Introduction to R

We will learn a lot more about statistical programming this semester, but we'll start with a crash course on `R` with the idea of getting you up-and-running.

I listed a few references in the Introduction, but this section will mostly follow the discussion in Introduction to Data Science: Data Wrangling and Visualization with R, by Rafael Irizarry. I'll abbreviate this reference as IDS throughout this section.

IDS is not specifically geared towards Econometrics, but I think it is a really fantastic book and resource. In this section, I cover what I think are the most important basics of R programming and additionally point you to the references for the material that I cover in class. I will cover chapters 1-3, 6, and 20 in some detail, and I strongly recommend reading all of these chapters closely. As you have time, I'd recommend reading 4-5 (about working with data) and 7-10 (about visualizing data). The course should set you up so that the remaining chapters of the book can serve as helpful reference material throughout the rest of the semester.

Setting up R

This section covers how to set up R and RStudio and then what RStudio will look like when you open it up.

What is R?

Related Reading: IDS 1.1

R is a statistical programming language. This is important for two reasons

- It looks like a "real" programming language. In my view, this is a big advantage. And many of the programming skills that we will learn in this class will be transferable. What I mean is that, if you one day want to switch to writing code in Stata or Python, I think the switch should be not-too-painful because learning new "syntax" (things like where to put the semi-colons) is usually relatively easy compared to the "way of thinking" about how to write code. Some other statistical programming languages are more "canned" than R. In some sense, this makes them easier to learn, but this also comes with the drawback that whatever skills that you learn are quite specific to that one language.
- Even though R is a real programming language, it is geared towards statistics. Compared to say, Matlab, a lot of common statistical procedures (e.g., running a regression) will be quite easy for you.

R is very popular among statisticians, computer scientists, economists.

It is easy to share code across platforms: Linux, Windows, Mac. Besides that, it is easy to write and contribute extensions. I have 10+ R packages that you can easily download and immediately use.

There is a large community, and lots of available, helpful resources.

- ChatGPT
- Your favorite search engine
- StackOverflow

Downloading R

We will use R (https://www.r-project.org/) to analyze data. R is freely available and available across platforms. You should go ahead and download R for your personal computer as soon as possible — this should be relatively straightforward. It is also available at most computer labs on campus.

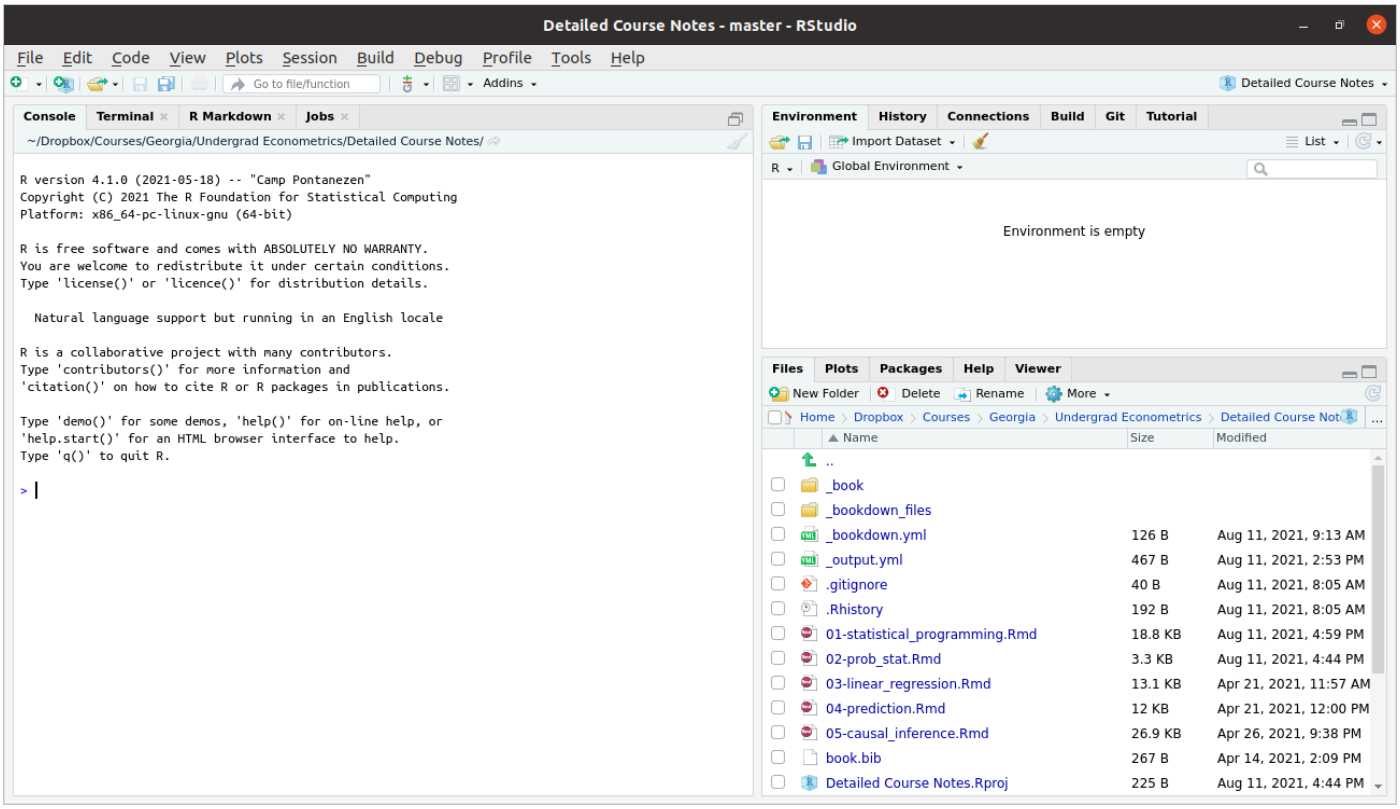
RStudio

Base `R` comes with a lightweight development environment (i.e., a place to write and execute code), but most folks prefer RStudio as it has more features. You can download it here: https://www.rstudio.com/products/rstudio/download/#download; choose the free version based on your operating system (Linux, Windows, Mac, etc.).

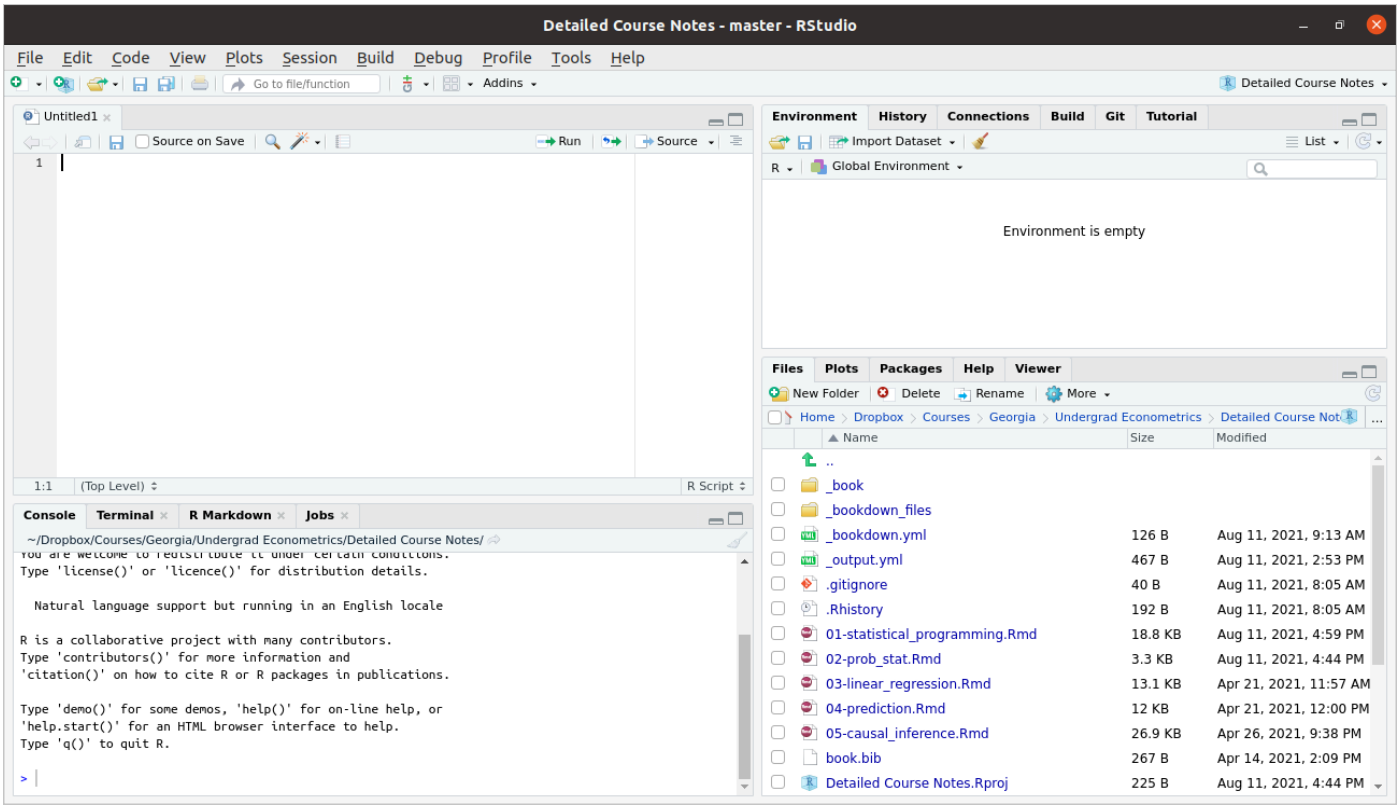
RStudio Development Environment

Related Reading: IDS 1.4

When you first open Rstudio, it will look something like this



Typically, we will write **scripts**, basically just as a way to save the code that we have written. Go to `File -> New File -> R Script`. This will open up a new pane, and your screen should look something like this



Let's look around here. The top left pane is called the "Source Pane". It is where you can write an R script. Try typing

```
1+1
```

in that pane. This is a very simple R program. Now, type `Ctrl+s` to save the script. This will likely prompt you to provide a name for the script. You can call it `first_script.R` or something like that. The only thing that really matters is that the file name ends in ".R" (although you should at least give the file a reasonably descriptive name).

Now let's move to the bottom left pane. This is called the "Console Pane". It is where the actual computations happen in R (Notice that, although we have already saved our first script, we haven't actually run any code). Beside the blue arrow in that pane, try typing

```
2+2
```

and then press `ENTER`. This time you should actually see the answer.

Now, let's go back to the Source pane. Often, it is convenient to run R programs line by line (mainly in order for it to be easy for you to digest the results). You can do this by pressing `Ctrl+ENTER` on any line in your script for it to run next. Try this on the first line of your script file where we previously typed `1+1`. This code should now run, and you should be able to see the result down in the bottom left Console pane.

We will ignore the two panes on the right for now and come back to them once we get a little more experience programming in R.

