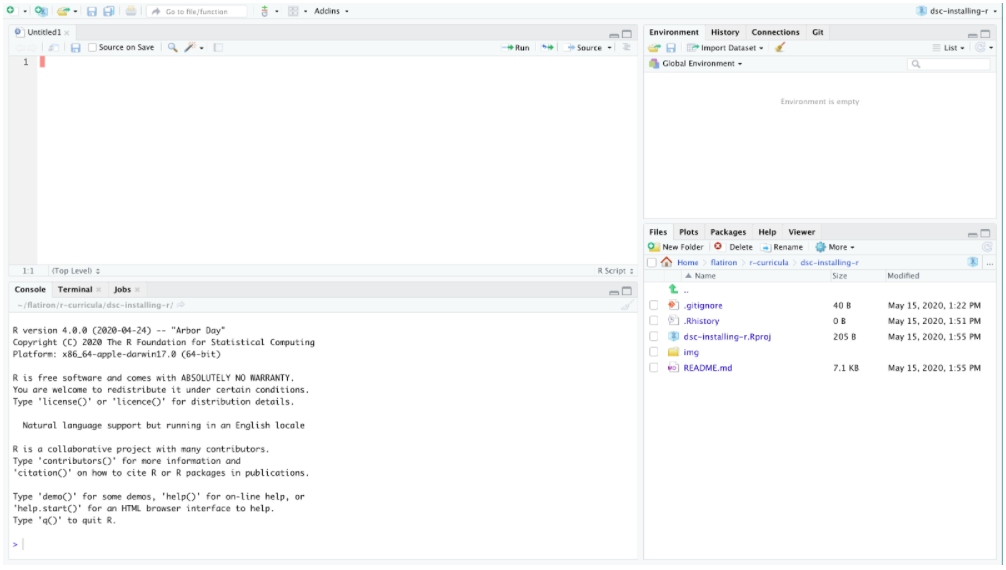
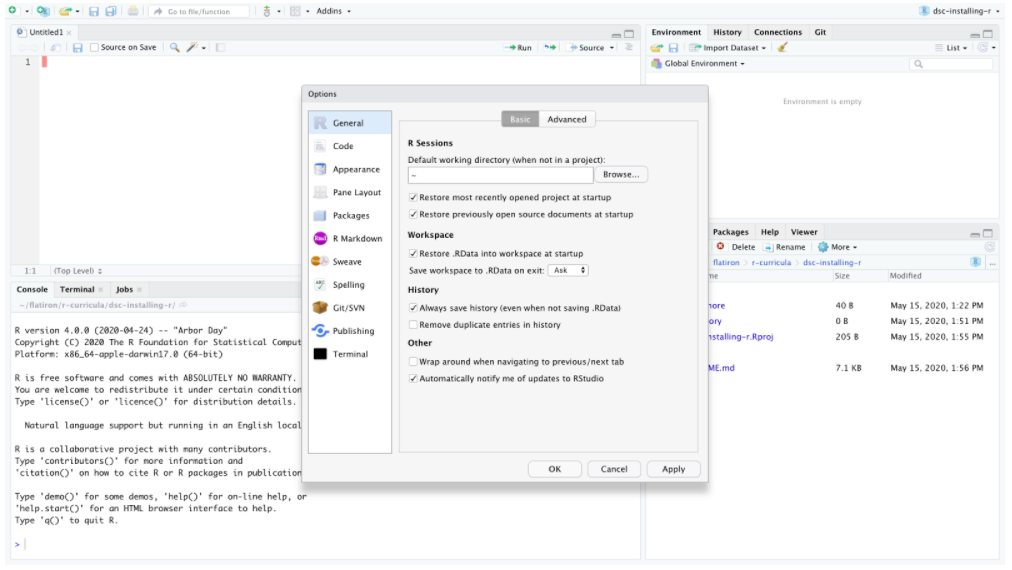
**RStudio IDE** RStudio is the GUI for all things R. When you first open RStudio, typically you will see four separate panels.



On the top left is your script editor where you write your code, on the bottom left you have your console where your code gets run. On the top right you see the environment-- something we’ll talk about soon-- and then on the bottom right we see our Viewer. You can change the positions of this if you’d like and [can find instructions to do that here](https://support.rstudio.com/hc/en-us/articles/200549016-Customizing-RStudio) and can also change the color schemes of your editor if you navigate to the preferences.

