A Beginner's Guide to R and RStudio

tutorial written by Anna Chase.

All scripts and data for this tutorial are available for download on [github].

Purpose of this Tutorial

For a lot of people (or at least for me) first learning about programing, it's overwhelming. There is a lot that I didn't know, and it just seemed like it was something that would be impossible to figure out. I was familiar with computers, I had installed operating systems, but I didn't have a strong grasp a lot of concepts that are very important in order to progress. This tutorial is aimed for anyone who, like me, would have benefitted from learning more about how to use R and how coding works before going to college. It is for people who know their way around an operating system, know the basics of using a graphing calculator, know how to use word, excel, and other office software, but don't understand (or maybe haven't even heard of) R or RStudio.

The links scattered throughout this tutorial are not required reading, but rather additional resources if you want to know more about what they pertain to.

important note

Please do copy and paste the code as you follow along rather than type it all out. Note that because of the ways that PDFs work, you will have to make sure you put the code into the correct number of lines to match the PDF formatting. For example,

```
library("readr")
library("leaflet")
library("tidyverse")
```

needs to look just like that, with different lines. It will not work if it reads as

```
library("readr")library("leaflet")library("tidyverse")
```

so take care when copying and pasting.

terms to learn: programing language, dataframe, table, packages, library, Documentation

What are R and RStudio? Why learn about them?

What are R and RStudio? If you've never heard of R, it's a versitile programing language with a simple built in interface that allows the user to do a wide range of things, from solving mathematical functions to the creation of maps. Basically, if you are in the program R, you tell it to do things, and it does them for

you. You tell it to solve an equation, it solves it for you, as long as you gave it the instructions in the right format.

RStudio is a program that makes the use of R more intuitive. If you are coming from using excel and other office software, you may wonder, like I did, why bother learning to use R at all. Excel should be able to do most any calculations that I might in day to day work, so why go through the effort of learning R?

There are several advantages to using R for data analsysis and visualization. R is free, there is a large online community dedicated to improving it, many tutorials online for how to use it and R has great versitility in how it can be used and applied.

R takes longer to learn, and for me trying to understand it was overwhelming at first. If learning about R seems hard, that might be because it is. But those hard fought skills, once learned, allow you to do so much more than excel or other programs. Those same things that you do in excel can be done faster, more accurately, and you can run statistical analysis in RStudio that would be completely impossible in excel. There are also problems with how excel runs its statistical analysis, for example, when faced with unequal numbers of samples. Excel will give results that are suboptimal, and there is as of yet no way be sure that analysis is done correctly in excel for every data situation, and it won't even tell you when you might have incorrect results. If you want to run, say, a hypothesis test correctly, and get back results that were calculated correctly, excel can let you down. You have to use some expensive statistical software... or learn to use R for free.

R can do more with data in general. Can excel be used to generate a map from data? No. But R can, and in this tutorial, I will show you how.

Getting R and RStudio set up

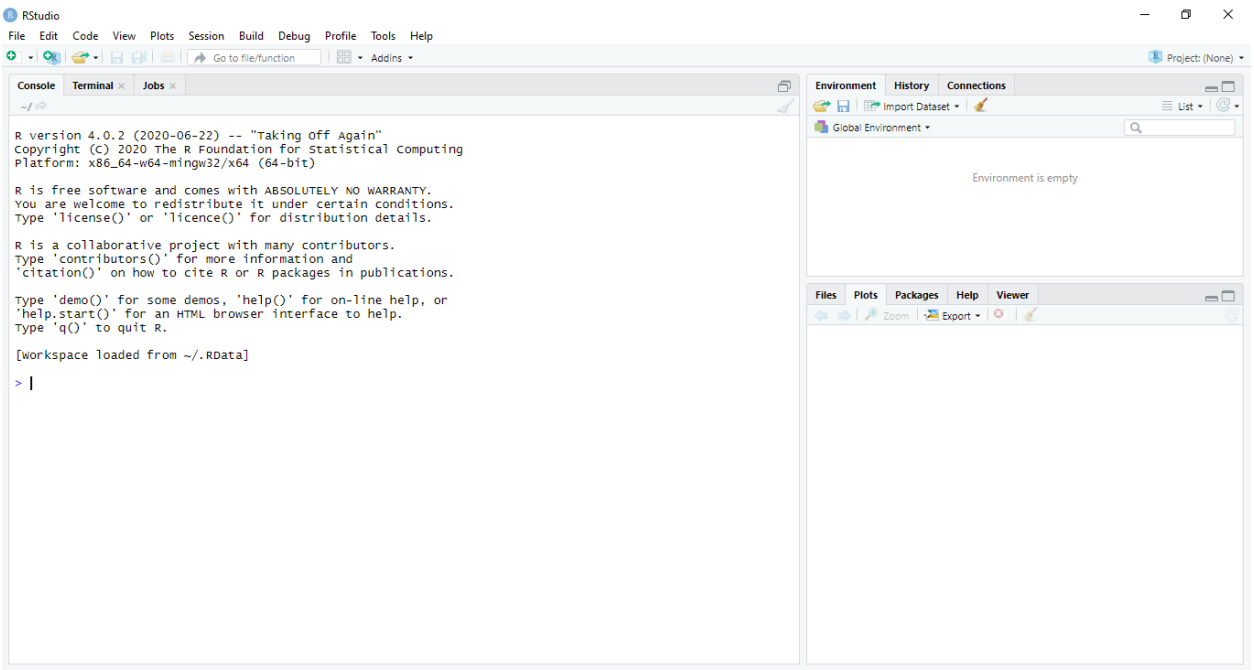
In order to continue with the tutorial, you will need to install Rstudio and R on your computer.

The most recent version of R can be downloaded by going [here]. Or, if you prefer a direct link, click [here for the Windows version], or [here for the Mac OS version]. Once downloaded, click on the file and it will allow you to install it like any other software. RStudio can be downloaded from [here], and installed by clicking on the file.

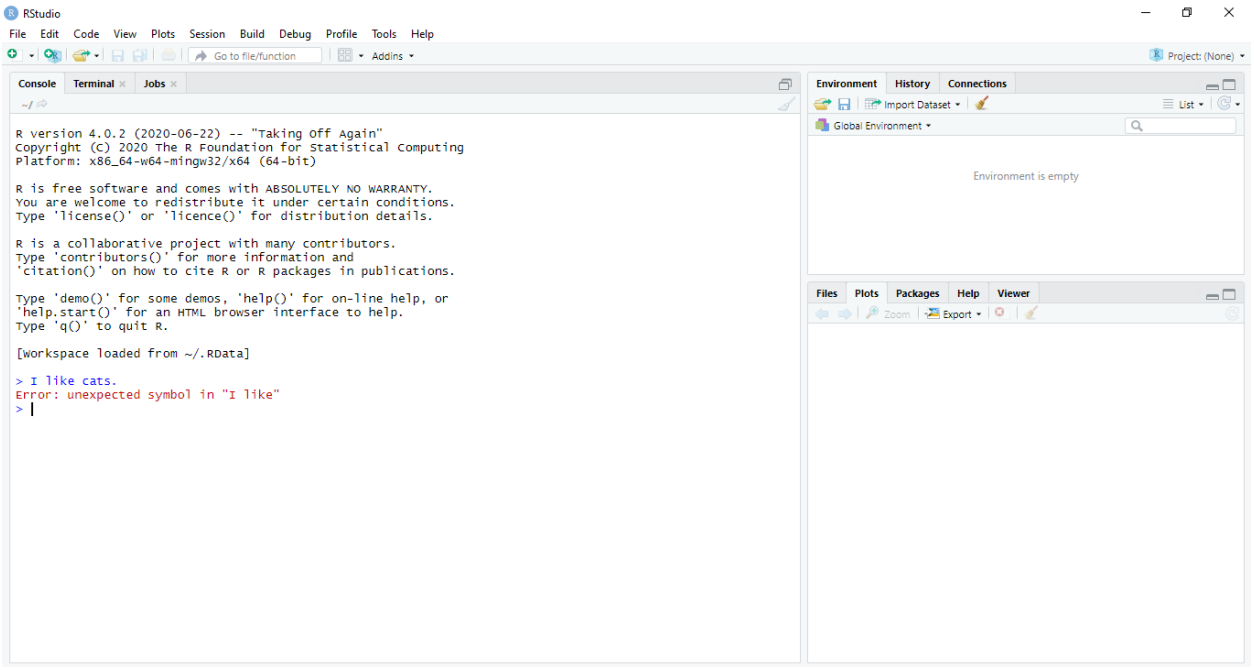
There is a great tutorial [here] for installing R and RStudio if you want a more detailed walkthrough.

Using RStudio for the first time Scripts, lines of code, and comments.

