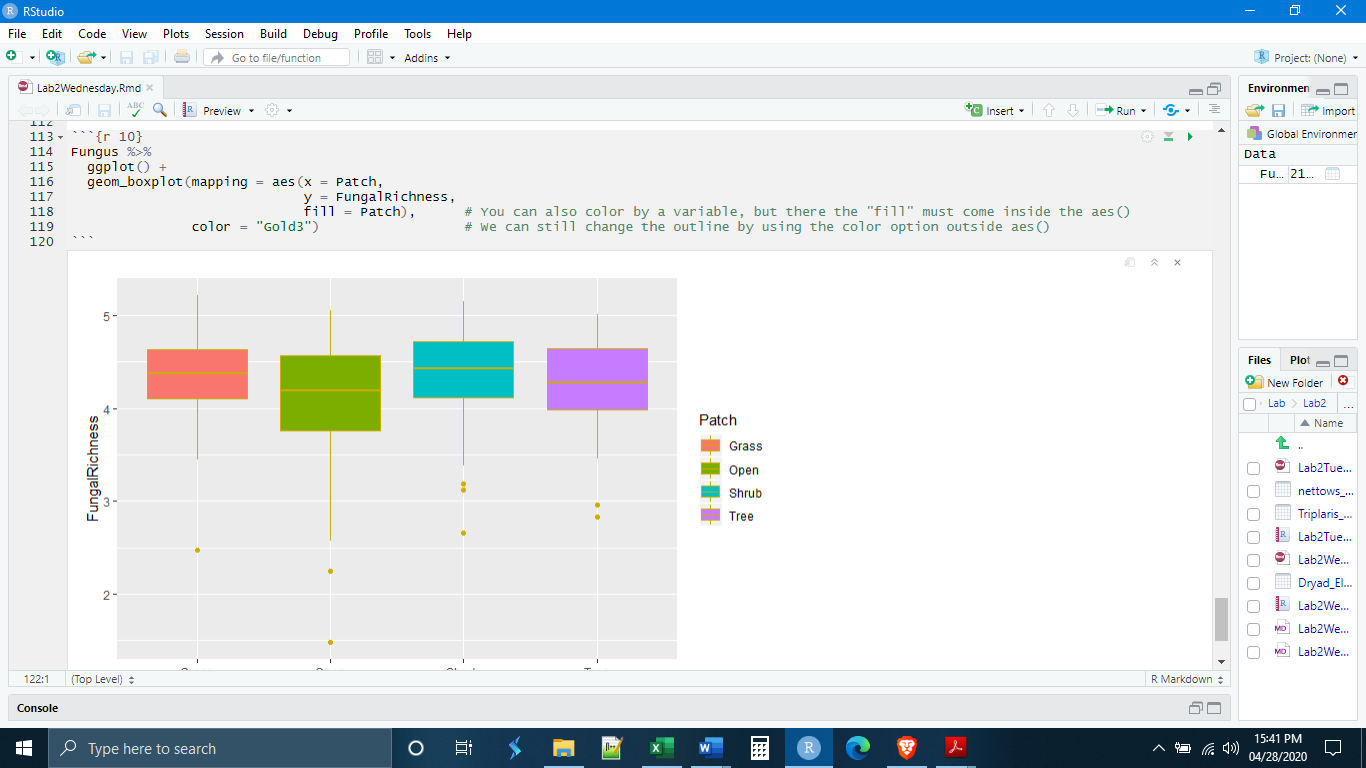
It is also useful, however, to compare the center and spread of multiple groups of data. For example, in today’s data, one of the variables was the patch types where fungal diversities were measured. We can use the patch type as the x variable to get a boxplot for each patch type.