Installing R Packages

Related Reading: IDS 1.5

When you download `R`, you get "base" `R`. Base R contains "basic" functions that are commonly used by most R users. To give some examples, base R gives you the ability add, subtract, divide, or multiply numbers. Base R gives you the ability to calculate the mean (the function is called `mean`) or standard deviation (the function is called `sd`) of a vector of numbers.

Base R is quite powerful and probably the majority of code you will write in R will only involve Base R.

That being said, there are many cases where it is useful to expand the base functionality of `R`. This is done through **packages**. Packages expand the functionality of R. R is open source so these packages are contributed by users.

It also typically wouldn't make sense for someone to install *all* available R packages. For example, a geographer might want to install a much different set of packages relative to an economist. Therefore, we will typically install only the additional functionality that we specifically want.

Example: In this example, we'll install the `dslabs` package (which is from the IDS book) and the `lubridate` package (which is a package for working with dates in R).

```
# install dslabs package
install.packages("dslabs")

# install lubridate package
install.packages("lubridate")
```

Installing a package is only the first step to using a package. You can think of installing a package like *downloading* a package. To actually use a package, you need to load it into memory (i.e., "attach" it) or at least be clear about the package where a function that you are trying to call comes from.

Example: Dates can be tricky to work with in R (and in programming languages generally). For example, they are not exactly numbers, but they also have more structure than just a character string. The `lubridate` package contains functions for converting numbers/strings into dates.

```
bday <- "07-15-1985"
class(bday) # R doesn't know this is actually a date yet
```

```
[1] "character"
```

```
        # load the package
        library(lubridate)
        # mdy stands for "month, day, year"
        # if date were in different format, could use ymd, etc.
        date_bday <- mdy(bday)
        date_bday
```

```
[1] "1985-07-15"
```

```
        # now R knows this is a date
        class(date_bday)
```

```
[1] "Date"
```

Another (and perhaps better) way to call a function from a package is to use the `::` syntax. In this case, you do not need the call to `library` from above. Instead, you can try

```
        lubridate::mdy(bday)
```

```
[1] "1985-07-15"
```

This does exactly the same thing as the code before. What is somewhat better about this code is that it is easier to tell that the `mdy` function came from the `lubridate` package.

A list of useful R packages

- `AER` — package containing data from *Applied Econometrics with R*
- `wooldridge` — package containing data from Wooldridge's text book
- `ggplot2` — package to produce sophisticated looking plots
- `dplyr` — package containing tools to manipulate data
- `haven` — package for loading different types of data files
- `plm` — package for working with panel data
- `fixest` — another package for working with panel data
- `ivreg` — package for IV regressions, diagnostics, etc.
- `estimatr` — package that runs regressions but with standard errors that economists often like more than the default options in `R`
- `modelsummary` — package for producing nice output of more than one regression and summary statistics

As of this writing, there are currently 18,004 R packages available on CRAN (R's main repository for contributed packages).

R Basics

Related Reading: IDS 2.1

In this section, we'll start to work towards writing useful R code.

Objects

Related Reading: IDS 2.2

The very first step to writing code that can actually do something is to able to store things. In R, we store things in **objects** (perhaps sometimes I will also use the word variables).

Earlier, we used R to calculate $1 + 1$. Let's go back to the Source pane (top left pane in RStudio) and type

```
answer <- 1 + 1
```

Press `Ctrl+ENTER` on this line to run it. You should see the same line down in the Console now.

Let's think carefully about what is happening here

- `answer` is the name of the variable (or object) that we are creating here.
- the `<-` is the **assignment** operator. It means that we should *assign* whatever is on the right hand side of it to the variable that is on the left hand side of it
- `1+1` just computes $1 + 1$ as we did earlier. Soon we will put more complicated expressions here.

You can think about the above code as computing $1 + 1$ and then *saving* it in the variable `answer`.

> **Side Comment:** The assignment operator, `<-`, is a "less than sign" followed by a "hyphen". It's often convenient though to use the keyboard shortcut `Alt+-` (i.e., hold down `Alt` and press the hypen key) to insert it. You can also use an `=` for assignment, but this is less commonly done in R.

> **Practice:** Try creating variable called `five_squared` that is equal to 5×5 (multiplication in R is done using the `*` symbol).

There are a number of reasons why you might like to create an object in R. Perhaps the main one is so that you can reuse it. Let's try multiplying `answer` by 3.

```
answer*3
```

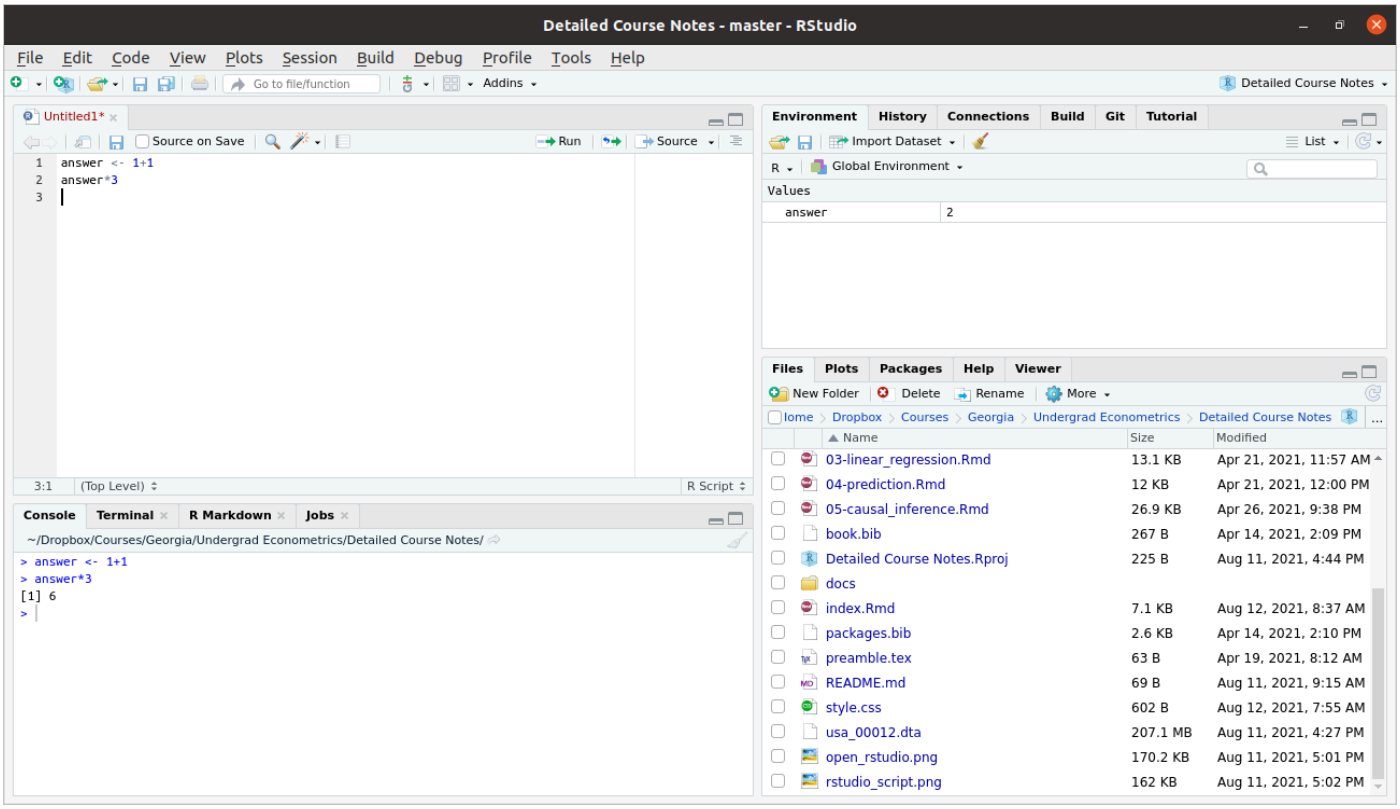
```
[1] 6
```

If you wanted, you could also save this as its own variable too.

Workspace

Related Reading: IDS 2.2

Before we move on, I just want to show you what my workspace looks like now.



As we talked about above, you can see the code in my script in the Source pane in the top left. You can also see the code that I actually ran in the Console pane on the bottom left.

Now, take a look at the top right pane. You will see under the Environment tab that `answer` shows up there with a value of `2`. The Environment tab keeps track of all the variables that you have created in your current session. A couple of other things that might be useful to point out there.

- Later on in the class, we will often import data to work with. You will notice the "Import Dataset" button that is located in this top right pane. I will suggest to you a different way of importing data in the next section, but this is also a way to do it.

- Occasionally, you might get into the case where you have saved a bunch of variables and it would be helpful to "start over". The broom in this pane will "clean" your workspace (this just means delete

everything).