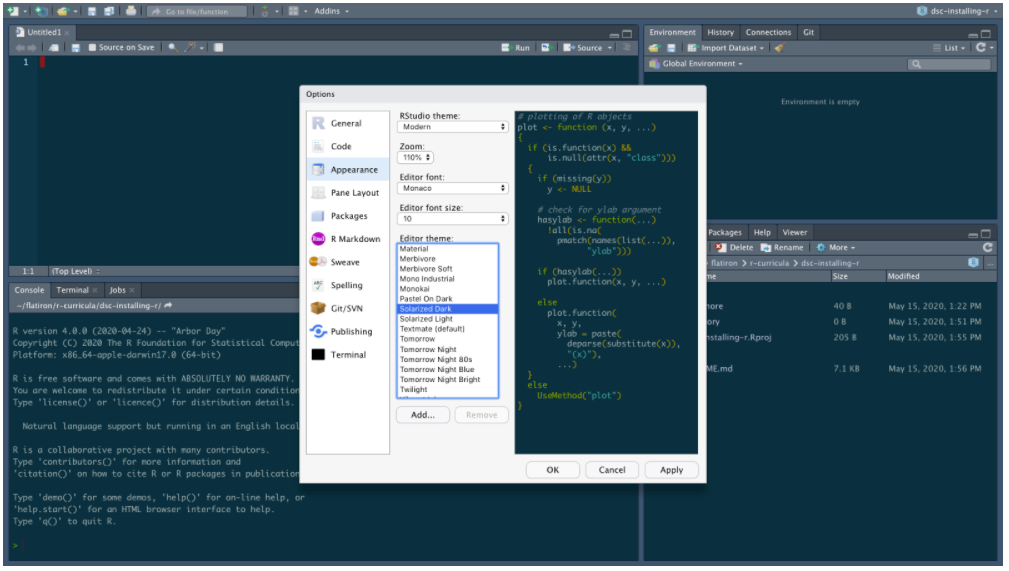
Let’s first try that!

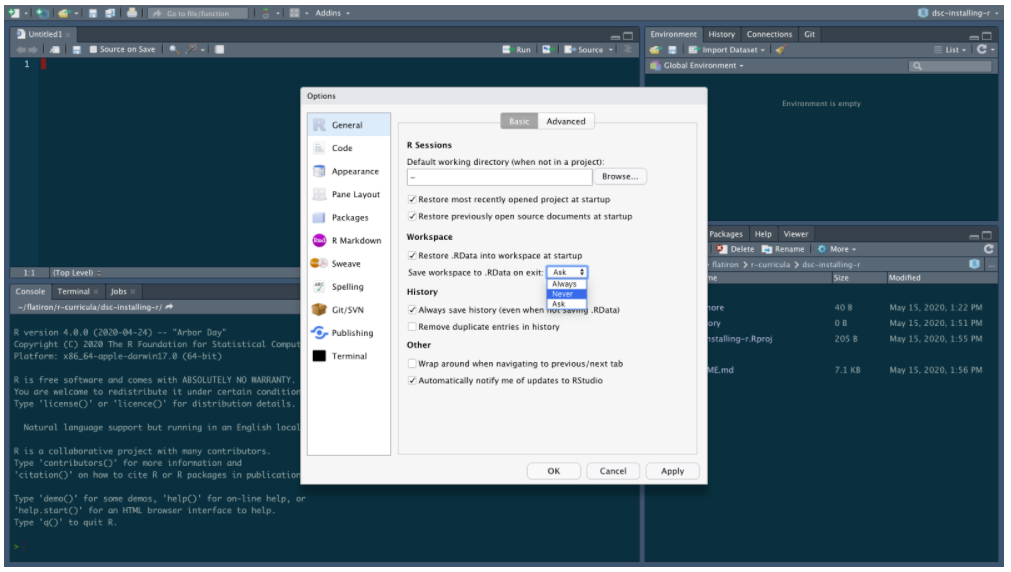
I’d like you to go in the top toolbar then select File > Preferences ... Windows: Tools>>Global Options…



We can change here to Solarized Dark.

Now while we’re here in Preferences, let’s also do something that’s going to save you a lot of pain in the long run which is make the default behavior to never save your work space.





**Running a Script**

Now we’ve done a lot here to get both R and RStudio installed and set up here, let’s end with running one script!