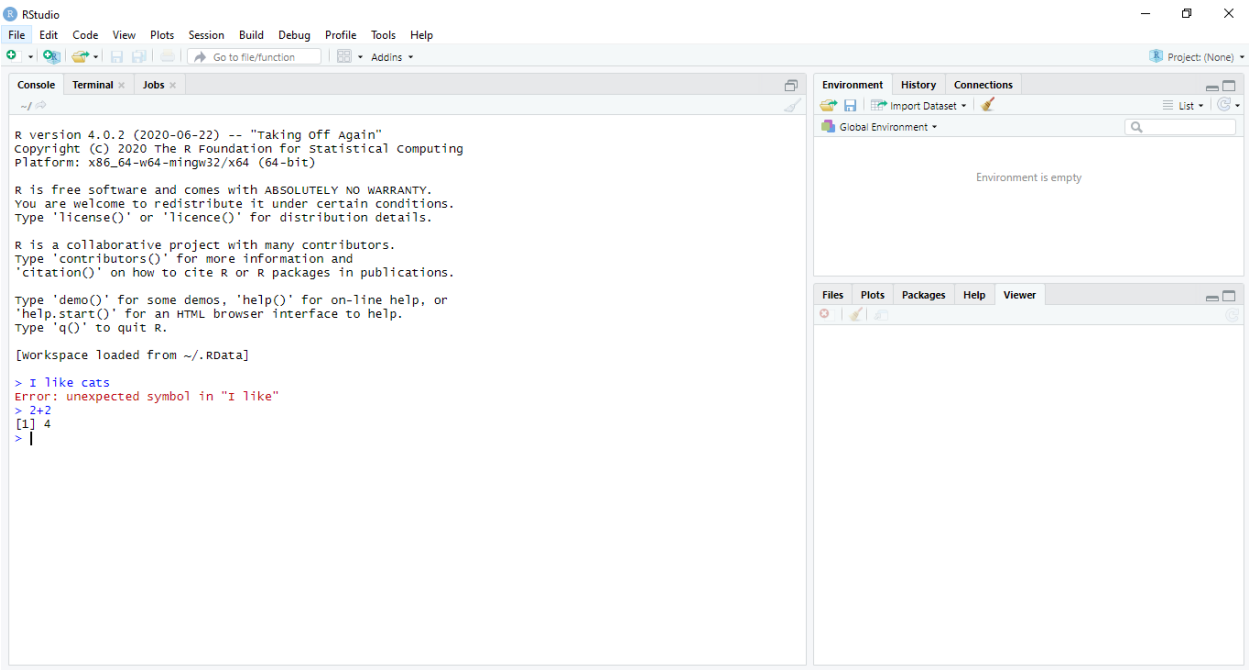
Now that R and RStudio are installed, load RSTudio and go ahead and look around at it for a few minutes. Take a look at the various menus and options available to click on.



You can see here that there are 3 windows, one on the left, and two on the right. The one on the left has a few tabs to click on, but the one we are most interested in is the console. If we click on the console, our typing curser will go there, and we can type all kinds of things. If I type a personal opinion in there, and then press enter, I get the following:



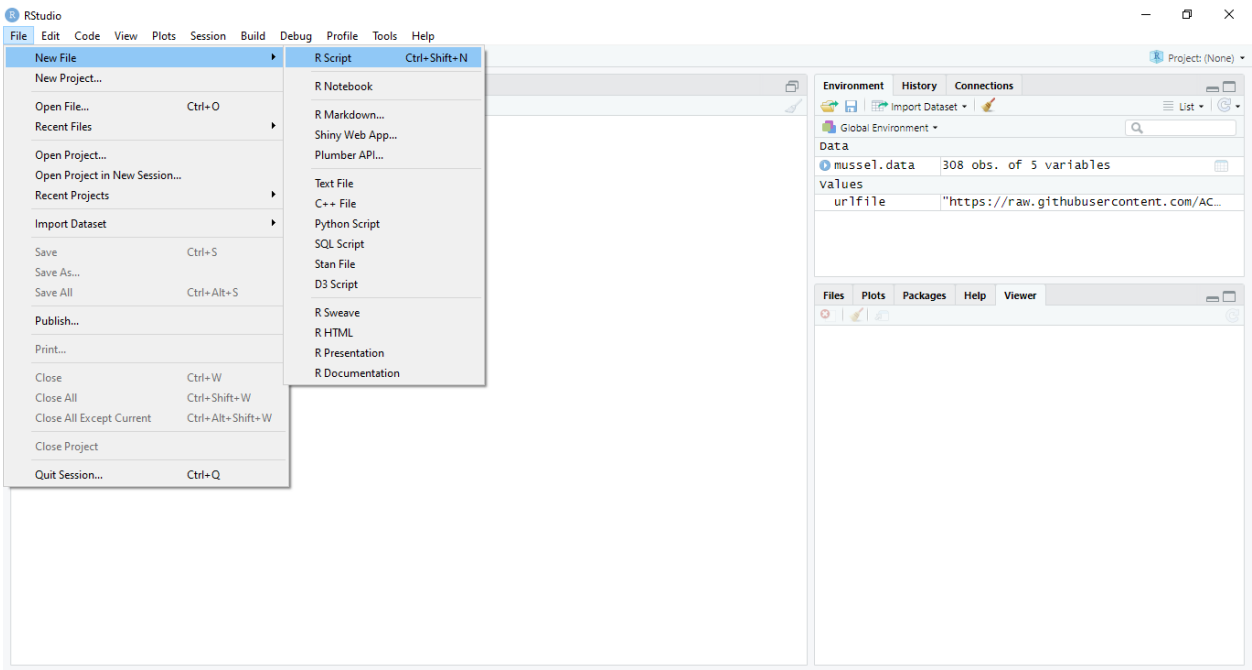
RStudio didn't understand the phrase "I like cats" so it returned an error code. RStudio will return errors like this whenever it encounters something it doesn't understand. It cannot understand everything we tell it, but it has a lot of built in things that it does understand. For instance, it will undestand calculations.



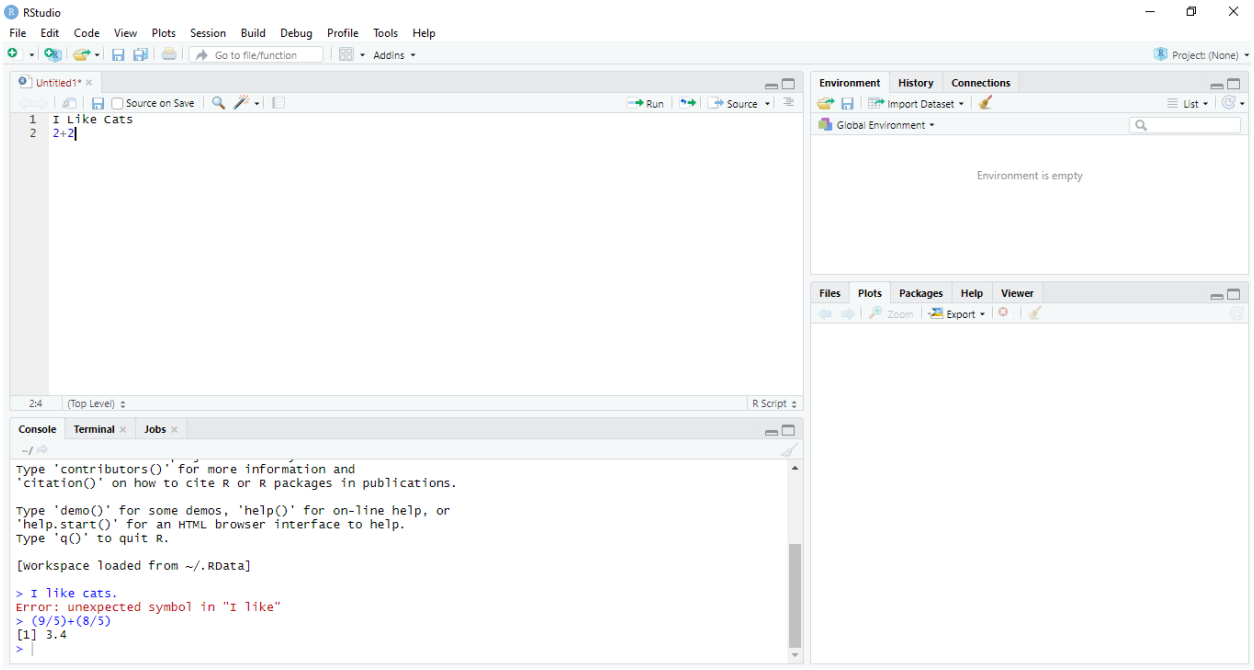
And it has other functions programed into it, such as the [poisson] function, which allows calculating the poisson approximation.

Each time we type into the console and hit enter, it will try to use what was typed in to do something. These instructions are called lines of code. They are called lines of code because every line is a complete instruction, and any instructions we want to give RStudio are sent in line by line. Typing code over and over again gets tedious fast though, so usually codes are stored in scripts. A script stores lines of typed text much the same as a word document, but in a different type of file so that R can understand it. You could save your code in a word or text document and then cut and paste into the console every time you want to run it, but I like using the RStudio's script file better.