Importing Data

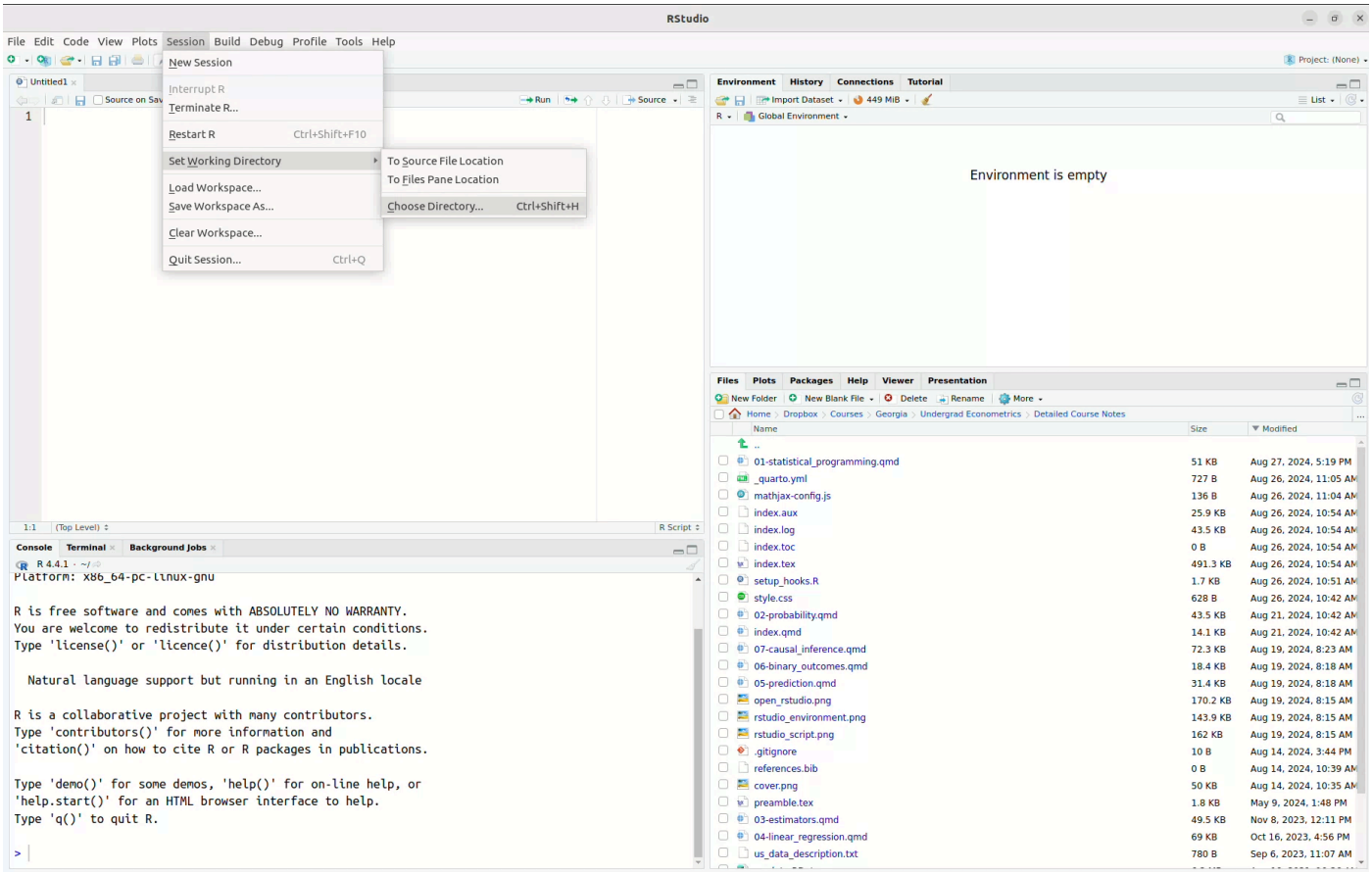
Related Reading: IDS 6

To work with actual data in R, we will need to import it. I mentioned the “Import Data” button above, but let me mention a few other possibilities here, including how to import data by writing code.

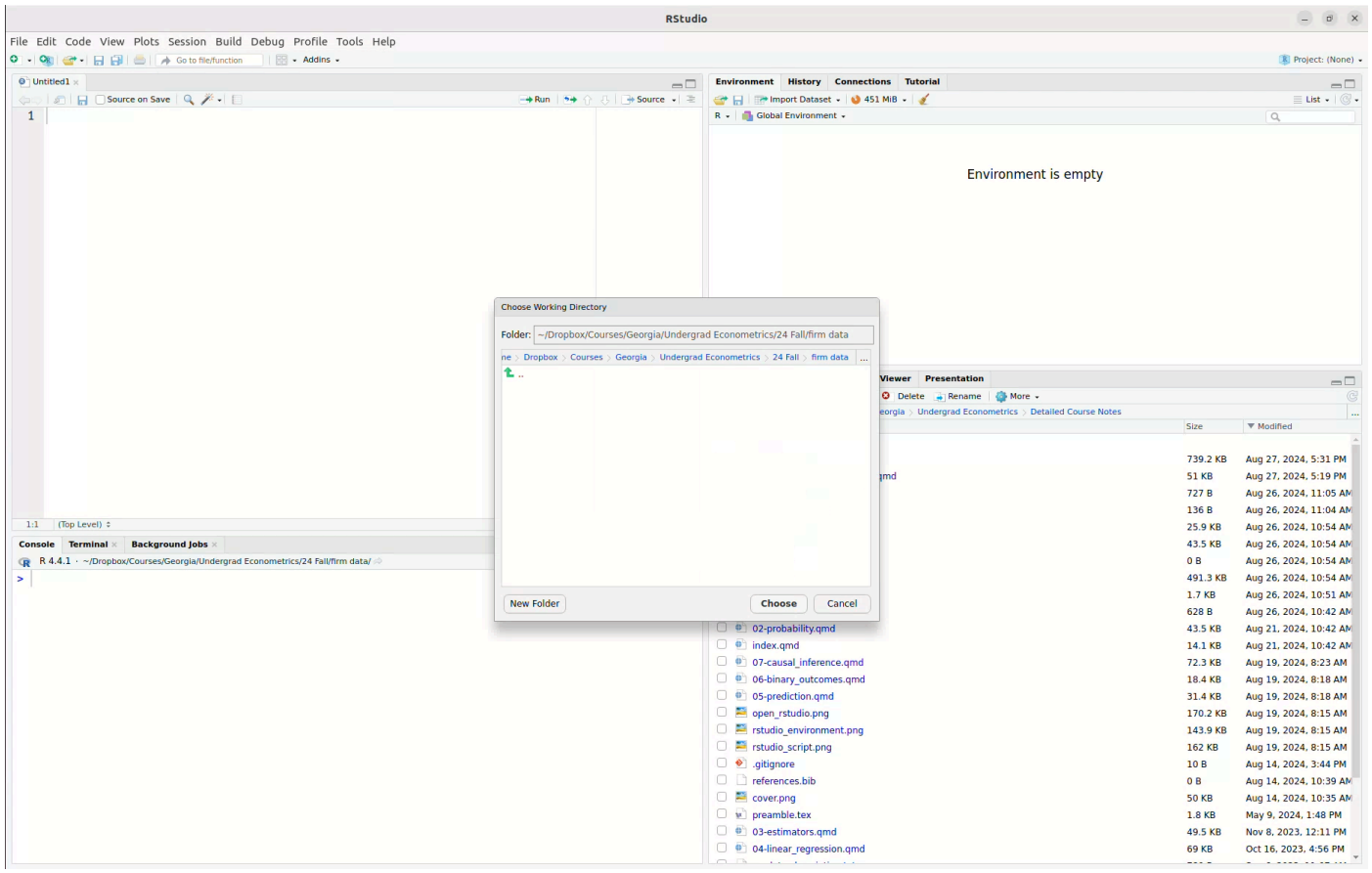
On the course website, I posted three files `firm.data.csv`, `firm_data.RData`, and `firm_data.dta`. All three of these contain exactly the same small, fictitious dataset, but are saved in different formats.

Probably the easiest way to import data in R is through the Files pane on the bottom right. But, in order to do this, you may need to change your **working directory**. We will do this using RStudio’s user interface in the following steps:

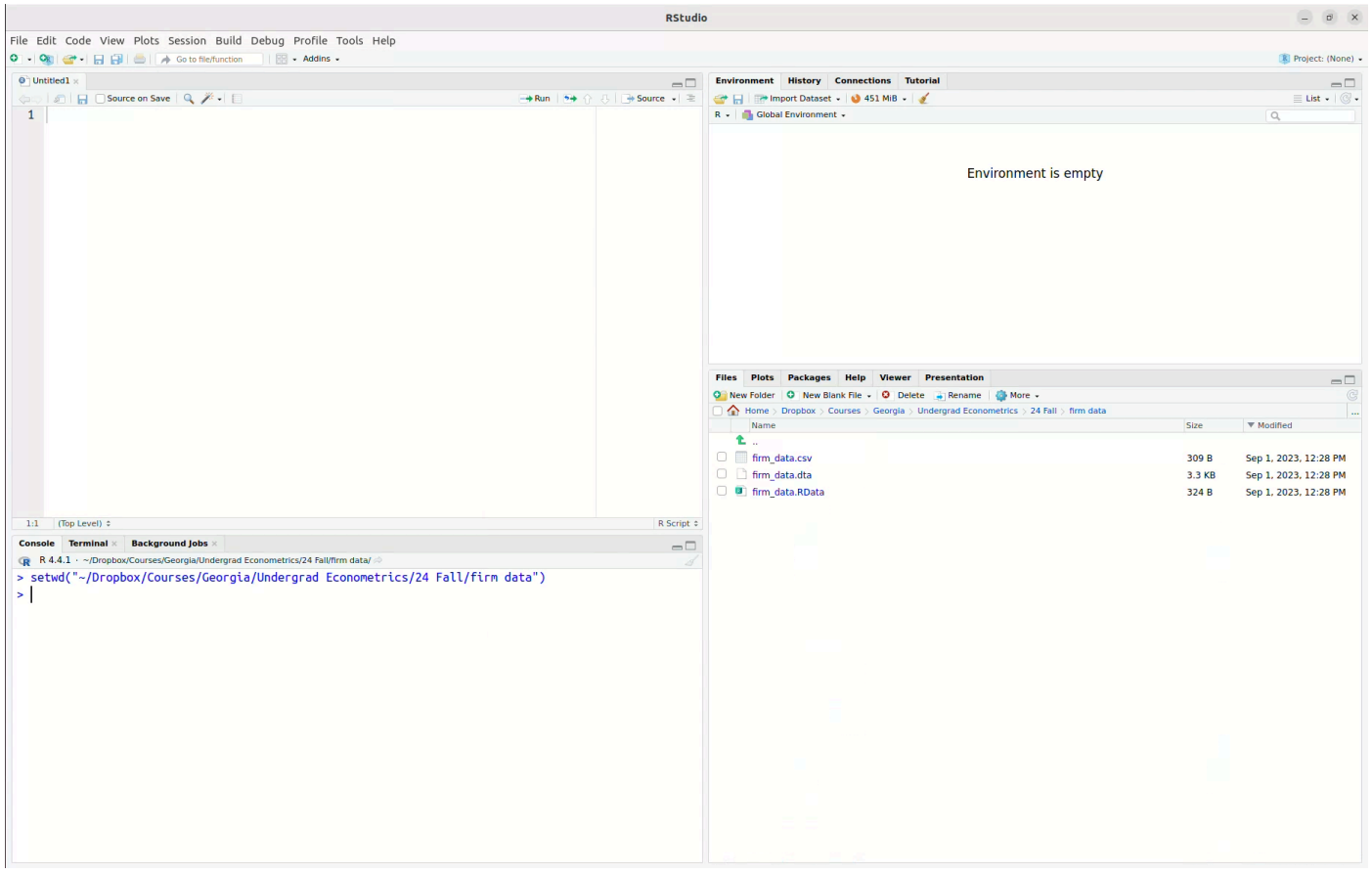
- First navigate to Sessions -> Set Working Directory -> Choose Directory. This will open a window that will allow you to choose the directory where you saved the data.