The .rproj file basically walls off the rest of your computer so RStudio thinks the entire universe of your project lives within this area. Using .rproj files helps eliminate absolute paths and makes it so it’s a lot easier to get your R code to run on others computers. If you’re serious about learning about good practices in working with [R and RStudio, please check out this e-book here (written in R)](https://rstats.wtf/index.html).

In this local repository for this lesson, you’ll find a file called tips\_report.Rmd that you should be able to see if you click the File tab on the bottom right quadrant of RStudio. This will open up your first RMarkdown file (the Juypter notebook of R).

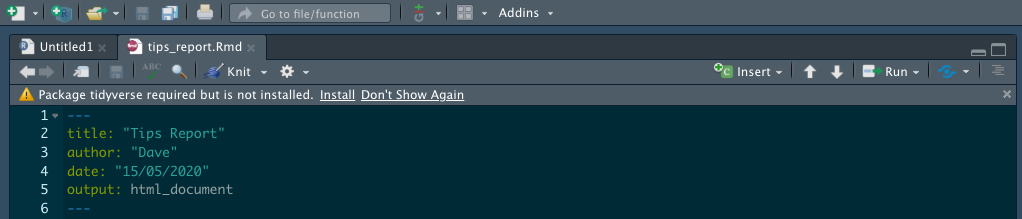
This file contains the data and narrative we will be using the next lessons. We’ll describe it more at the start of the next lesson!

With this open, let’s just click where it says Knit at the top to see what happens.

Note here that because you’ve done a fresh install of R, you might be prompted to install a lot of software.

**Make sure you agree to all of this!**

The first time you run this, you will also see something like this which asks you if you want to install the library (or suite of libraries we’re going to use) this time. Make sure to also install this and say Yes when it asks you at the command prompt to install everything!



Typically we would do this at the command line with something like:

install.packages("tidyverse")

But RStudio is smart and realizes that we don’t have it and we wanted to show you that!

**DOES NOT HAPPEN AND ERRORS OUT!!!!**

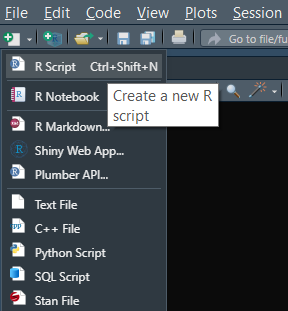
Once all that software is installed, you should be able to run your script.

This will run the RMarkdown script and create a little report for you. Notice it’s an HTML file of your analysis meaning you can now just put the file.html that was just created on any website! RMarkdown allows data scientists to make quick reports in HTML, LaTeX, or even Word formats.

#### Getting Comfortable

Let’s start typing some R code! In order to get practice working in RStudio, we suggest typing out this code in the RStudio script editor (the top left panel in RStudio).

In order to make a new script you need to click the little green icon in the top left corner and select NEW SCRIPT. (New R Script)



What is great about RStudio is that you can run any line of your script, just like you can run any cell in a Jupyter Notebook, individually. If you hold down CMD and press RETURN on any selected line in the editor, you can run a line. We’ll try this together.

#### First Commands

Like before, let’s try to type in some basic math into R. Instead of just typing it into the Console, let’s instead write out a line in a new script.

Again, let’s just add two numbers.

2 + 2

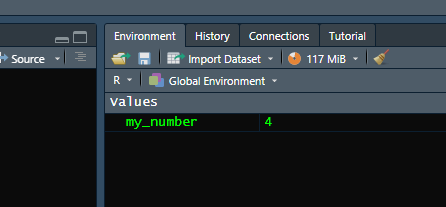
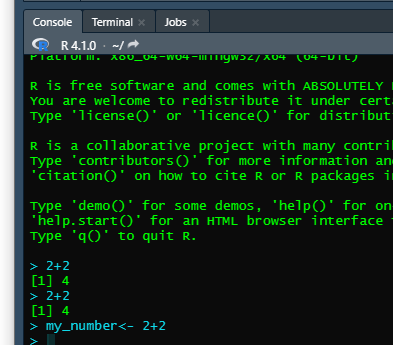
We can run this line by typing Ctrl + Enter assuming that the cursor is on the line you want to run. This will send this line of output to be run through the console. Notice that your output is now shown below.