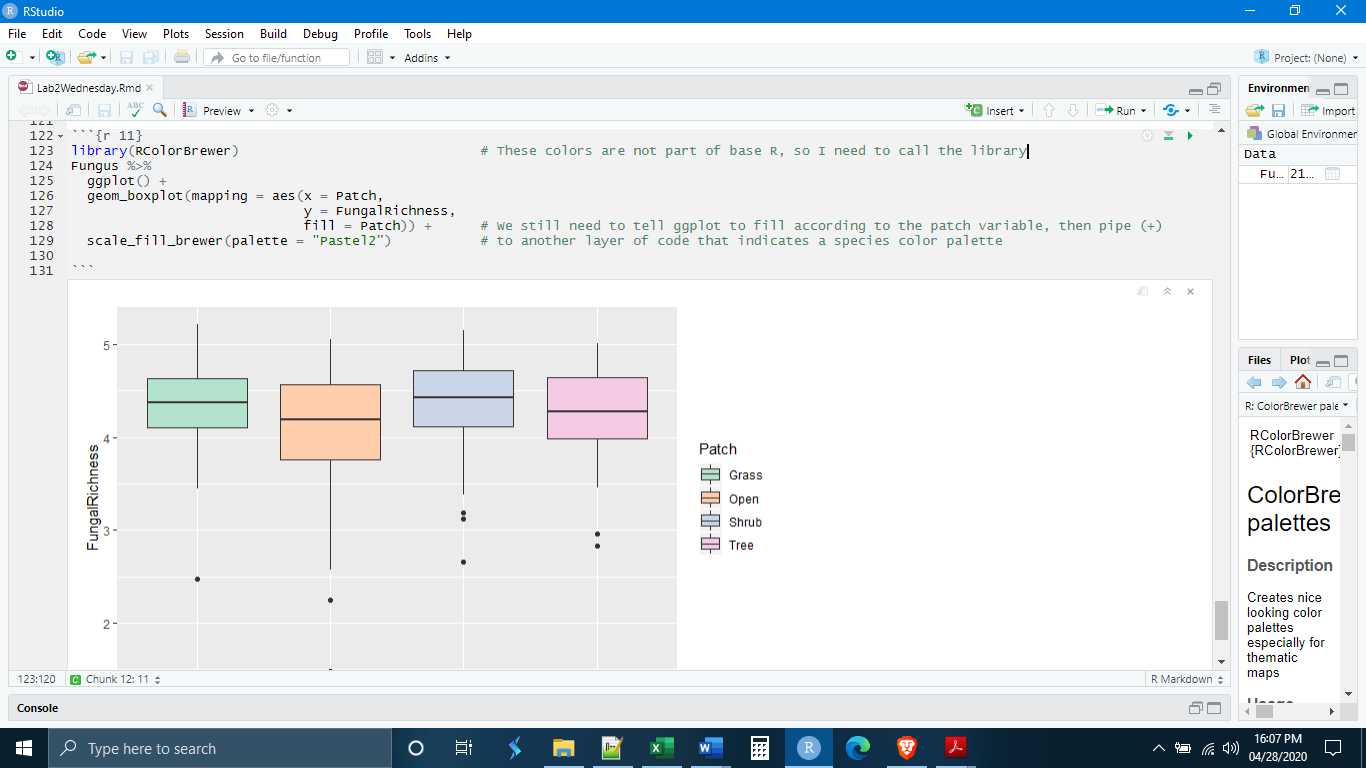
Changing general colors in boxplots can be accomplished in the same way as with the histogram. One thing to notice is where we are placing the fill and color options; they are outside the aes() parentheses. Also, the colors are placed inside quotation marks. If you leave out the quotation marks, R will think you are referencing a particular data object.



It is also possible to color the boxplots according to the observations in a particular variable. For example, if we wanted to highlight the patch types, we could stipulate a fill according to the Patch variable itself. In this case, ggplot will give each of the patch types its own color. Note, however, that this fill option goes inside the aes() parentheses.



Finally, when ggplot colors according to a variable, it selects from one of its default color palettes (collections of colors intended to look good together) depending on how many colors it needs (in our case, it would need four colors). We can manually change this color palette by adding another layer to our ggplot code specifying a special color palette. Here I am selecting from the RColorBrewer, which is a collection of color palettes that can be downloaded. Notice that I had to call a new library before the code to remind R that is had RColorBrewer installed.



*Individual Exercises*

**1.** Make a new chunk and write the code that will import the ToothGrowth data set (found on Moodle), and save it as a new data object. (Hint: make sure you load the proper libraries, if necessary).

**2.** Make a new chunk and write the code that will import lab1data2.csv (found on Moodle), and save it as a new data object. (Hint: do the data need/have column names? What does that mean for col\_names = ?)

**3.** Make a new chunk and write code that reads in the Lab1data2.csv and uses the str() function to determine the number of variables (columns) in the Tibble and the number of observations (rows). How many are there of each? Record the answer as a comment in the chunk.

**4.** Make a new chunk and write code that reads in the ToothGrowth.csv data and mutates the “supp” variable to a factor variable.

**5.** Make a new chunk and write code that reads in the ToothGrowth.csv data, mutates the “supp” variable to a factor variable, and then gets a summary of the data. How many “levels” are in the factor variable “supp”? What are the names of the levels? Record the answers as a comment in the chunk.

**6.** Make a new chunk and, without saving any new data objects, write a single dplyr pipe that:

- Reads in the data set Lab1data2.csv

- Mutates variables X2 and X3 to factor variables

- Renames variables X1 to “ID Number”, X2 to “Class”, and X3 to “Model”

- Summarizes the data

In the factor variable “Model”, what are the names of the factor “levels”? How many observations are in each level? Record the answers as a comment in the chunk.

**7.** Make a new chunck and, using the Fungus data we made today (you may need to copy and rerun the cleaning/import code from the beginning of today), write code to make a histogram of the pH variable. Set the bin width to 0.1.

**8.** Make a new chunk and, using the Fugus data, write code that first filters out any PercentPlantCover above 25, and then pipes this data directly into ggplot to make boxplots of the PercentPlantCover variable in each of the patch types. Color the boxplots based on the patch types.

**9.** Make a new chunk called “chunk 8” and paste in the code below. You have seen that I have sometimes put in comments to indicate what each of the lines of code is doing. I want you to add comments for each line of this code indicating what that line is doing. Feel free to run the code first to see what happens.

Fungus %>%

group\_by(Community,

Site) %>%

summarize(SiteKangDung = mean(KangarooDung)) %>%

ggplot() +

geom\_boxplot(mapping = aes(x = Community,

y = SiteKangDung,

fill = Community))