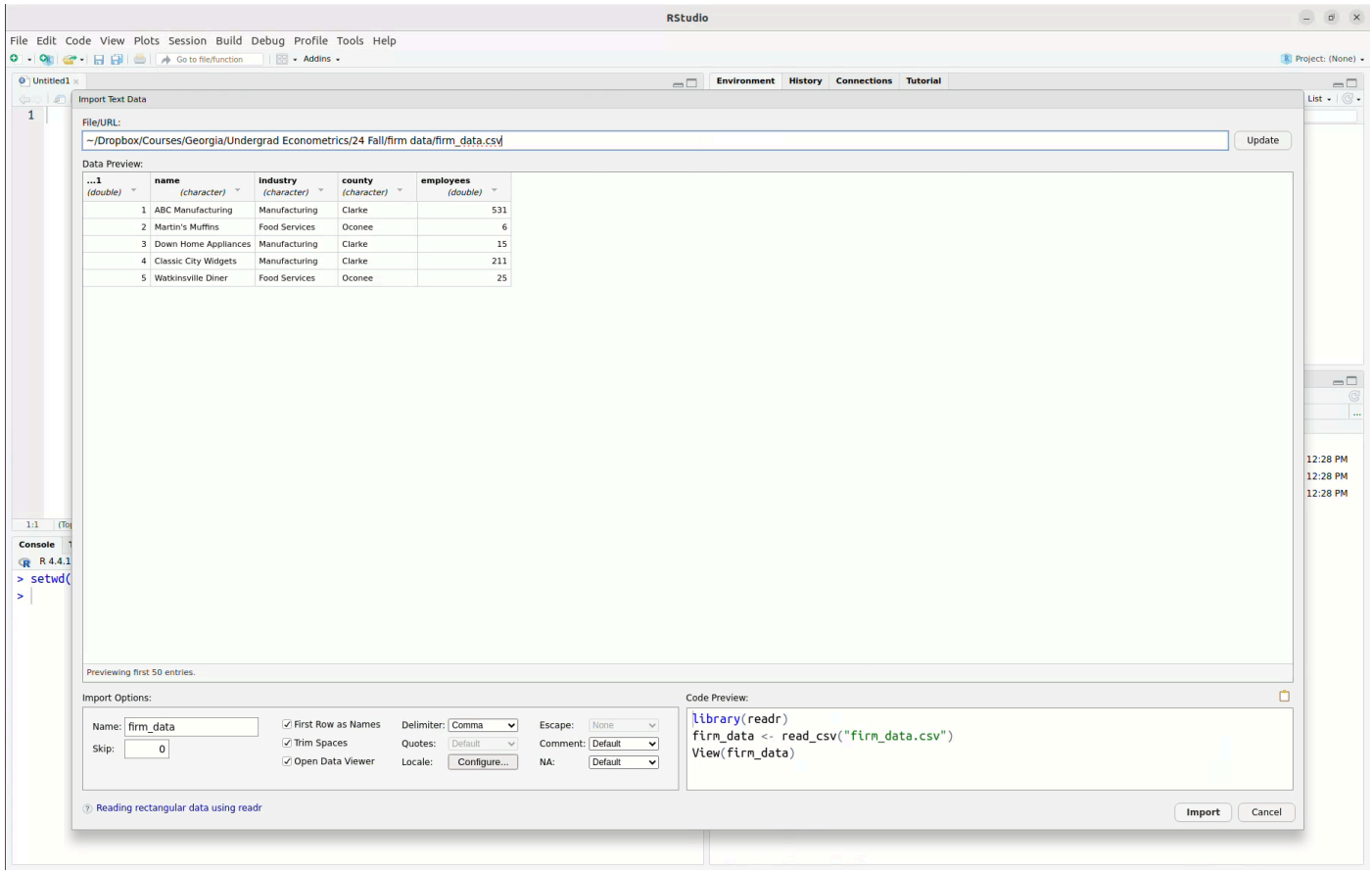
- Next, use the menu to navigate to the place where you saved `firm_data.csv`. I created a folder `~/Dropbox/Courses/Georgia/Undergrad Econometrics/24 Fall/firm data/` and saved it there.



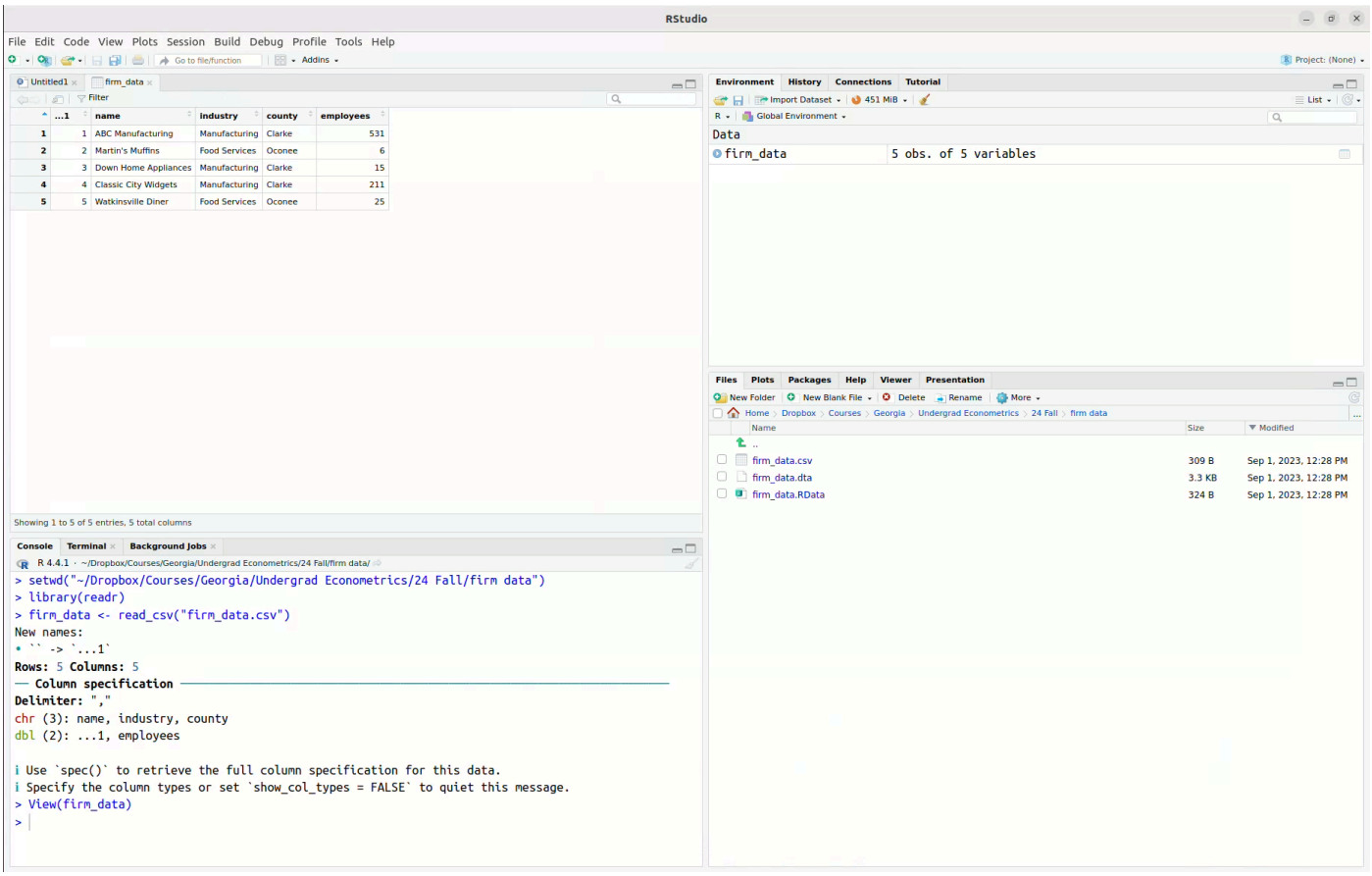
- Now, we have set the working directory, and this is what RStudio looks like for me. Notice that the working directory is now set to the folder where I saved the data. You can see the difference in the Files pane.



- Next, we will load the data, just by clicking it in the Files pane. I picked `firm_data.csv`, but any of the three files will work. `R` is quite good at recognizing different types of data files and importing them, so this same procedure will work for `firm_data.RData` and `firm_data.dta` even though they are different types of files. Once you click it, you will get a screen that should look like this



- Click “Import” and the data should be imported. You can see that it is now in the Environment pane.



Next, let's discuss how to import data by writing computer code (by the way, this is actually what is happening behind the scenes when you import data through the user interface as described above). "csv" stands for "Comma Separated Values". This is basically a plain text file (e.g., try opening it in Notepad or Text Editor) where the columns are separated by commas and the rows are separated by being on different lines. Most any computer program can read this type of file; that is, you could easily import this file into, say, R, Excel, or Stata. You can import a `.csv` file using `R` code by

```
firm_data <- read.csv("firm_data.csv")
```

An `RData` file is the native format for saving data in `R`. You can import an `RData` file using the following command:

```
firm_data <- load("firm_data.RData")
```

Similarly, a `dta` file the native format for saving data in Stata. You can import a `dta` file using the following command:

```
library(haven) # external package for reading dta file
firm_data <- read_dta("firm_data.dta")
```

In all three cases above, what we have done is to create a new `data.frame` (a `data.frame` is a type of object that we'll talk about in detail later on in this chapter) called `firm_data` that contains the data that we were trying to load.

Programming in R

Functions in R

Related Reading: IDS 3.2

R has a large number of helpful, built-in functions. Let's start with a pretty representative example: computing logarithms. This can be done using the R function `log`.

```
log(5)
```

```
[1] 1.609438
```

You can tell this is a function because of the parentheses. The `5` inside of the parentheses is called the **argument** of the function. As practice, try computing the $\log$ of 7.

> **Side Comment:** As a reminder, the logarithm of some number, let's call it b, is is the value of a that solves $\text{base}^a = b$.

The default base in R is $e \approx 2.718$, so that `log(5)` actually computes what you might be more used to calling the "natural logarithm". You can change the default value of the base by adding an extra argument to the function.

```
log(5, base=10)
```

```
[1] 0.69897
```

In order to learn about what arguments are available (and what they mean), you can access the help files for a particular function by running either

```
help(log)
?log
```

and, of course, substituting the name of whatever function you want to learn about in place of `log`.

In RStudio, it can also be helpful to press `Tab` and RStudio will provide possible completions to the function you are typing as well as what arguments can be provided to that function.

> **Practice:** R has a function for computing absolute value (you'll have to find the name of it on your own). Try computing the absolute value of 5 and -5. Try creating a variable called

`negative_three` that is equal to -3; then, try to compute the absolute value of `negative_three`.

Data types

Related Reading: IDS 2.3

Numeric Vectors

The most basic data type in `R` is the vector. In fact, above when we created variables that were just a single number, they are actually stored as a numeric vector.

To more explicitly create a vector, you can use the `c` function in `R`. For example, let's create a vector called `five` that contains the numbers 1 through 5.

```
five <- c(1,2,3,4,5)
```

We can print the contents of the vector `five` just by typing its name

```
five
```

```
[1] 1 2 3 4 5
```

Another common operation on vectors is to get a particular element of a vector. Let me give an example

```
five[3]
```

```
[1] 3
```

This code takes the vector `five` and returns the third element in the vector. Notice that the above line contains braces, `[` and `]` rather than parentheses.

If you want several different elements from a vector, you can do the following

```
five[c(1,4)]
```

```
[1] 1 4
```

This code takes the vector `five` and returns the first and fourth element in the vector.

One more useful function for vectors is the function `length`. This tells you the number of elements in vector. For example,

```
length(five)
```

```
[1] 5
```

which means that there are five total elements in the vector `five`.