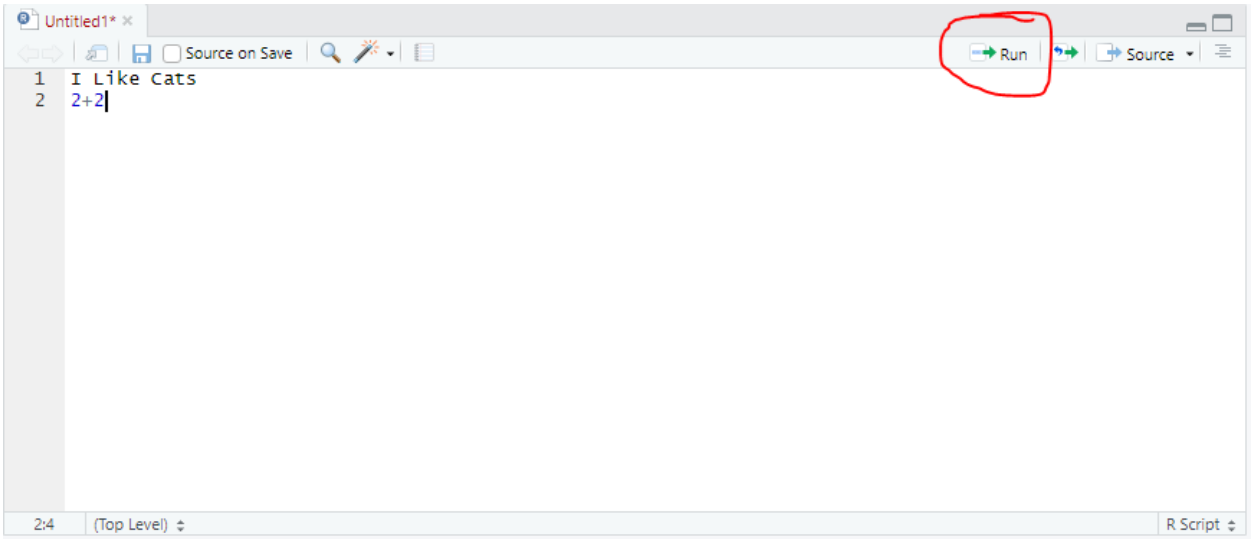
Open a new script by clicking the menu **File>New File>R Script**.



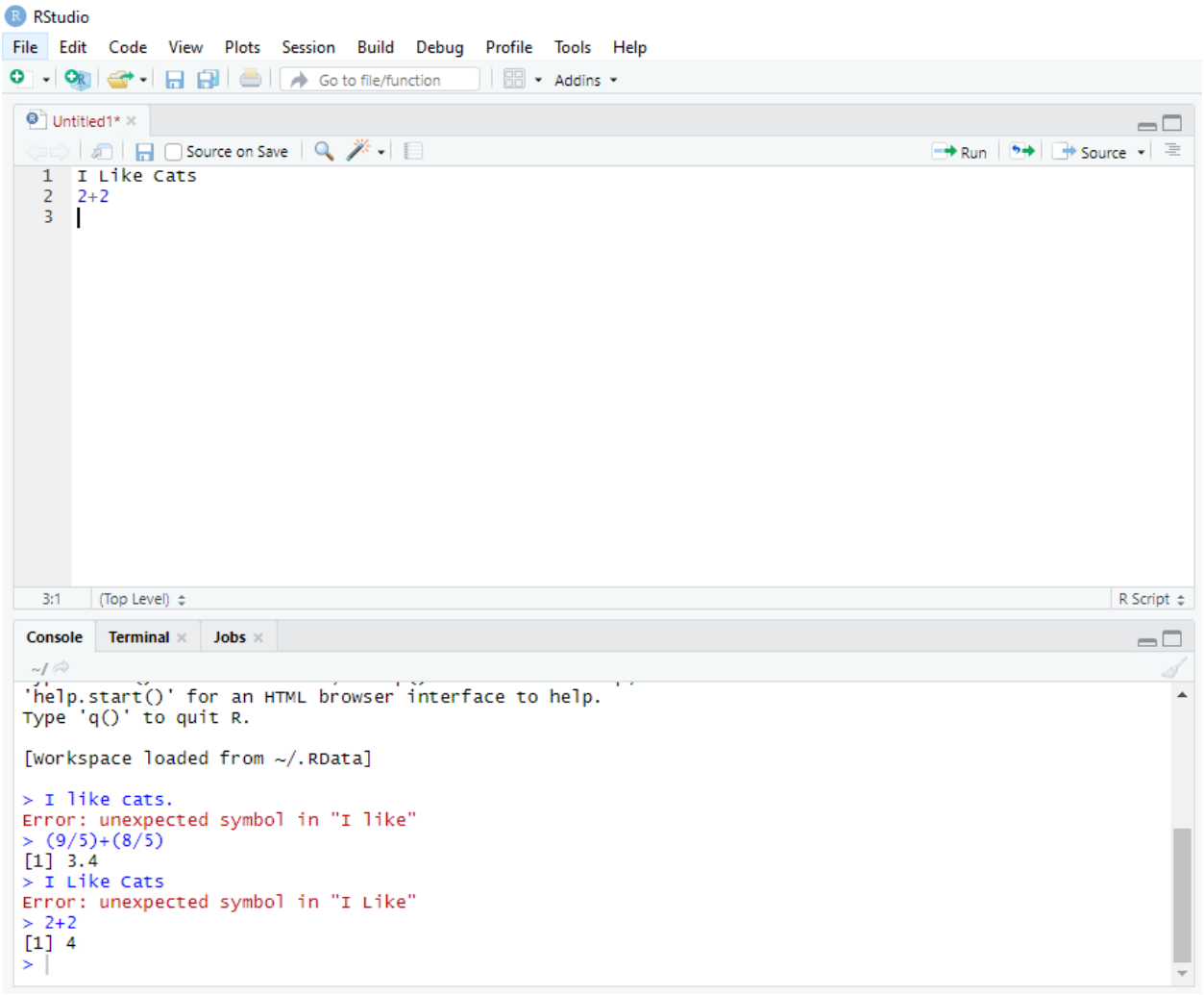
Now we have another pane open in our window. You can see that the line the cursor is in has a number to the left of it, 1. What happens if we type the line of code "I like cats" and press enter? And then type 2+2?



Nothing happpend other than that we now have another line labeled 2 with 2+2 in it. A script in R will not be run in the console unless you tell R to run it. We can run specific line of code we are on by clicking on the run code button here:



If I run a script, the consol will take each line in numerical order according to the line number. It runs line 1 first, then line 2, and so forth. If you go to the menu **Code>Run Region** there should be an option for running the entire script, along with a keyboard shortcut to do so.



And that's how a script works. It stores lines of code that you can run in order through a console. It is the foundation for how to code in R, for how to use R and RStudio. A more comprehensive introduction to R and its capabilities can be found [here], including how to do many things that are commonly done in excel. Our goal here is to make a map though, so let's keep moving.

packages, libraries, and documentation

Now that RStudio is up and running, you will need to make sure you have installed and loaded a few packages. Which for me, when I was first learning, was confusing. What is a Package?

To me, a package is something that you can download and add to RStudio in order to do more things. For instance, there might be a package that instead of giving me an error, will reply "I like cats, but dogs are better" if I type "I like cats" into the console. A package always gives RStudio some new ability that is useful to the user. They also have their own unique names, such as:

tidyverse
ggmap
mapview
ggplot
sf
rnaturalearth

rnaturalearthdata
readr

As you can see, packages often have strange and interesting names. Usually the names are directly linked to what they can do. tidyverse, for example, is great for keeping data tidy by allowing more robust control of data. It keeps things in your universe "tidy" so to speak. ggmap adds more options for making maps, ggplot adds options for plotting data, and so on. Even sf has a meaning, it deals with turning spatial data (like points on a map, or areas in a city) into Simple Features for visualization and analysis.

```
install.packages("package.name")
```

For each of the packages we will want. Once a package is installed, it should not need to be installed again. Once it is installed, we cannot use any of the new abilities it put into RStudio unless it is loaded into RStudio's "library". This library represents the codes that RStudio knows how to recognize and use. Packages are not automatically loaded into the library when installed, packages have to be loaded in by typing

```
library("package.name")
```

for each package that you want to use. This loads the package into our RStudio session's "library", giving RStudio to the added functionality from these packages.

Packages are not loaded automatically into RStudio, and the library resets whenever RStudio is restarted. This means packages must be loaded into the library every time we start a new session of RStudio. Typing this many lines of code may sound tedious, because it is. Typing in that much code each time RStudio is opened would slow down workflow. This is where we can start to take advantage of a core property of programming that we discussed earlier, the script.