Not that impressive, most programming languages can do that! Let’s now write something that actually looks like R.

my\_number <- 2 + 2

Let’s now run this code here that saves our operation into an object. Now don’t just run it right away, let’s take a second to think about what is the same and different as Python. As with Python, we are assigning some sort of expression to an object. The naming conventions of objects in R as pretty much the same as Python, but notice that in R we use the assignment operator <- as opposed to equals =. There are a couple of different reasons why this is. The short answer as to why this is, is because this is part of R’s [style guide](http://adv-r.had.co.nz/Style.html). You can google around if you want to find the long answer for why this is the case.

Now as we run this line, we know from before that it will get sent to the console. So knowing that we know what will happen, let’s instead direct our attention to the top right panel when we run this. This top right panel is our Global Environment and keeps track of what variables are in our work space.

f you did this, you screen will look something like above. The command was sent below and we now have a new value in our Global environment.

Just like in Python, we can now manipulate this new object. For example if we did:

my\_number \* 2

Our number, 4, would get multiplied by 2 just like in Python!

But we know now that in data science, we don’t usually want to multiply just ONE number, but rather a whole collection of numbers. This is where R’s differences start to show.

Let’s now make a vector (what R calls a one dimensional collection of objects of the same type) of a some numbers using R’s c() function. We can pretend this is a bunch of data on the number of coffees you might drink in a day.

coffees <- c(2,1,2,3,1,2,0,2,3,1)

Now if we were in Python, this might start out a as a list and we’d have to numpyifiy it in order to do some math operations on it. Since R is a programming language that really is designed for manipulating numbers, we don’t have to do something equivalent.

Let’s imagine we’re trying to calculate how much caffeine we’ve taken in each day and realized that mug we’re drinking out of is actually a little big bigger than the normal cup so we need to scale our entire data by a factor of 1.2. We can just multiply the whole object by 1.2.

coffees \* 1.2

If you run this, you’ll notice that it ran just fine. No need to turn a list into a numpy array.

This works because R uses element-wise execution. If you want to read more about this, check out this chapter on [The Very Basics of R](https://rstudio-education.github.io/hopr/basics.html#objects) by Garrett Grolemund. There’s a lot of other strange (if you’re a Pythonista) results that happen when you have this as a basic feature of the language.

Let’s save our new output into a new variable.