Vector arithmetic

Related Reading: IDS 2.8

The main operations on numeric vectors are `+`, `-`, `*`, `/` which correspond to addition, subtraction, multiplication, and division. Often, we would like to carry out these operations on vectors.

There are two main cases. The first case is when you try to add a single number (i.e., a scalar) to all the elements in a vector. In this setup, the operation will happen element-wise which means the same number will be added to all numbers in the vector. This will be clear with some examples.

```
five <- c(1,2,3,4,5)

# adds one to each element in vector
five + 1
```

```
[1] 2 3 4 5 6
```

```
# also adds one to each element in vector
1 + five
```

```
[1] 2 3 4 5 6
```

Similar things will happen with the other mathematical operations above. Here are some more examples:

```
five * 3
```

```
[1]  3  6  9 12 15
```

```
five - 3
```

```
[1] -2 -1  0  1  2
```

```
five / 3
```

```
[1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667
```

The other interesting case is what happens when you try to apply any of the same mathematical operators to two different vectors.

```
# just some random numbers
vec2 <- c(8,-3,4,1,7)

five + vec2
```

```
[1]  9 -1  7  5 12
```

```
five - vec2
```

```
[1] -7  5 -1  3 -2
```

```
five * vec2
```

```
[1]  8 -6 12  4 35
```

```
five / vec2
```

```
[1]  0.1250000 -0.6666667  0.7500000  4.0000000  0.7142857
```

You can immediately see what happens here. For example, for `five + vec2`, the first element of `five` is added to the first element of `vec2`, the second element of `five` is added to the second element of `vec2` and so on. Similar things happen for each of the other mathematical operations too.

There's one other case that might be interesting to consider too. What happens if you try to apply these mathematical operations to two vectors of different lengths? Let's find out

```
vec3 <- c(2,6)
five + vec3
```

```
Warning in five + vec3: longer object length is not a multiple of shorter
object length
```

```
[1]  3  8  5 10  7
```

You'll notice that this computes *something* but it also issues a warning. What happens here is that the result is equal to the first element of `five` plus the first element of `vec3`, the second of `five` plus the second element of `vec3`, the third element of `five` plus *the first element of* `vec3`, the fourth element of `five` plus *the second element of* `vec3`, and the fifth element of `five` plus *the first element of* `vec3`. What's happening here is that, since `vec3` contains fewere elements that `five`, the elements of `vec3` are getting *recycled*. In my experience, this warning often indicates a coding mistake. There are many cases where I want to add the same number to all elements in a vector, and many other cases where I want to add two vectors that have the same length, but I cannot think of any cases where I would want to add two vectors the way that is being carried out here.

The same sort of things will happen with subtraction, multiplication, and division (feel free to try it out).

More helpful functions in R

This is definitely an incomplete list, but I'll point you here to some more functions in R that are often helpful along with quick examples of them.

- `seq` function — creates a "sequence" of numbers

```
seq(2,7)
```

```
[1] 2 3 4 5 6 7
```

- `sum` function — computes the sum of a vector of numbers

```
sum(c(1,5,8))
```

```
[1] 14
```

- `sort`, `order`, and `rev` functions — functions for understanding the order or changing the order of a vector

```
sort(c(3,1,5))
```

```
[1] 1 3 5
```

```
order(c(3,1,5))
```

```
[1] 2 1 3
```

```
rev(c(3,1,5))
```

```
[1] 5 1 3
```

- `%%` — modulo function (i.e., returns the remainder from dividing one number by another)

```
8 %% 3
```

```
[1] 2
```

```
1 %% 3
```

```
[1] 1
```

Practice: The function `seq` contains an optional argument `length.out`. Try running the following code and seeing if you can figure out what `length.out` does.

```
seq(1,10,length.out=5)
seq(1,10,length.out=10)
seq(1.10,length.out=20)
```

Other types of vectors

There are other types of vectors in R too. Probably the main two other types of vectors are **character vectors** and **logical vectors**. We'll talk about character vectors here and defer logical vectors until later. Character vectors are often referred to as **strings**.

We can create a character vector as follows

```
string1 <- "econometrics"
string2 <- "class"
string1
```

```
[1] "econometrics"
```

The above code creates two character vectors and then prints the first one.

> **Side Comment** `c` stands for "concatenate". Concatenate is a computer science word that means to combine two vectors. Probably the most well known version of this is "string concatenation" that combines two vectors of characters. Here is an example of string concatenation.
>
> ```
> c(string1, string2)
> ```
>
> ```
> [1] "econometrics" "class"
> ```
>
> Sometimes string concatenation means to put two (or more strings) into the same string. This can be done using the `paste` command in R.
>
> ```
> paste(string1, string2)
> ```
>
> ```
> [1] "econometrics class"
> ```
>
> Notice that `paste` puts in a space between `string1` and `string2`. For practice, see if you can find an argument to the `paste` function that allows you to remove the space between the two strings.

Data Frames

Another very important type of object in R is the **data frame**. I think it is helpful to think of a data frame as being very similar to an Excel spreadsheet — sort of like a matrix or a two-dimensional array. Each row typically corresponds to a particular observation, and each column typically provides the value of a particular variable for that observation.

Just to give a simple example, suppose that we had firm-level data about the name of the firm, what industry a firm was in, what county they were located in, and their number of employees. I created a data

frame like this (it is totally made up, BTW) and show it to you next

```
        firm_data
```

name	industry	county	employees
ABC Manufacturing	Manufacturing	Clarke	531
Martin's Muffins	Food Services	Oconee	6
Down Home Appliances	Manufacturing	Clarke	15
Classic City Widgets	Manufacturing	Clarke	211
Watkinsville Diner	Food Services	Oconee	25

Side Comment: If you are following along on R, I created this data frame using the following code

```
        firm_data <- data.frame(name=c("ABC Manufacturing",
                                       "Martin\'s Muffins",
                                       "Down Home Appliances",
                                       "Classic City Widgets",
                                       "Watkinsville Diner"),
                                industry=c("Manufacturing",
                                           "Food Services",
                                           "Manufacturing",
                                           "Manufacturing",
                                           "Food Services"),
                                county=c("Clarke",
                                         "Oconee",
                                         "Clarke",
                                         "Clarke",
                                         "Oconee"),
                                employees=c(531, 6, 15, 211, 25))
```

This is also the same data that we loaded earlier in Section 2.3.

Often, we'll like to access a particular column in a data frame. For example, you might want to calculate the average number of employees across all the firms in our data.

Typically, the easiest way to do this, is to use the **accessor** symbol, which is `$` in R. This will make more sense with an example:

```
        firm_data$employees
```

```
[1] 531   6  15 211  25
```

`firm_data$employees` just provides the column called "employees" in the data frame called "firm_data". You can also notice that `firm_data$employees` is just a numeric vector. This means that you can apply any of the functions that we have been covering on it

```
        mean(firm_data$employees)
```

```
[1] 157.6
```

```
        log(firm_data$employees)
```

```
[1] 6.274762 1.791759 2.708050 5.351858 3.218876
```

Side Comment: Notice that the function `mean` and `log` behave differently. `mean` calculates the average over all the elements in the vector `firm_data$employees` and therefore returns a single number. `log` calculates the logarithm of each element in the vector `firm_data$employees` and therefore returns a numeric vector with five elements.

Side Comment:

The `$` is not the only way to access the elements in a data frame. You can also access them by their position. For example, if you want whatever is in the third row and second column of the data frame, you can get it by

```
        firm_data[3,2]
```

```
[1] "Manufacturing"
```

Sometimes it is also convenient to recover a particular row or column by its position in the data frame. Here is an example of recovering the entire fourth row

```
        firm_data[4,]
```

```
                  name      industry county employees
4 Classic City Widgets Manufacturing Clarke       211
```

Notice that you just leave the "column index" (which is the second one) blank

Side Comment: One other thing that sometimes takes some getting used to is that, for programming in general, you have to be very precise. Suppose you were to make a very small typo. R is not going to understand what you mean. See if you can spot the typo in the next line of code.

```
        firm_data$employes
```

```
NULL
```

A few more useful functions for working with data frames are:

- `nrow` and `ncol` — returns the number of rows or columns in the data frame
- `colnames` and `rownames` — returns the names of the columns or rows

Lists

Vectors and data frames are the main two types of objects that we'll use this semester, but let me give you a quick overview of a few other types of objects. Let's start with **lists**. Lists are very generic in the sense that they can carry around complicated data. If you are familiar with any object oriented programming language like Java or C++, they have the flavor of an "object", in the object-oriented sense.

I'm not sure if we will see any examples this semester where you *have* to use a list. But here is an example. Suppose that we wanted to put the vector that we created earlier `five` and the data frame that we created earlier `firm_data` into the same object. We could do it as follows

```
        unusual_list <- list(numbers=five, df=firm_data)
```

You can access the elements of a list in a few different ways. Sometimes it is convenient to access them via the `$`

```
        unusual_list$numbers
```

```
[1] 1 2 3 4 5
```

Other times, it is convenient to access them via their position in the list

```
        unusual_list[[2]] # notice the double brackets
```

```
                  name      industry county employees
1    ABC Manufacturing Manufacturing Clarke       531
2     Martin's Muffins Food Services Oconee         6
3 Down Home Appliances Manufacturing Clarke        15
4 Classic City Widgets Manufacturing Clarke       211
5   Watkinsville Diner Food Services Oconee        25
```

Matrices

Matrices are very similar to data frames, but the data should all be of the same type. Matrices are very useful in some numerical calculations that are beyond the scope of this class. Here is an example of a matrix.

```
mat <- matrix(c(1,2,3,4), nrow=2, byrow=TRUE)
mat
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

You can access elements of a matrix by their position in the matrix, just like for the data frame above.

```
# first row, second column
mat[1,2]
```

```
[1] 2
```

```
# all rows in second column
mat[,2]
```

```
[1] 2 4
```

Factors

Sometimes variables in economics are **categorical**. This sort of variable is somewhat between a numeric variable and a string. In `R`, categorical variables are called **factors**.

A good example of a categorical variable is `firm_data$industry`. It tells you the "category" of the industry that a firm is in.

Oftentimes, we may have to tell R that a variable is a "factor" rather than just a string. Let's create a variable called `industry` that contains the industry from `firm_data` but as a factor.