We will need the packages

leaflet
readr
tidyverse

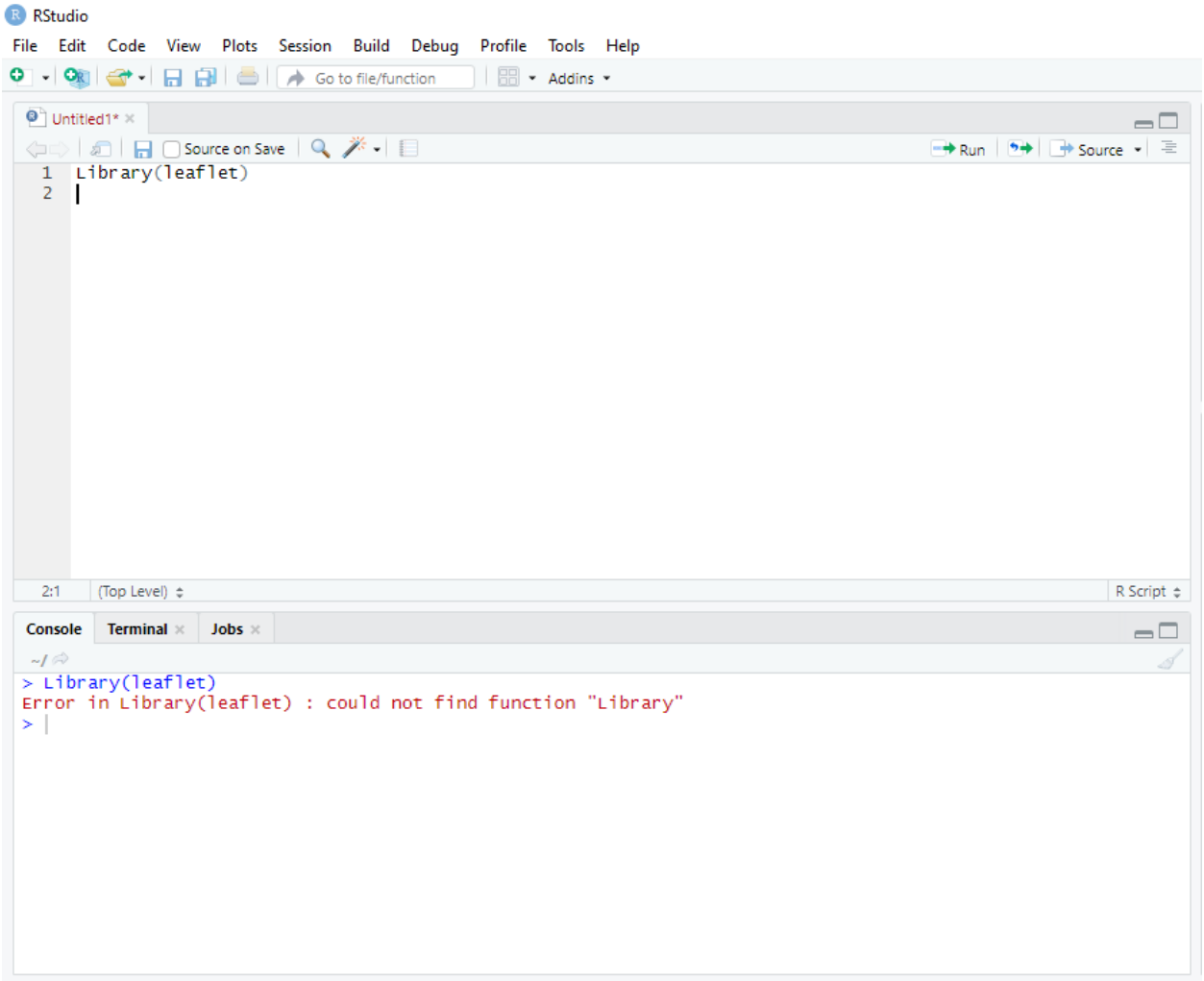
You will have to install each of these by typing in your script

```
install.packages("readr")
install.packages("leaflet")
install.packages("tidyverse")
```

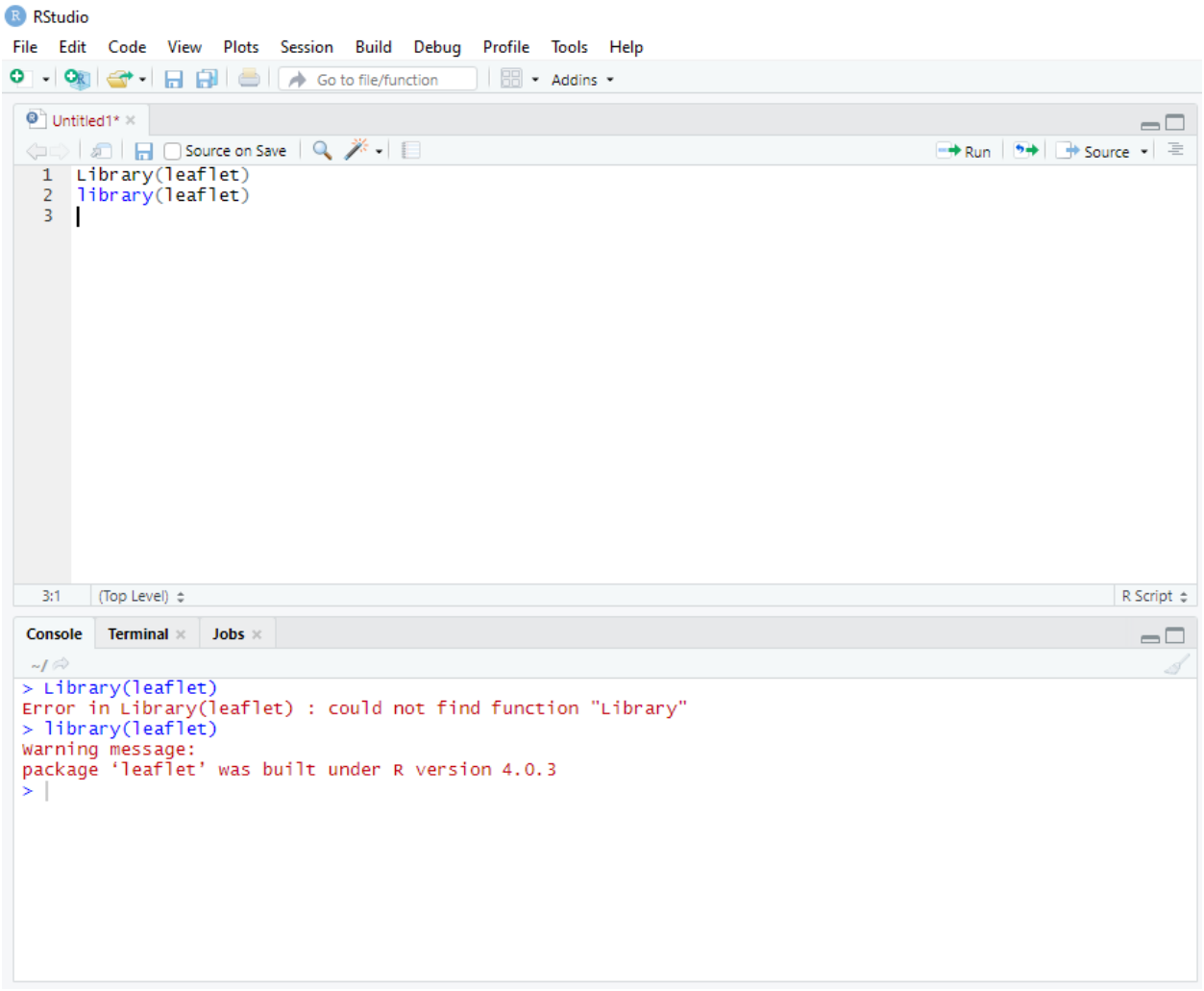
and running it. It may take a few minutes for the packages to install. I already have those packages intalled on my computer, so the first part of my script file looks like this:

```
library("readr")
library("leaflet")
library("tidyverse")
```

and to run this code, all I have to do is click the run button and run each line. For example, If just wanted to load the package leaflet:



Woops! As you can see, I must have typed something wrong. R is a case sensitive language, so just like my computer password, **if I type something capitalized that shouldn't have been, it won't work.** Let's try that again.



You may have noticed that there were a few error messages in the console. The first error is telling me that "Library" isn't a function that exists. Because R is case sensitive, it recognizes library, but not Library. The second one is warning me that the package leaflet was built under R version 4.0.3. It wants me to know this because I am running a different version of R, and sometimes packages don't work right if they were built under an older or newer version of R than the one I am using. It doesn't necesarily mean that the package won't work, but R warns me anyway.

Error messages in R do not always mean that something completely failed to work-sometimes they are more like warnings to notify you of a possible issue, without there being anything to worry about. In my experience, knowing which messages to ignore and which to pay attention to becomes easier with practice.