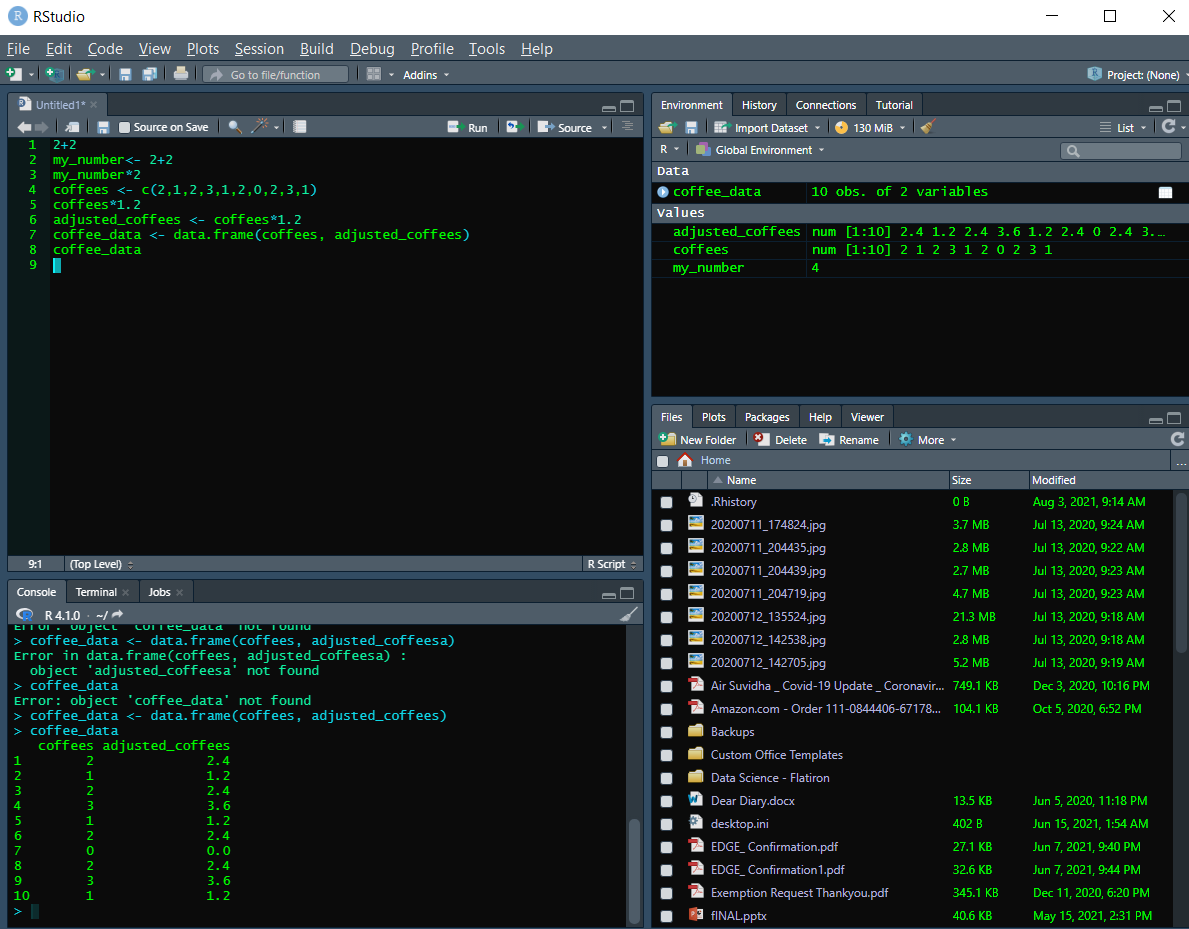
adjusted\_coffees <- coffees \* 1.2

Now it might seems a bit extra to have two objects of two things that are related not as part of the same entity. Since they are two vectors of the same length, we can combine them into a data frame. In order to do this, let’s make a new object with the data.frame() function.

coffee\_data <- data.frame(coffees, adjusted\_coffees)

coffee\_data

If we now run coffee\_data in RStudio, we can see something that looks like what we’re more familiar with.



#### Accessing Data

At this point coffee\_data should look pretty familiar as the kind of tabular data you’re used to working with. So how do you then subset/index parts of coffee\_data? There are a couple of ways to do this with base R, but in the next few lessons, we’ll explore a way of doing this that is a bit easier to read.

The way to get subsets of data from an R object is with the square bracket operators. When using the square brackets [ and ], the first argument will correspond to rows and the second to columns. For example, if we wanted to get the first row from our coffee data we would index our data by typing:

coffee\_data[1,]

After months of Python, this might be a bit jarring to see. Yes, R is 1 as opposed to 0 index. If you want the first element of an ordered object in R you use the number 1.

Notice here that there is a comma that lets R know that we’re operating on a two dimensional object (it has rows and columns). Also notice that since we want all other columns, we leave everything after the comma blank.

If we instead wanted just the first column, we would type:

coffee\_data[,1]



And if we wanted the data from the first row and first column, we would type:

coffee\_data[1,1]



There are other ways to get data out of a data frame and the last thing we’ll show you is one way to extract a column from a data frame. If you want to, for example, grab out the column coffees, we can use the $ to do this.

coffee\_data$coffees



This will print out our original data from before. And if we wanted to get just the first entry of this, we could again use the [ ], but since we have only one dimension here, we don’t need the comma ,.

coffee\_data$coffees[1]

Just like Python, there are many, many ways to expand this out to get the data you might want and you can read about it [here if you would like](https://rstudio-education.github.io/hopr/), but in the later R lessons, we’re actually going to focus on a different way of working with R to chop up data.

#### Functions

At this point you’ve probably written more than a couple of functions in Python. You’ve had points in your analyses where there is no exact function to get the exact data and formatting you need, so you write your own functions in Python.

Here we look at functions in R.

Before writing our own functions, let’s look at some similarities between R and Python functions that you might be comfortable with.

In the last lesson, we started with one of the most basic math operations we could think of which is adding 2 + 2.