```
industry <- as.factor(firm_data$industry)
industry
```

```
[1] Manufacturing Food Services Manufacturing Manufacturing Food Services
Levels: Food Services Manufacturing
```

A useful package for working with factor variables is the `forcats` package.

Understanding an object in R

Sometimes you may be in the case where there is a variable where you don't know what exactly it contains. Some functions that are helpful in this case are

- `class` — tells you, err, the class of an object (i.e., its "type")
- `head` — shows you the "beginning" of an object; this is especially helpful for large objects (like some data frames)
- `str` — stands for "structure" of an object

Let's try these out

```
        class(firm_data)
```

```
[1] "data.frame"
```

```
        # typically would show the first five rows of a data frame,
        # but that is the whole data frame here
        head(firm_data)
```

```
                  name      industry county employees
1    ABC Manufacturing Manufacturing Clarke       531
2     Martin's Muffins Food Services Oconee         6
3 Down Home Appliances Manufacturing Clarke        15
4 Classic City Widgets Manufacturing Clarke       211
5   Watkinsville Diner Food Services Oconee        25
```

```
        str(firm_data)
```

```
'data.frame':   5 obs. of  4 variables:
 $ name     : chr  "ABC Manufacturing" "Martin's Muffins" "Down Home Appliances" "Classic
City Widgets" ...
 $ industry : chr  "Manufacturing" "Food Services" "Manufacturing" "Manufacturing" ...
 $ county   : chr  "Clarke" "Oconee" "Clarke" "Clarke" ...
 $ employees: num  531 6 15 211 25
```

> **Practice:** Try running `class`, `head`, and `str` on the vector `five` that we created earlier.

Logicals

Related Reading: IDS 2.9

All programming languages have ways of tracking whether variables meet certain criteria. These are often called Booleans or Logicals. For us, this will particularly come up in the context of subsetting data (i.e., selecting data based on some condition) and in running particular portions of code based on some condition.

Some main logical operators are `==`, `<=`, `>=`, `<`, `>` corresponding to whether or not two things are equal, less than or equal to, greater than or equal, strictly less than, and strictly greater than. These can be applied to vectors. And the comparisons result in either `TRUE` or `FALSE`. Here are some examples

```
five <- c(1,2,3,4,5)

# only 3 is equal to 3
five == 3
```

```
[1] FALSE FALSE  TRUE FALSE FALSE
```

```
# 1,2,3 are all less than or equal to 3
five <= 3
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE
```

```
# 3,4,5, are all greater than or equal to 3
five >= 3
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

```
# 1,2 are strictly less than 3
five < 3
```

```
[1]  TRUE  TRUE FALSE FALSE FALSE
```

```
# 4,5 are strictly greater than 3
five > 3
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE
```

Example: Often, we might be interested in learning about a subset of our data. As a simple example, using our `firm_data` from earlier, you could imagine being interested in average employment for manufacturing firms.

We can do this using the `subset` function along with the logical operations we've learned in this section.

```
manufacturing_firms <- subset(firm_data, industry=="Manufacturing")
mean(manufacturing_firms$employees)
```

```
[1] 252.3333
```

As practice, try creating a subset of `firm_data` based on firms having more than 100 employees.

Additional Logical Operators

Related Reading: IDS 2.9

There are a number of additional logical operators that can be useful in practice. Here, we quickly cover several more.

- `!=` — not equal

```
c(1,2,3) != 3
```

```
[1]  TRUE  TRUE FALSE
```

- We can link together multiple logical comparisons. If we want to check whether multiple conditions hold, we can use "logical AND" `&`; if we want to check whether any of multiple conditions hold, we can use "logical OR" `|`.

```
# AND
( c(1,2,3,4,5) >= 3 ) & ( c(1,2,3,4,5) < 5 )
```

```
[1] FALSE FALSE  TRUE  TRUE FALSE
```

```
# OR
( c(1,2,3,4,5) >= 4 ) | ( c(1,2,3,4,5) < 2 )
```

```
[1]  TRUE FALSE FALSE  TRUE  TRUE
```

- `%in%` — checks whether the elements of one vector show up *in* another vector

```
# 1 is in the 2nd vector, but 7 is not
c(1,7) %in% c(1,2,3,4,5)
```

```
[1]  TRUE FALSE
```

- Often it is useful to check whether any logical conditions are true or all logical conditions are true. This can be done as follows

```
# this one is TRUE because 1 is in the 2nd vector
any(c(1,7) %in% c(1,2,3,4,5))
```

```
[1] TRUE
```

```
# this one is FALSE because 7 is not in the 2nd vector
all(c(1,7) %in% c(1,2,3,4,5))
```

```
[1] FALSE
```

Programming basics

Writing functions

Related Reading: IDS 3.2

It is often helpful to write your own functions in R. If you ever find yourself repeating the same code over and over, this suggests that you should write this code as a function and repeatedly call the function.

Suppose we are interesting in solving the quadratic equation

$$ax^2 + bx + c = 0$$

If you remember the quadratic formula, the solution to this equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

It would be tedious to calculate this by hand (especially if we wanted to calculate it for many different values of a, b, and c), so let's write a function to do it.

```
quadratic_solver <- function(a, b, c) {
  root1 <- ( -b + sqrt(b^2 - 4*a*c) ) / 2*a
  root1
}
```

Before we try this out, let's notice a few things. First, while this particular function is for solving the quadratic equation, this is quite representative of what a function looks like in R.

- `quadratic_solver` — This is the name of the function. It's good to give your function a descriptive name related to what it does. But you could call it anything you want. If you wanted to call this function `uga`, it would still work.
- the part `<- function` finishes off assigning the function the name `quadratic_solver` and implies that we are writing down a function rather than a `vector` or `data.frame` or something else. This part will show up in all function definitions.

- the part `(a, b, c)`, `a`, `b`, and `c` are the names of the *arguments* to the function. In a minute when we call the function, we need to tell the function the particular values of `a`, `b`, and `c` for which to solve the quadratic equation. We could name these whatever we want, but, again, it is good to have descriptive names. When you write a different function, it can have as many arguments as you want it to have.
- the part `{ ... }` everything that the function does should go between the curly brackets
- the line `root1 <- ( -b + sqrt(b^2 - 4*a*c) ) / 2*a` contains the main thing that is calculated by our function. Notice that we only calculate one of the "roots" (i.e., solutions to the quadratic equation) because of the $+$ in this expression.
- the line `root1` R returns whatever variable is on the last line of the function. It might be somewhat more clear to write `return(root1)`. The behavior of the code would be exactly the same, but it is just the more common "style" in R programming to not include the explicit `return`.

Now let's try out our function

```
        # solves quadratic equation for a=1, b=4, c=3
        quadratic_solver(1,4,3)
```

```
[1] -1
```

```
        # solves quadratic equation for a=-1, b=5, c=10
        quadratic_solver(-1,5,10)
```

```
[1] -1.531129
```

Two last things that are worth pointing out about functions:

- Functions in R can be set up to take default values for some of their arguments
- Because the arguments have names, if you are explicit about the name of the argument, then the order of the argument does not matter.

To give examples, let's write a slightly modified version of our function to solve quadratic equations.

```
        quadratic_solver2 <- function(a=1, b, c) {
          root1 <- ( -b + sqrt(b^2 - 4*a*c) ) / 2*a
          root1
        }
```

The only thing different here is that `a` takes the default value of 1. Now let's try some different calls to `quadratic_solver` and `quadratic_solver2`

```
        # solve again for a=1,b=4,c=3
        quadratic_solver2(b=4,c=3)
```

```
[1] -1
```

```
        # replace default and change order
        quadratic_solver2(c=10,b=5,a=-1)
```

```
[1] -1.531129
```

```
        # no default set for quadratic_solver so it will crash if a not provided
        quadratic_solver(b=4,c=3)
```

```
Error in quadratic_solver(b = 4, c = 3): argument "a" is missing, with no default
```

if/else

Related Reading: IDS 3.1

Often when writing code, you will want to do different things depending on some condition. Let's write a function that takes in the number of employees that are in a firm and prints "large" if the firm has more than 100 employees and "small" otherwise.

```
        large_or_small <- function(employees) {
          if (employees > 100) {
            print("large")
          } else {
            print("small")
          }
        }
```

I think, at this point, this code should make sense to you. The only new thing is the if/else. The following is not code that will actually run but is just to help understand the logic of if/else.

```
        if (condition) {
          # do something
        } else {
          # do something else
        }
```

All that happens with if/else is that we check whether `condition` evaluate to `TRUE` or `FALSE`. If it is `TRUE`, the code will do whatever is inside the first set of brackets; if it is `FALSE`, the code will do whatever is in the set of brackets following `else`.

for loops

Related Reading: IDS 3.4

Often, we need to run the same code over and over again. A `for` loop is a main programming tool for this case (`for` loops show up in pretty much all programming languages).

We'll have more realistic examples later on in the semester, but we'll do something trivial for now.

```
        out <- c()
        for (i in 1:10) {
          out[i] <- i*3
        }
        out
```

```
 [1]  3  6  9 12 15 18 21 24 27 30
```

The above code, starts with $i = 1$, calculates $i * 3$ (which is 3), and then stores that result in the first element of the vector `out`, then i increases to 2, the code calculates $i * 3$ (which is now 6), and stores this result in the second element of `out`, and so on through $i = 10$.

Vectorization

Related Reading: IDS 3.5

Vectorizing functions is a relatively advanced topic in R programming, but it is an important one, so I am including it here.

Because we will often be working with data, we will often be performing the same operation on all of the observations in the data. For example, suppose that you wanted to take the logarithm of the number of employees for all the firms in `firm_data`. One way to do this is to use a `for` loop, but this code would be a bit of a mess. Instead, the function `log` is **vectorized** — this means that if we apply it to a vector, it will calculate the logarithm of each element in the vector. Besides this, vectorized functions are often faster than `for` loops.

Not all functions are vectorized though. Let's go back to our function earlier called `large_or_small`. This took in the number of employees at a firm and then printed "large" if the firm had more than 100 employees and "small" otherwise. Let's see what happens if we call this function on a vector of employees (Ideally, we'd like the function to be applied to each element in the vector).

```
        employees <- firm_data$employees
        employees
```

```
[1] 531   6  15 211  25
```

```
        large_or_small(employees)
```

```
Error in if (employees > 100) {: the condition has length > 1
```

This is not what we wanted to have happen. Instead of determining whether each firm was large or small, we get an error basically said that something may be going wrong here. What's going on here is that the function `large_or_small` is not vectorized.

In order to vectorize a function, we can use one of a number of "apply" functions in R. I'll list them here

- `sapply` — this stands for “simplify” apply; it “applies” the function to all the elements in the vector or list that you pass in and then tries to “simplify” the result
- `lapply` — stands for “list” apply; applies a function to all elements in a vector or list and then returns a list
- `vapply` — stands for “vector” apply; applies a function to all elements in a vector or list and then returns a vector
- `apply` — applies a function to either the rows or columns of a matrix-like object (i.e., a matrix or a data frame) depending on the value of the argument `MARGIN`

Let’s use `sapply` to vectorize `large_or_small`.

```
large_or_small_vectorized <- function(employees_vec) {
  sapply(employees_vec, FUN = large_or_small)
}
```

All that this will do is call the function `large_or_small` for each element in the vector `employees`. Let’s see it in action

```
large_or_small_vectorized(employees)
```

```
[1] "large"
[1] "small"
[1] "small"
[1] "large"
[1] "small"

[1] "large" "small" "small" "large" "small"
```

This is what we were hoping for.