Each one of the packages we installed and loaded do a lot of things. We won't be using their full functionality in this tutorial, but you can see more about what each one can do by typing

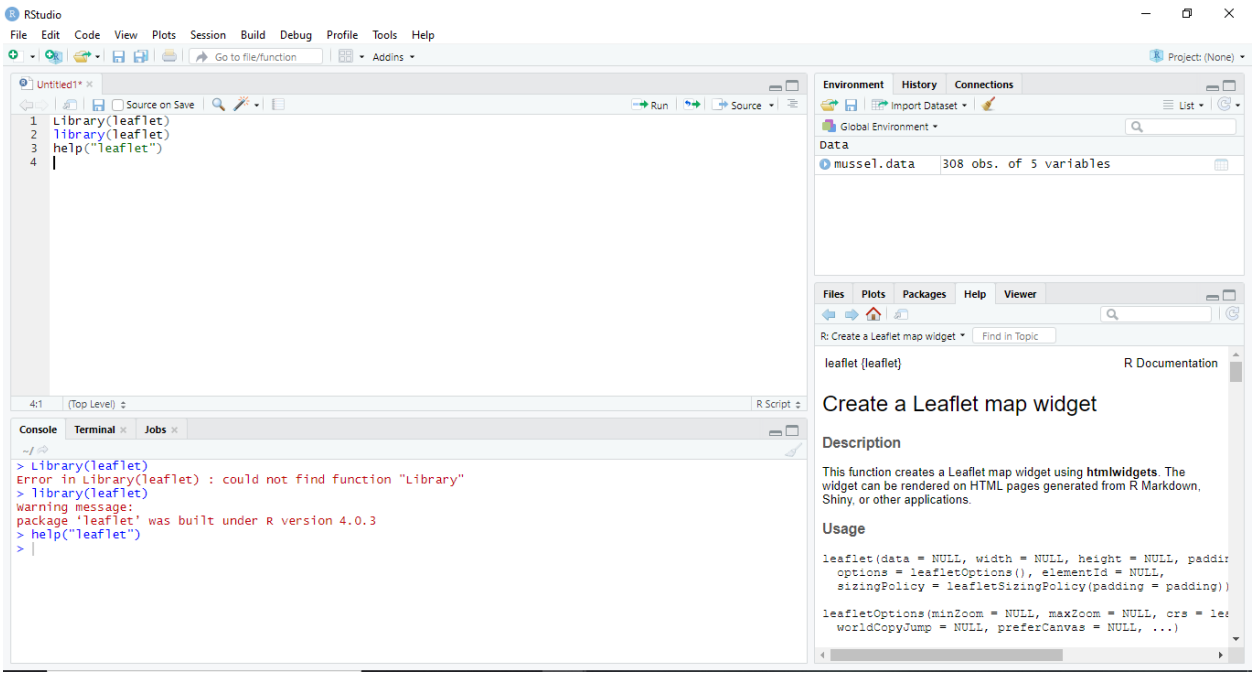
```
help("package.name")
```

into the console and pressing enter to run the code.

for example, typing

```
help("leaflet")
```

into the console and pressing enter to run it will bring up a description of the package in the help window in the lower right of the screen.



Which brings us to another important term-**documentation**. What it is actually showing is the documentation for the package ggplot2. Every package has documentation, which tells us about who made the package, what it does, how to use it, and how it works. If you are like me, you might be thinking right now, that's great! I can just go to documentation and it will always explain to me how to use the software and do what I am trying to do. This is true, but, especially if you are newer to programming, there might be a lot of jargon that is hard to understand. Don't be afraid to google the name of the package or look for answers to your question online if the documentation is difficult to parse. There are plenty of great tutorials outside of documentation, such as [this one] that can show you how to use packages to their fullest potential.

Speaking of documentation, if you want to have a note in your code that won't cause an error in Rstudio, you can put in a comment by using the # symbol:

```
#this is a comment. It will appear in the console, but won't do anything. If I type
2+2 #and then a comment, it will run the part that came before the #, and output 4.
# it is a good idea to comment lines of code so that when you come back to them,
#you know what they were for. This is part of good data management,
#which, while extremely important, is not covered in this tutorial.
```

Now that we know how to install and load packages, let's move on to importing some data and using a package to create a map with it.

Getting data into R

If you are familiar with Excel, you are used to using workbooks with cells in columns and rows. R does data a little differently, which is explained really well [here].

For now, note that R cannot import .xlsx files (the default filetype for excel), but it can import .csv files. The data we will be using today is saved as a .csv file. .csv files can be imported into R as a **dataframe**, which is one way that R reads and stores data. R stores data as vectors, but if you ask R to show you a dataframe, it looks and functions much like a table in excel will. It's easy to think of it as a table, but be

sure to remember that it is in fact called a dataframe, and has different properties than a table or other ways of storing data. Certain packages and functions won't work with tables but will work with dataframes, which makes it an important distinction.

There are a few ways that data can be imported into R. More information can be found at sites such as [this one], but let's go ahead and look at one way of bringing the data in. We are going to use a package called readr, which allows us to pull data directly from websites, as a subtype of dataframe called a tibble.

```
mussel.data<-read_csv("https://raw.githubusercontent.com/
                      AChase44/Mussels-Rule-Repository/master/data/Tutorial_1.csv")
#read file into an R data.frame, names the dataframe mussel.data.
```

This line of code includes a new function(). Technically, install.packages() and library() are also functions. install.packages() installs whatever package name you put into (), and library() loads whatever package name you put into (). But functions can do a lot more than just that.

The function read_csv() takes whatever item is in the () and tries to turn it into a dataframe. So when we write

mussel.data<-read_csv("https://raw.githubusercontent.com/AChase44/Mussels-Rule-Repository/master/data/Tutorial_1.csv")

we are telling R to take the data from that url and turn it into a dataframe. We also want to make sure that our dataframe has a name, so we tell it that we want to call that dataframe mussel.data by saying that the result of the function read_csv(url) is mussel.data. That's what the <- is for. (<- can also be used to add things to an item that already exists, depending on what packages you are using.)

mussel.data<-read_csv(url)

mussel.data (is) our imported dataframe from the data url

Generally dataframes are always named something related to what the data is from. You can pull data from your own files on your own computer, but that require that the data are formatted correctly for R, which is outside the scope of this tutorial. let's focus on seeing what we can do with the provided data. We know that we pulled it into RStudio, but we can't see it. Let's look at the first few rows of data using the head() command.

the function head() will pull the first few rows of the dataframe we put in the ().

```
head(mussel.data)
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.4340	45.81066	NA	NA
Big Creek	-114.9372	45.12670	NA	NA
Fish Lake Creek	-114.9901	46.46025	NA	NA
Lick Creek	-115.7622	45.06290	NA	NA
Lick Creek	-115.7846	45.07168	NA	NA
Lochsa River	-114.7620	46.51069	NA	NA

We can also look at the entire dataframe by typing

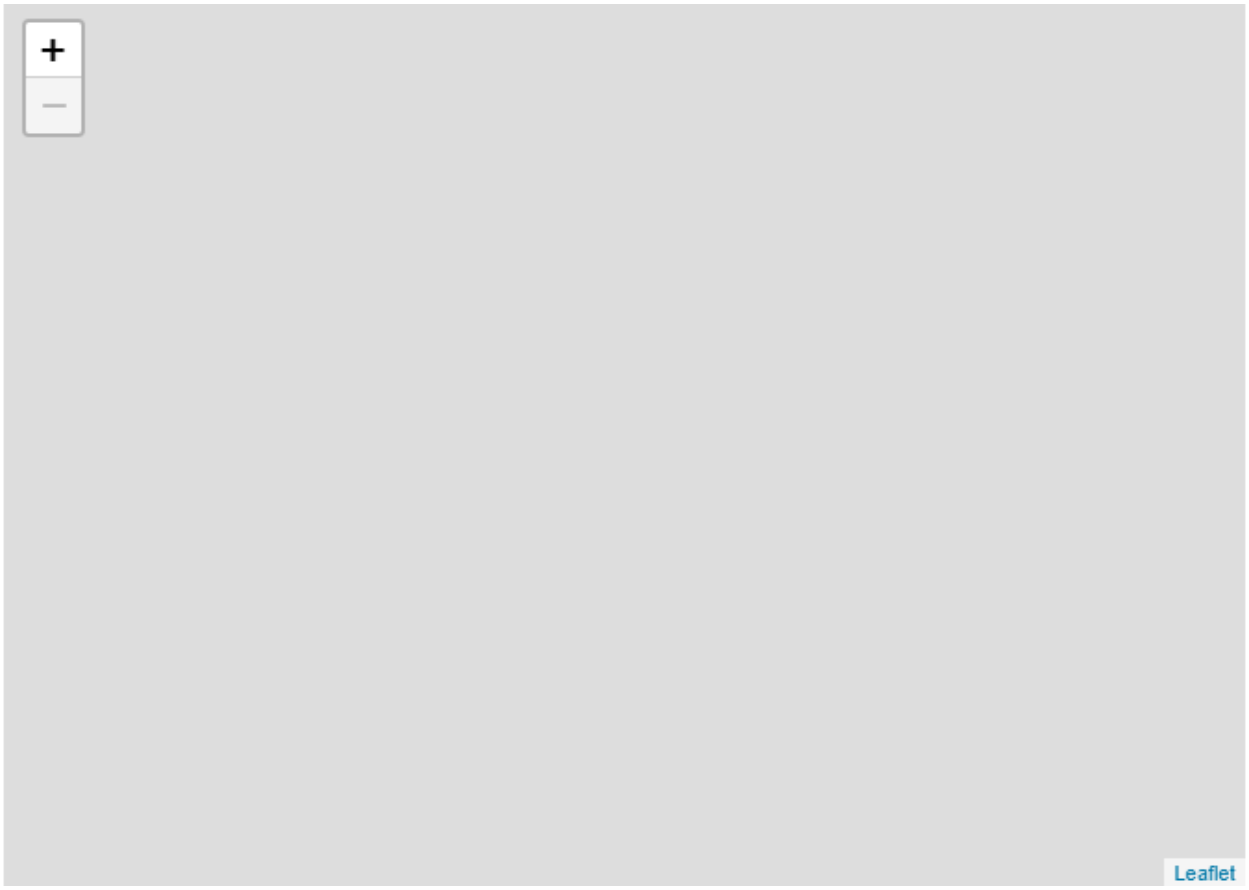
```
view(mussel.data)
```

(you can return to your R script by closing the mussel.data tab or clicking on your script's tab.)

making a map using the package leaflet

Ok, so we have this data. I know that this is from Idaho of streams that were surveyed for freshwater mussel pressence. There are 5 columns: Stream, which tells us the name of the stream that the survey took place in, NewLong and NewLat which are gps coordinates in in decimal degrees, and then two species columns that show approximately how many mussels of each species were found at the site. It is sorted alphabetically by stream name. Personally, I'm not great at visualizing longitude and latitude in my head, much less when it is over 100 different rows of data. We can get a better look at it using a function from a package called leaflet.