2+2

If you just looked at the code above, you actually could run that line in either R or Python since it’s just basic math. In Python, if we wanted to do this using something a bit more sophisticated, we could use the .add() method from numpy.

import numpy as np

np.add(2,2)

Now since R is designed to deal with data first and foremost, the normal way to do this in R doesn’t require grabbing an external package.

sum(2,2)

Notice here that we don’t really have to specify what package sum is coming from. Now it’s not just coming out of thin air. Behind the scenes R’s sum() function here is coming from R’s base package. We could re-write the above as:

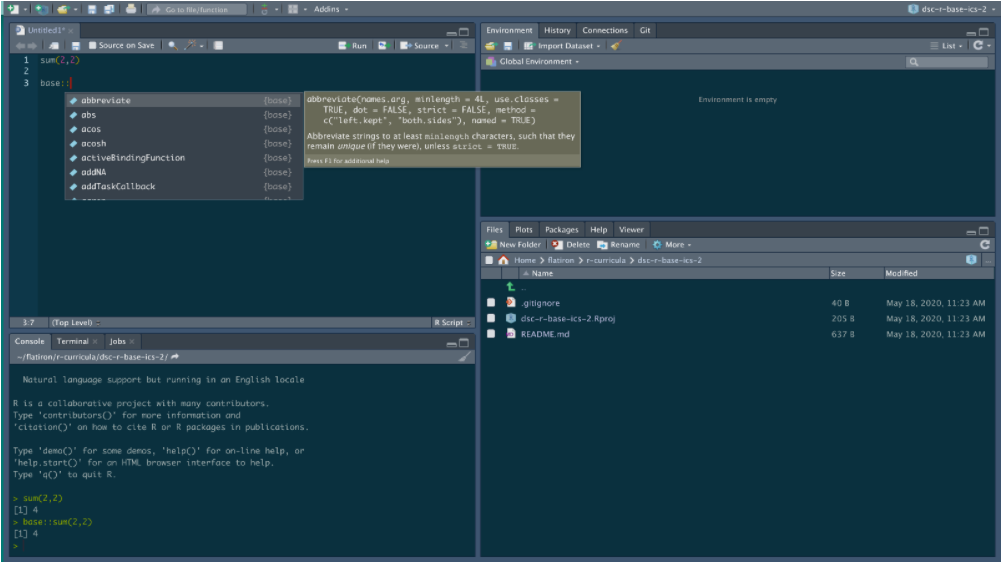
base::sum(2,2)

In order to be more explicit about where the function is coming from. Now we won’t get into the idea of environments and function masking here, for more reading on that check out [Advanced R](http://adv-r.had.co.nz/Introduction.html). But we wanted to show how R can kind of look like Python.

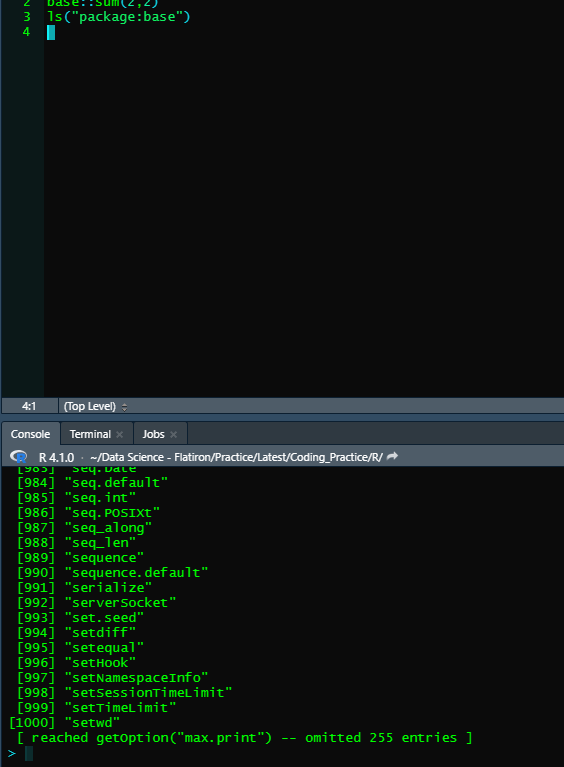
Now there are TONS of functions from R’s {base} package that you can find by either typing:

lsf("package:base")

Or can be a bit more casual and explore using RStudio’s auto complete feature. If you type out base:: and letting RStudio’s auto complete to do the rest!



If lsf doesn’t work, try ls.