> **Side Comment:** I also typically replace most all `for` loops with an `apply` function. In most cases, I don’t think there is much of a performance gain, but the code seems easier to read (or at least more concise).
>
> Earlier we wrote a function to take a vector of numbers from 1 to 10 and multiply all of them by 3. Here’s how you could do this using `sapply`
>
> ```
> sapply(1:10, function(i) i*3)
> ```
>
> ```
>  [1]  3  6  9 12 15 18 21 24 27 30
> ```
>
> which is considerably shorter.

One last thing worth pointing out though is that multiplication is already vectorized, so you don't actually need to do `sapply` or the `for` loop; a better way is just

```
(1:10)*3
```

```
 [1]  3  6  9 12 15 18 21 24 27 30
```

Side Comment: A relatively popular alternative to `apply` functions are `map` functions provided in the `purrr` package.

Side Comment: It's often helpful to have a vectorized version of if/else. In `R`, this is available in the function `ifelse`. Here is an alternative way to vectorize the function `large_or_small`:

```
large_or_small_vectorized2 <- function(employees_vec) {
  ifelse(employees_vec > 100, "large", "small")
}
large_or_small_vectorized2(firm_data$employees)
```

```
[1] "large" "small" "small" "large" "small"
```

Here you can see that `ifelse` makes every comparison in its first argument, and then returns the second element for every `TRUE` coming from the first argument, and returns the third element for every `FALSE` coming from the first argument.

`ifelse` also works with vectors in the second and third element. For example:

```
ifelse(c(1,3,5) < 4, yes=c(1,2,3), no=c(4,5,6))
```

```
[1] 1 2 6
```

which picks up 1 and 2 from the second (`yes`) argument and 6 from the third (`no`) argument.

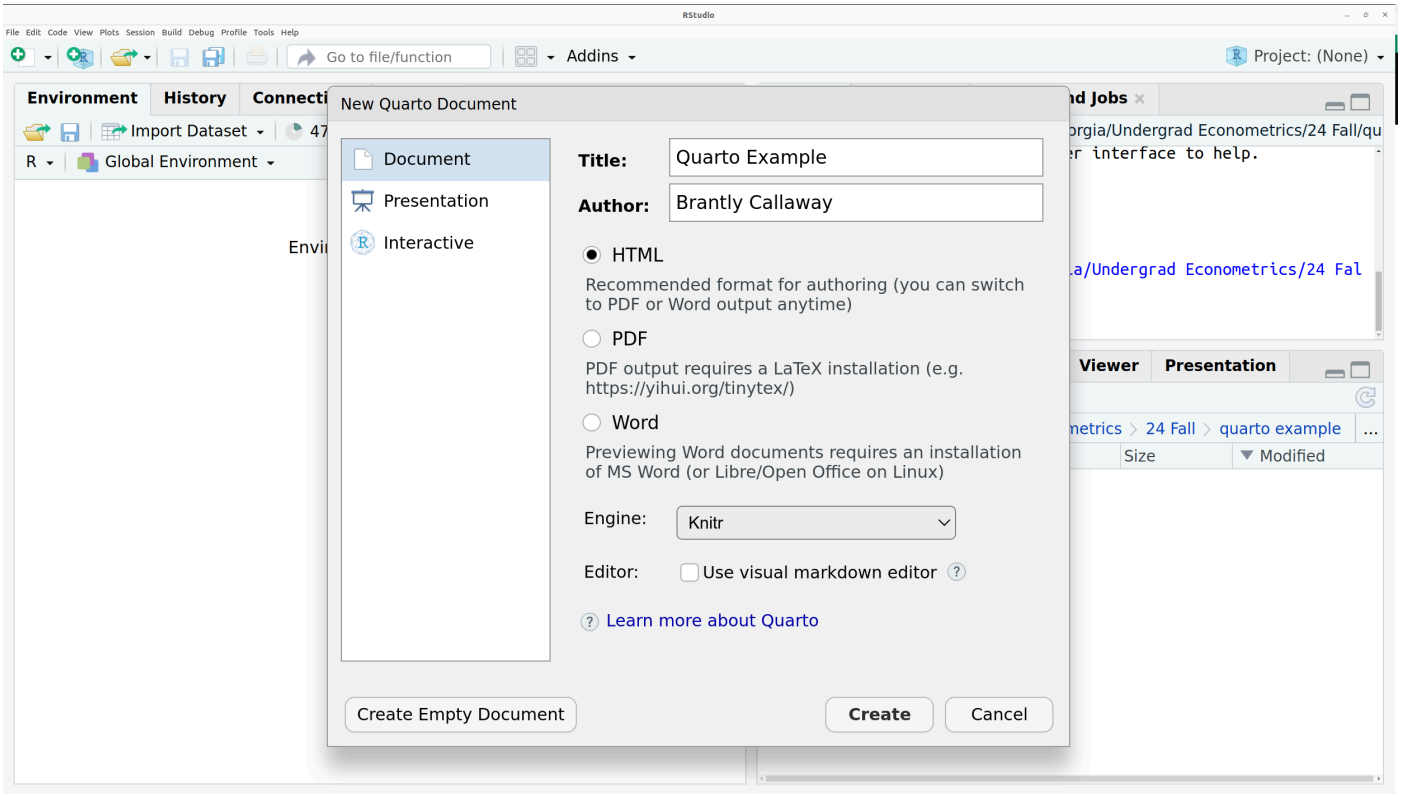
Reproducible Research

Related Reading: IDS Ch. 20

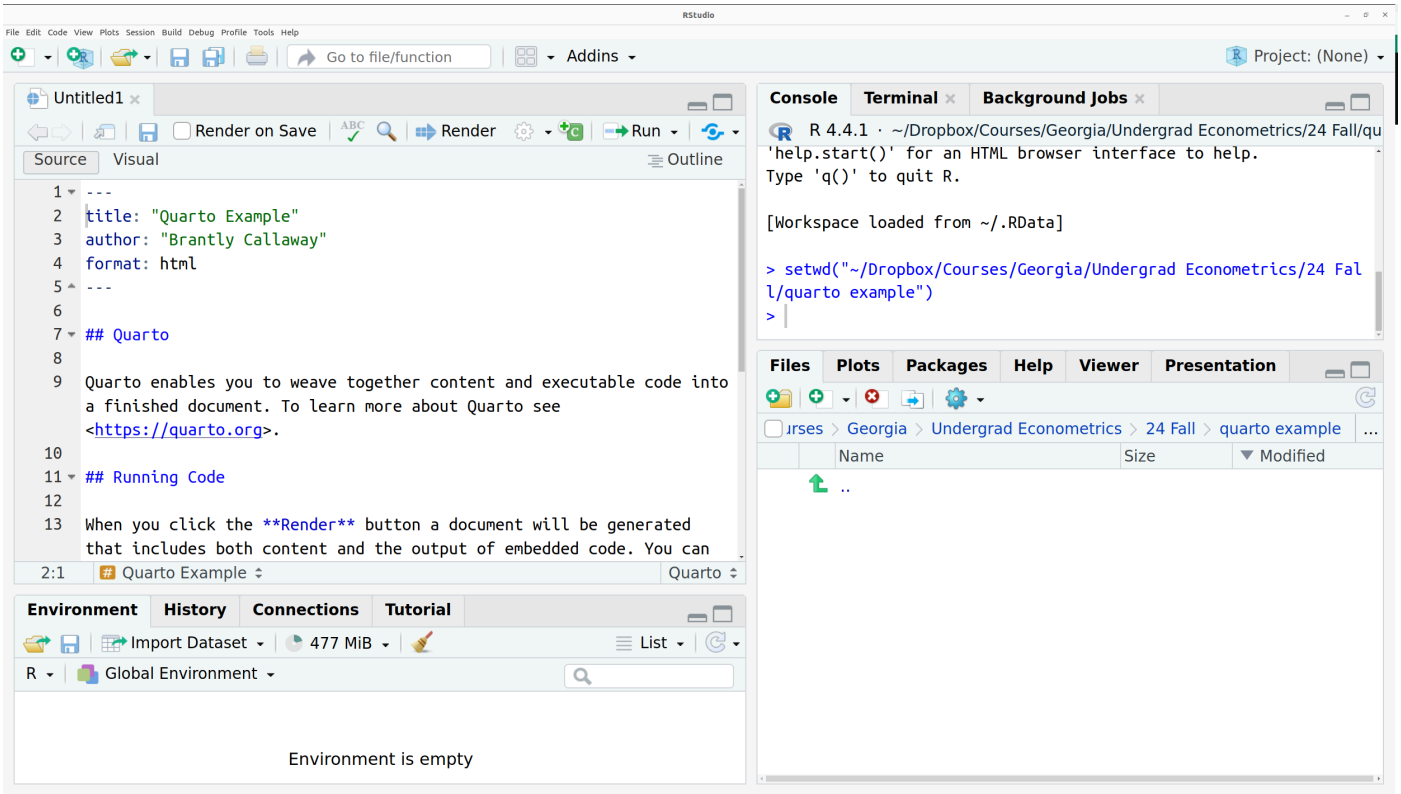
R has very useful tools for mix code and writing to produce reports (or, for our purpose, homeworks and project).

As of this writing, the most common way to do this seems to be switching from Rmarkdown to Quarto. I am going to explain how to create a Quarto document below, but, if you are familiar with Rmarkdown, the workflow is going to be very similar.

To create a new Quaro document, click `File -> New File -> Quarto Document`. That will open up a menu that looks like the following



You will need to set a few options. I set the title to be `Quarto Example` and author to be `Brantly Callaway`. I also unchecked the box at the bottom that says “Use visual markdown editor” [I prefer this setting to be unchecked, but you can try it both ways and see what you like.] Now click “Create” and your screen will look something like this



When you save a Quarto file, you should save it with a `.qmd` extension.

Here is a quick example of how you could use Quarto to write homework solutions. Suppose the first homework of the semester asked you to write a function called `sum10` that took in a vector of numbers and calculated the sum of the first 10 numbers in the vector. Then, report `sum10(1:100)`.

````
---
title: "Homework 1"
author: "Brantly Callaway"
format: html
---

## Question 1

To calculate the sum of the first ten numbers in a vector,
I wrote the following function:

```{r}
sum10 <- function(x) {
  # get first 10 elements of x
  x10 <- x[1:10]
  # calculate their sum and return it
  sum(x10)
}

sum10(1:100)
```
````

A few last comments for you:

- These notes are written in Quarto, and I write homework solutions in Quarto too.
- If you are interested, you can view the source for this book at http://github.com/bcallaway11/econ_4750_notes. The source code for this chapter is in the file `02-programming_in_R.qmd`.

Advanced Topics

To conclude this section, I want to briefly point you towards some advanced material. We will probably brush up against some of this material this semester. That being said, R has some very advanced capabilities related to data science, data manipulation, and data visualization. If you have time/interest you might push further in all of these directions. By the end of the semester, we may not have mastered these topics, but they should at least be accessible to you.