```
mmap<-leaflet(mussel.data)
#leaflet knows to use mussel.data. to create something called "mmap"
#(I chose the name mmpa because it stands for "mussel map".)
mmap #shows us the map
```

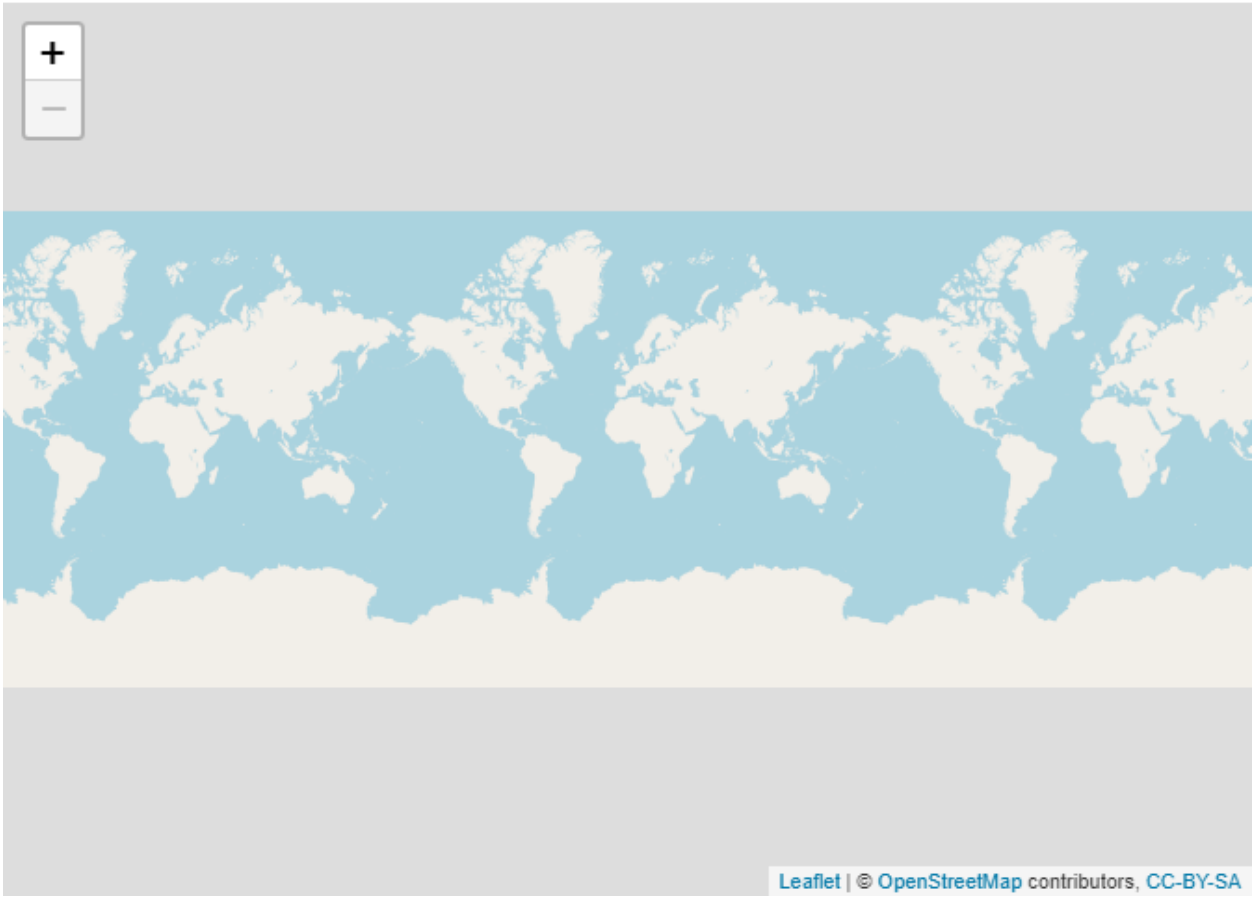


AAH! That isn't helpful at all! Sometimes when learning new functions from new packages, it is easy to make mistakes that at first seem catastrophic. (you RStudio may)

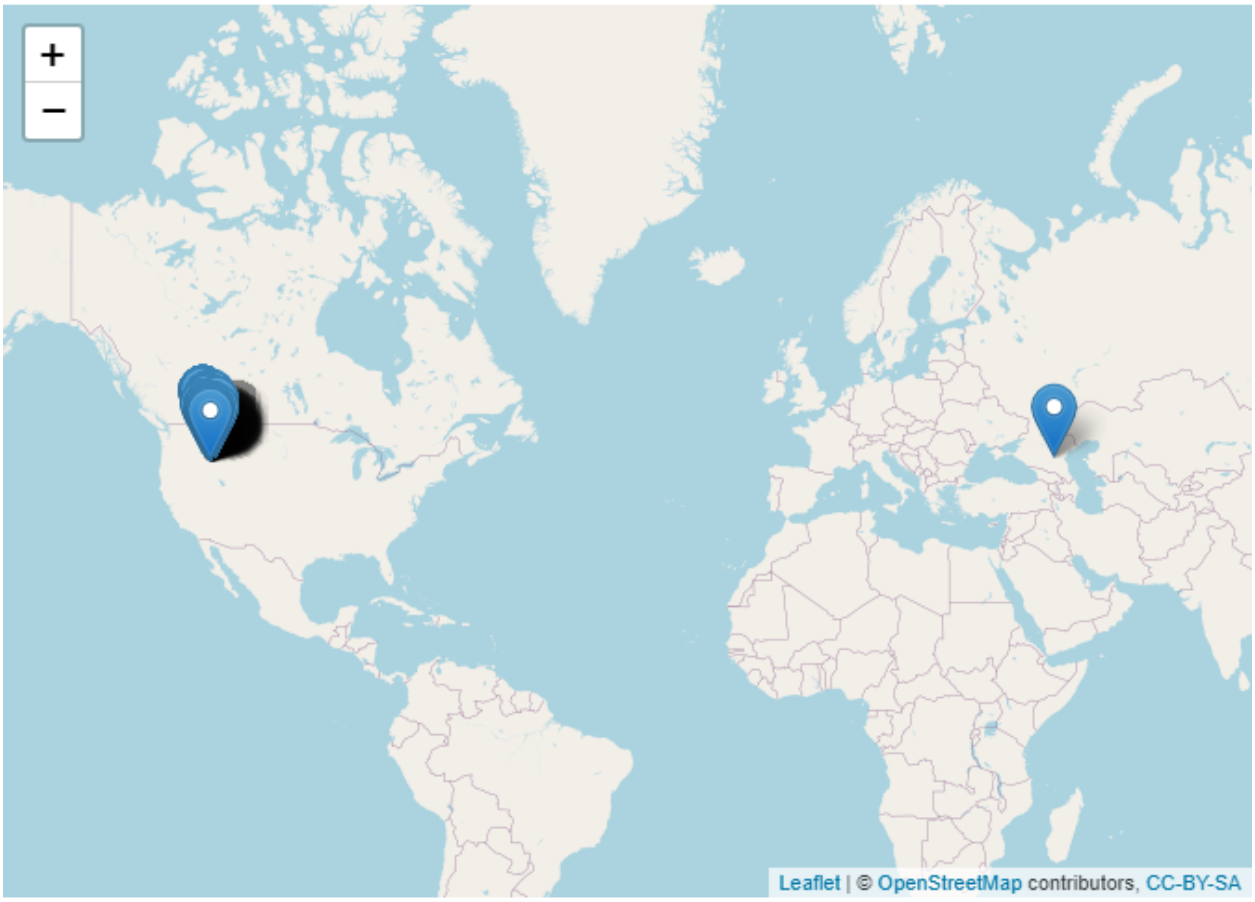
What happened is that leaflet is a set of functions that work additively. We told leaflet to use the dataframe mussel.data, but we didn't tell it anything else. It needs to know more before it can tell us something useful, like if we want markers, what columns have the latitude and longitude data for said markers, and what kind of background tiles we want for our map. We already made this thing called mmap, and we can just keep adding to it, by using the <- symbol and more functions from the leaflet package.

```
mmap<-addTiles(mmap) #we add background tiles to the map
mmap #displays mmap
```

Telling it we wanted a background with addTiles(mmap) gave us a nice background. let's add our data points with the leaflet function addMarkers().