Tidyverse

Related Reading: IDS Chapter 4 — strongly recommend that you read this

• R has very good data cleaning / manipulating tools

- Many of them are in the "tidyverse"
- Mostly this semester, I'll just give you a data set that is ready to be worked with. But as you move to your own research projects or do work for a company one day, you will realize that a major step in analyzing data is organizing ("cleaning") the data in a way that you can analyze it

• Main packages

- `ggplot2` – see below
- `dplyr` — package to manipulate data
- `tidyr` — more ways to manipulate data
- `readr` — read in data
- `purrr` — alternative versions of `apply` functions and `for` loops
- `tibble` — alternative versions of `data.frame`
- `stringr` — tools for working with strings
- `forcats` — tools for working with factors

• If you see code that uses the pipe operator `%>%`, it is tidyverse-style code. [You need to load a package to get access to the pipe function. I think this was introduced in the `magrittr` package, but you can also load it with the `dplyr` package, which is one of the main tidyverse packages.] This is unusual syntax for

most programming languages, but it is (arguably) easier to read. Basically the pipe operator takes the result from one line of code and "pipes" it into the first argument of the next function. Here is an example

```
library(dplyr) # or library(magrittr)
firm_data %>%
  subset(employees > 100) %>%
  nrow()
```

```
[1] 2
```

What the above code does is it takes the data frame `firm_data`, subsets it to firms that have more than 100 rows, and calculates the number of rows in this subset (i.e., the number of large firms).

It is equivalent to the following, more traditional-looking code:

```
large_firms <- subset(firm_data, employees > 100)
nrow(large_firms)
```

```
[1] 2
```

• I won't emphasize the tidyverse too much as I prefer (at least to some extent) writing code with a more traditional syntax. That said, tidyverse packages are really quite useful for data cleaning / wrangling. And, if you are interested, these are good (and marketable) skills to have.

Data Visualization

Related Reading: IDS Ch. 7-10 — `R` has very good data visualization tools. I strongly recommend that you read this.

- Another very strong point of `R`
- Base `R` comes with the `plot` command, but the `ggplot2` package provides cutting edge plotting tools. These tools will be somewhat harder to learn, but we'll use `ggplot2` this semester as I think it is worth it.
- You can produce professional quality plots in `R` that are publication ready

We will use `ggplot2` this semester, but I will save a longer discussion for later.

Version Control

Related Reading: IDS Ch. 19

If you are interested, GitHub is a very useful version control tool (i.e., keeps track of the version of your project, useful for merging projects, and sharing or co-authoring code) and Dropbox (also useful for sharing code). I use both of these extensively — in general, I use GitHub relatively more for bigger projects and more public projects and Dropbox more for smaller projects and early versions of projects.

RStudio Projects

Related Reading: IDS 20.1

You can create a new project by navigating to `File -> New Project`. Projects in RStudio give a way to organize your, well… projects. For this course, you don't necessarily need to use projects, but you could, for example, create separate projects for each of your homeworks. This would give you separate environments for each homework (so you don't have to worry about accidentally using the same variable name across homeworks leading to any issues) and a separate set of tabs for scripts in each project. Your current project is listed in the very top right corner of the RStudio workspace.

Technical Writing Tools

This is starting to get beyond the scope of the course, but, especially for students in ECON 6750, I recommend that you look up LaTeX. This is a markup language mainly for technical, academic writing. The big payoff is on writing mathematical equations. The equations in the Course Notes are written in LaTeX. For example, the LaTeX code for the solution to the quadratic equation written above is

```
$$
  x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}
$$
```

where the `$$` is a delimiter that tells LaTeX to render the text between the delimiters as an equation, `\frac` is a command that tells LaTeX to render the text as a fraction, and `\pm` is a command that tells LaTeX to render the text as a plus or minus sign.

As I mentioned above, the course notes are written in `quarto`, but it is possible to write entire documents in LaTeX. For example, all of my academic papers are written in pure LaTeX. An easy way to get started is to use the website Overleaf. This is a website that allows you to write LaTeX documents in your web browser. Writing homework solutions fully in LaTex would be overkill for this course, but (especially if you are thinking about doing a Ph.D. in economics), it would be a good thing to poke around with as you have time.

Lab 1: Introduction to R Programming

For this lab, we will do several practice problems related to programming in R.

1. Create two vectors as follows

   ```
   x <- seq(2,10,by=2)
   y <- c(3,5,7,11,13)
   ```

   Add `x` and `y`, subtract `y` from `x`, multiply `x` and `y`, and divide `x` by `y` and report your results.

2. The geometric mean of a set of numbers is an alternative measure of central tendency to the more common "arithmetic mean" (this is the mean that we are used to). For a set of J numbers, $x_1, x_2, \ldots, x_J$, the geometric mean is defined as

$$(x_1 \cdot x_2 \cdot \cdots \cdot x_J)^{1/J}$$

Write a function called `geometric_mean` that takes in a vector of numbers and computes their geometric mean. Compute the geometric mean of `c(10,8,13)`

3. Use the `lubridate` package to figure out how many days elapsed between Jan. 1, 1981 and Jan. 10, 2022.

4. `mtcars` is one of the data frames that comes packaged with base R.

    a. How many observations does `mtcars` have?

    b. How many columns does `mtcars` have?

    c. What are the names of the columns of `mtcars`?

    d. Print only the rows of `mtcars` for cars that get at least 20 mpg

    e. Print only the rows of `mtcars` that get at least 20 mpg and have at least 100 horsepower (it is in the column called `hp`)

    f. Print only the rows of `mtcars` that have 6 or more cylinders (it is in the column labeld `cyl`) or at least 100 horsepower

    g. Recover the 10th row of `mtcars`

    h. Sort the rows of `mtcars` by mpg (from highest to lowest)

Lab 1: Solutions

1.

```
x <- seq(2,10,by=2)
y <- c(3,5,7,11,13)

x+y
```

```
[1]  5  9 13 19 23
```

```
x-y
```

```
[1] -1 -1 -1 -3 -3
```

```
x*y
```

```
[1]   6  20  42  88 130
```

```
x/y
```

```
[1] 0.6666667 0.8000000 0.8571429 0.7272727 0.7692308
```

2.

```
geometric_mean <- function(x) {
  J <- length(x)
  res <- prod(x)^(1/J)
  res
}

geometric_mean(c(10,8,13))
```

```
[1] 10.13159
```

3.

```
first_date <- lubridate::mdy("01-01-1981")
second_date <- lubridate::mdy("01-10-2022")
second_date - first_date
```

```
Time difference of 14984 days
```

4.

a.

```
nrow(mtcars)
```

```
[1] 32
```

b.

```
ncol(mtcars)
```

```
[1] 11
```

c.

```
colnames(mtcars)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

d.

```
subset(mtcars, mpg >= 20)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
```

Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

e.

```
subset(mtcars, (mpg >= 20) & (hp >= 100))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

f.

```
subset(mtcars, (cyl >= 6) | (hp >= 100))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4

```
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

g.

```
mtcars[10,]
```

```
          mpg cyl  disp  hp drat   wt qsec vs am gear carb
Merc 280 19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
```

h.

```
# without reversing the order, we would order from lowest to smallest
mtcars[rev(order(mtcars$mpg)),]
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
```

Coding Exercises

1. The `stringr` package contains a number of functions for working with strings. For this problem create the following character vector in R

   ```
   x <- c("economics", "econometrics", "ECON 4750")
   ```

   Install the `stringr` package and use the `str_length` function in the package in order to calculate the length (number of characters) in each element of `x`.

2. For this problem, we are going to write a function to calculate the sum of the numbers from 1 to n where n is some positive integer. There are actually a lot of different ways to do this.

   - Approach 1: write a function called `sum_one_to_n_1` that uses the R functions `seq` to create a list of numbers from 1 to n and then the function `sum` to sum over that list.
   - Approach 2: The sum of numbers from 1 to n is equal to $n(n+1)/2$. Use this expression to write a function called `sum_one_to_n_2` to calculate the sum from 1 to n.
   - Approach 3: A more brute force approach is to create a list of numbers from 1 to n (you can use `seq` here) and add them up using a `for` loop — basically, just keep track of what the current total is and add the next number to the total in each iteration of the for loop. Write a function called `sum_one_to_n_3` that does this.

   Hint: All of the functions should look like

   ```
   sum_one_to_n <- function(n) {
     # do something
   }
   ```

   Try out all three approaches that you came up with above for $n = 100$. What is the answer? Do you get the same answer using all three approaches?

3. The Fibonacci sequence is the sequence of numbers $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ that comes from starting with 0 and 1 and where each subsequent number is the sum of the previous two. For example, the 5 in the sequence comes from adding 2 and 3; the 55 in the sequence comes from adding 21 and 34.

   a. Write a function called `fibonacci` that takes in a number `n` and computes the nth element in the Fibonacci sequence. For example `fibonacci(5)` should return `3` and `fibonacci(8)` should return `13`.

   b. Consider an alternative sequence where, starting with the third element, each element is computed as the sum of the previous two elements (the same as with the Fibonacci sequence) but where the first two elements can be arbitrary. Write a function `alt_seq(a,b,n)` where `a` is the first element in the sequence, `b` is the second element in the sequence, and `n` is which

element in the sequence to return. For example, if $a = 3$ and $b = 7$, then the sequence would be $3, 7, 10, 17, 27, 44, 71, \ldots$ and `alt_seq(a=3,b=7,n=4) = 17`.

4. This problem involves writing functions related to computing prime numbers. Recall that a prime number is a positive integer whose only (integer) factors are 1 and itself (e.g., 6 is not prime because it factors into 2×3, but 5 is a prime number because its only factors are 1 and 5).

   For this problem, you cannot use any built-in functions in `R` for computing prime numbers or checking whether or not a number is a prime number. However, a helpful function for this problem is the *modulo* function, `%%` discussed earlier in the notes. **Hint:** Notice that `6 %% 2 = 0` indicates that `2` is a factor of `6`; on the other hand, if you divide 5 by any integer small than itself (except for 1), the remainder will always be non-zero.

   a. Write a function `is_prime` that takes `x` as an argument and returns `TRUE` if `x` is a prime number and returns `FALSE` if `x` is not a prime number.

   b. Write a function `prime` that takes `n` as an argument and returns a vector of all the prime numbers from 1 to n. If it is helpful, `prime` can call the function `is_prime` that you wrote for part (a).

5. Base `R` includes a data frame called `iris`. This is data about iris flowers (you can read the details by running `?iris`).

   a. How many observations are there in the entire data frame?

   b. Calculate the average `Sepal.Length` across all observations in `iris`.

   c. Calculate the average `Sepal.Width` among the `setosa` iris species.

   d. Sort `iris` by `Petal.Length` and print the first 10 rows.

6. One of the examples that we gave above was about writing a function to solve quadratic equations, but, in the code presented above, we only returned one solution to the quadratic equation. Write a function `quadratic_solver` that takes in `a`, `b`, and `c` as arguments and returns both solutions to the quadratic equation in a list. For example, `quadratic_solver(1,4,3)` should return a list with two elements, `-1` and `-3`.