```
mmap<-addMarkers(mmap,lng=~NewLong,lat=~NewLat) #this adds point data from our
#mussel.data dataframe, and tells leaflet which columns the spatial data are stored in.
mmap #displays mmap.
```

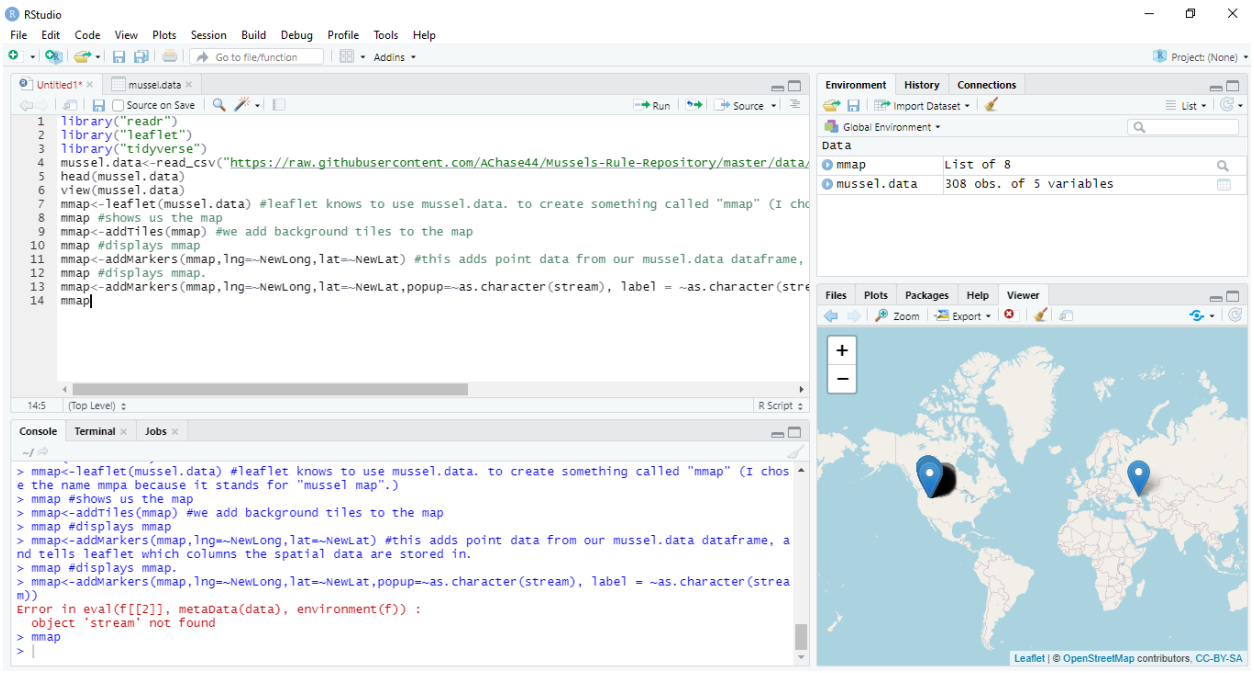


It may seem like I pulled those additional lines of code out of thin air. However, that's not the case. Leaflet has a [really good website] with guides regarding how to use its codes. It's easy to feel lost because it can seem like there are invisible rules regarding how to code for what you want, and in some ways that is true. Coding is a language, like any other it requires memorization and practice to master. The only way to get good at using a package is read about how to use it, from documentation or from a tutorial, and then experiment with what you learn.

Leaflet is useful because it is intuitive and interactive. It allow us to drag and click, and zoom, much like googlemaps, but with our own data. If you zoom out, you can see we have a large cluster of data points in Idaho, where we know our data were collected. However, there is also a lone site in Russia. We know that our data were not collected outside of Idaho, so there must have been a mistake. We either need to find out what the error was, or take note of the error and remove the data point before moving on. Generally speaking, even if we remove it, someone will have to come back and fix it later anyway. Let's see if we can figure out what went wrong.

Right now there is no way to know what point belongs to what row of data. A lot of functions will allow for the inclusion or omission of certain information. We only told it to show us the locations of our data, but we can tell it to do more. However, we can tell leaflet to label each of the data points, by changing the code for addMarkers() just a little bit.

```
mmap<-addMarkers(mmap,lng=~NewLong,lat=~NewLat,popup=~as.character(stream),
                 label = ~as.character(stream)) #new changes
mmap #show the map
```



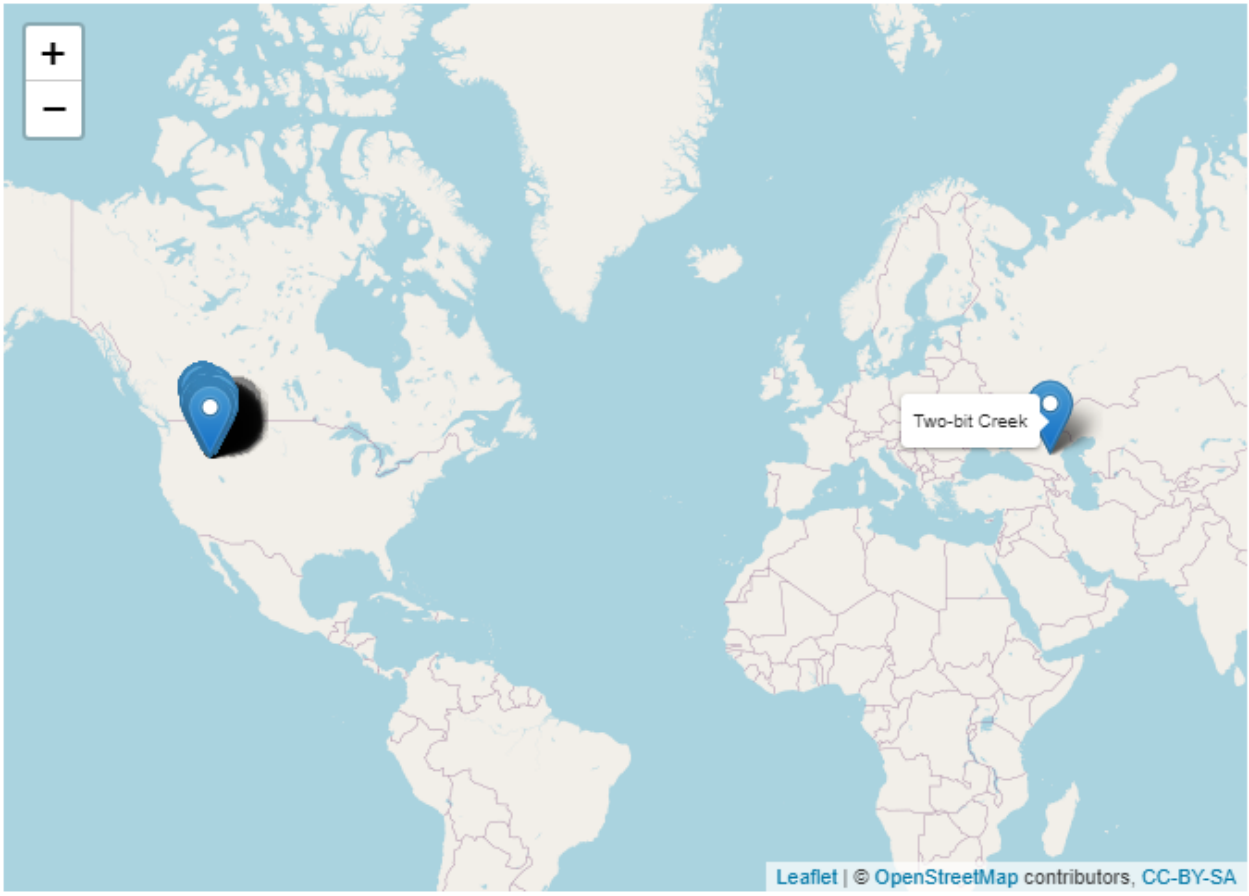
uh-oh! we got an error message. Let's see what could be going on. It says stream does not exist. Let's check our data column names again to see if we spelled it wrong.

```
head(mussel.data)
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.4340	45.81066	NA	NA
Big Creek	-114.9372	45.12670	NA	NA
Fish Lake Creek	-114.9901	46.46025	NA	NA
Lick Creek	-115.7622	45.06290	NA	NA
Lick Creek	-115.7846	45.07168	NA	NA
Lochsa River	-114.7620	46.51069	NA	NA

It is spelled the same, but R is case sensitive. We'll have to run it again with stream capitalized.

```
mmap<-addMarkers(mmap,lng=~NewLong,lat=~NewLat,popup=~as.character(Stream),
                 label = ~as.character(Stream)) #fixed typo that caused error
mmap #show us the map
```



there we go!

If we mouse over the errant data point, we can see that it came from Two-bit creek, and that gives us a place to start. If we go back to the data set, we know that the stream name is Two-bit creek, so we can look at the table and try to find it visually.

```
view(mussel.data)
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.43398	45.81066	NA	NA
Big Creek	-114.93725	45.12670	NA	NA
Fish Lake Creek	-114.99005	46.46025	NA	NA
Lick Creek	-115.76220	45.06290	NA	NA
Lick Creek	-115.78461	45.07168	NA	NA
Lochsa River	-114.76200	46.51069	NA	NA
Potlatch River	-116.44164	46.79001	NA	NA
South Fork Salmon River	-115.70495	44.48330	NA	NA
Two-bit Creek	44.66734	44.66734	NA	NA
East Fork Potlatch River	-116.36327	46.85105	NA	10-100
South Fork Salmon River	-115.69436	44.63488	NA	10-100
American River	-115.43398	45.81066	0-10	NA

Because we know that when this data set was made it was sorted by stream name, we know that there is only one entry for Two-bit Creek. It looks like the longitude is wrong, an error must have occurred when the

data was first put in. It may be fun to guess that the person who recorded the data accidentally put in the latitude values for both columns. We cannot know for sure what happened here to cause the error, but the long and lat values are very similar. We do know the name of the creek. In this situation, the best solution would be to go back to the person who recorded the data and ask them if they know the correct longitude and latitude for this data point. That solution is outside of the scope of this tutorial however, and it is not uncommon to find errors in data that does not have a good point of contact. Another solution would be to estimate where this was taken, by assuming the latitude is correct and estimating what the longitude would have to be for the data point to fall on Two-bit creek. The problem with this solution is that it assumes that the latitude is correct and we don't know if that is the case for sure.

Remember that R stores data in vectors, but they act much like a table. If we know what row it is in, we can remove it. R stores dataframes in such a way that each row below the column names have their own numerical identification. we can call and look at each of the rows or columns that we want by using the following code:

data.frame(row,column)

where data.frame is the name of our dataframe (mussel.data) and the row and columns are referred to by numbers. for instance, if we wanted to look at data in the first row and the first column of mussel.data, we would type

```
mussel.data[1,1]
```

Stream
American River

which in this case is just "American River".

We can also call several rows of data. For example, we could call the first 3 rows of column 1:

```
mussel.data[1:3,1]
```

Stream
American River
Big Creek
Fish Lake Creek

the colon tells us that we want from rows 1 "to"3.

we can also call the first few rows with all columns included.

```
mussel.data[1:3,]
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.4340	45.81066	NA	NA
Big Creek	-114.9372	45.12670	NA	NA
Fish Lake Creek	-114.9901	46.46025	NA	NA

And this time, the colon still tells us that we want from rows 1 "to" 3, and by leaving the space after the comma empty, we are saying we want all of the columns.

By counting down the rows, we know that the errant data point is in row 9.

For the purposes of this tutorial, let's omit that data point, using a method from [here].

We can view a table that includes all of the data except for row nine by the following code:

```
mussel.data[-9,]
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.4340	45.81066	NA	NA
Big Creek	-114.9372	45.12670	NA	NA
Fish Lake Creek	-114.9901	46.46025	NA	NA
Lick Creek	-115.7622	45.06290	NA	NA
Lick Creek	-115.7846	45.07168	NA	NA
Lochsa River	-114.7620	46.51069	NA	NA
Potlatch River	-116.4416	46.79001	NA	NA
South Fork Salmon River	-115.7050	44.48330	NA	NA
East Fork Potlatch River	-116.3633	46.85105	NA	10-100
South Fork Salmon River	-115.6944	44.63488	NA	10-100
American River	-115.4340	45.81066	0-10	NA
Big Creek	-114.8606	45.10480	0-10	NA

What this does is it says I want everything from the mussel.data dataframe, except for row 9. This just displays the dataframe without row nine, it does not actually change mussel.data. But, we can use this to create a new dataframe that does not have row nine.

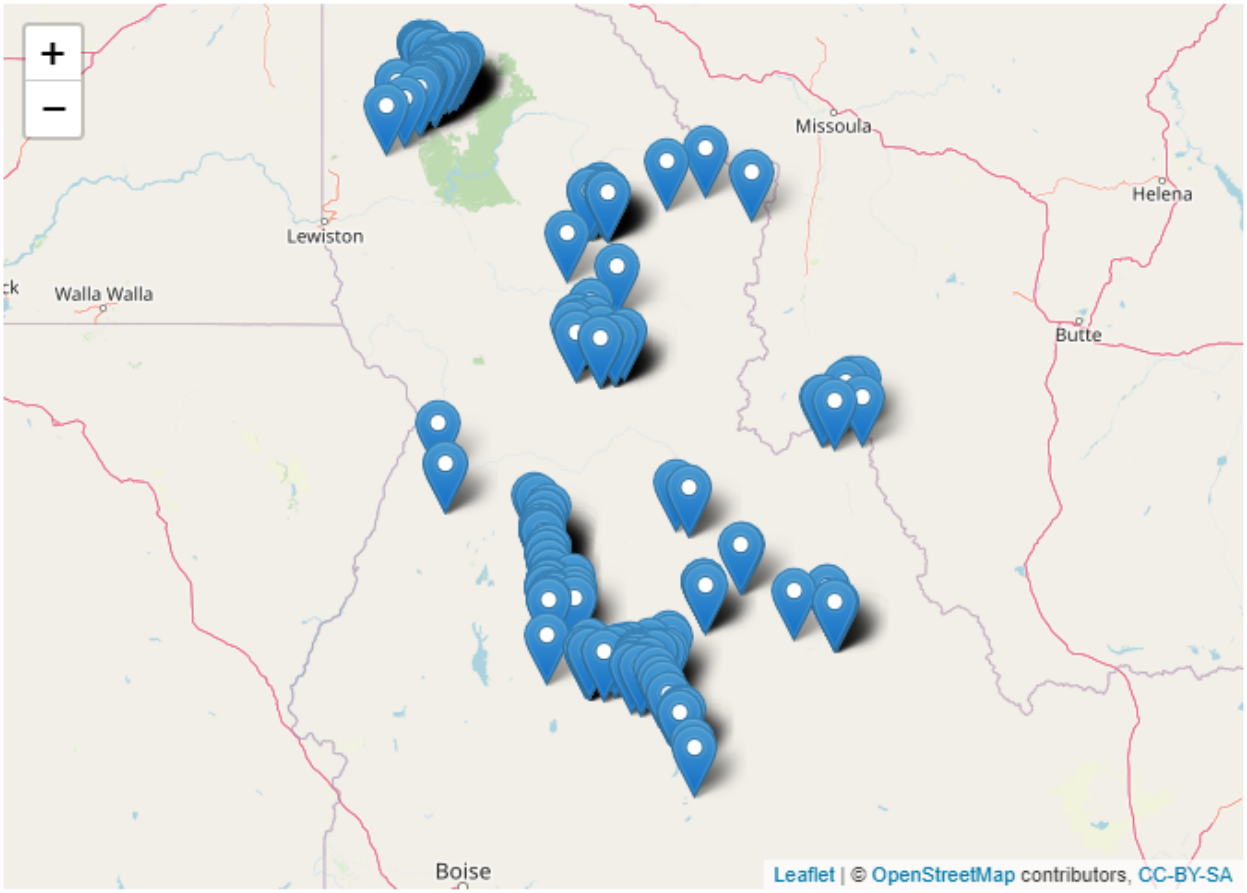
```
no9mussel.data<-mussel.data[-9,]
no9mussel.data
```

Stream	NewLong	NewLat	Western_Pearlshell	Western_Ridged
American River	-115.4340	45.81066	NA	NA
Big Creek	-114.9372	45.12670	NA	NA
Fish Lake Creek	-114.9901	46.46025	NA	NA
Lick Creek	-115.7622	45.06290	NA	NA
Lick Creek	-115.7846	45.07168	NA	NA
Lochsa River	-114.7620	46.51069	NA	NA
Potlatch River	-116.4416	46.79001	NA	NA
South Fork Salmon River	-115.7050	44.48330	NA	NA
East Fork Potlatch River	-116.3633	46.85105	NA	10-100
South Fork Salmon River	-115.6944	44.63488	NA	10-100
American River	-115.4340	45.81066	0-10	NA
Big Creek	-114.8606	45.10480	0-10	NA

Here we told R that we want to make mussel.data without row 9 into a new dataframe called no9mussel.data.

So now we have a dataframe with the data we want, without having to leave R. Let's map it again, like before, but with the no9mussel.data dataframe, and let's call the map no9mmap.

```
no9mmap<-leaflet(no9mussel.data)
no9mmap<-addTiles(no9mmap)
no9mmap<-addMarkers(no9mmap,lng=~NewLong,lat=~NewLat,popup=~as.character(Stream),
                    label = ~as.character(Stream))
no9mmap
```



Now when we map the data, the Russian mussel data point is omitted. We have successfully created a map of our data points.

Final Thoughts

I hope this was a helpful tutorial for anyone who is just looking to see what R can do. Learning the very basics of using R takes time and patience, and will at times be frustrating, but can also be extremely rewarding. I personally found it intimidating, but ultimately possible, and I hope you will give it a chance.

questions for review

What is the difference between a line of code and a script?

What is a package, and how do you make sure it is ready to use in R?

What would you have to change in this code to make it work correctly?

```
Install.Packages("Leaflet")
```

coding challenge level 1

This challenge, if you wish to accept it, is to create a map just like the one we did in this tutorial-but with your own csv datafile. Use some data that has meaning to you-where your favorite people were born, the most important places you have visited throughout your life, or something else that matters to you.

Hint: Refer to [this webpage] for guidance on how to import your own data.

coding challenge level 2

The challenge, if you wish to accept it, is to find a way to change the icon for the points on the map to an image of your choosing.

Hint: Refer to [this webpage] for guidance on how to use the package leaflet.