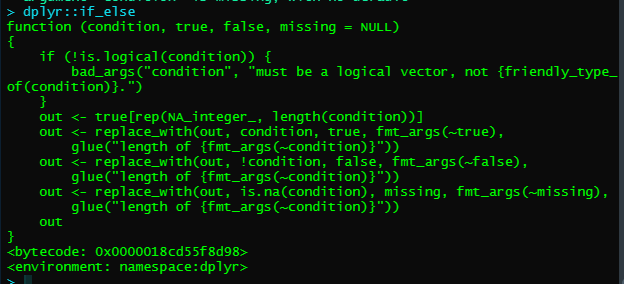


Lastly, before moving on to writing our own functions, if you want to see the code underlying any function in R, you can always type it without the ().

For example, if we look at the if\_else() function out of dplyr by just running:

dplyr::if\_else

Our output will look something like this:



One fantastic way to to learn a lot about any language when starting out is to try to spend a lot of time reading it before committing to writing it (just like learning a new spoken language!).

#### Writing Our Own Functions

As listed in [Hands on Programming with R](https://rstudio-education.github.io/hopr/basics.html#functions), functions in R have three basic parts:

* Name
* The Body of Code
* Set of Arguments

and take the general form of

my\_new\_r\_function <- function() {}

Now since you probably know a bit about functions in Python, let’s jump straight to looking at a function in R and try to find similarities and differences! Let’s imagine you’re planning an European holiday and need to practice understanding what temperatures mean in Celsius so you write yourself a program to convert your Fahrenheit temperature to what you’ll read on your trip.

convert\_f\_to\_c <- function(farh\_number) {

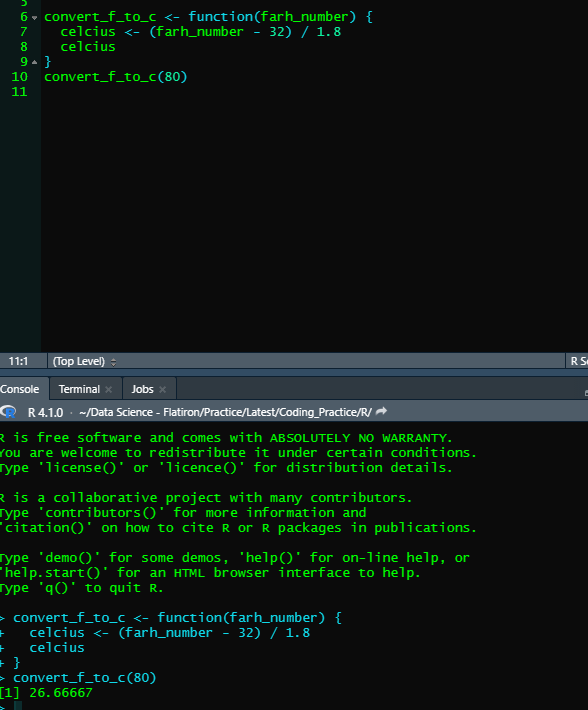
celc <- (farh\_number - 32) / 1.8

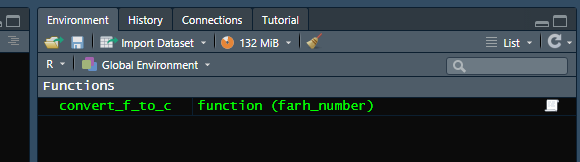
celc

}

On the left of the assignment operator we have the function name, in this case it’s convert\_f\_to\_c. Our argument here is x, which is going to be our temperature in Fahrenheit. After declaring what arguments we’re going to put into our function inside the ( ) parenthesis, we then write our function body between the { }. The code here could almost be Python with the exception of the <- operator!

In order to get better at writing functions in R, one thing to do would just be to keep things easy and try to convert some of your favorite functions from Python to R.





#### Tidyverse

No introduction to the R programming language would be complete without an introduction to the [tidyverse](https://www.tidyverse.org/). According to tidyverse website,

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

While there’s no way to cover everything you would need to know to be a tidyverse master in the next few lessons, the next few lessons will show you some of what the tidyverse has to offer.

#### Tidyverse

Most of data science is cleaning data and the small majority that is left after cleaning is normally spent about talking about cleaning data. If anything is left after that, you can actually run your models.

By this point, you have a lot of lived experience cleaning data and hopefully recognize how helpful it can be when people create tools that help with this process. In the world of R, a growing set of tools has been developed over the past decade or so that has tried to make working with data easier called the Tidyverse.

There’s a wealth of information you can read about ranging from [this blog post](https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html) to the actual [paper](https://vita.had.co.nz/papers/tidy-data.html) that was written about tidy principles, but for our purposes, we just have to know that the tidyverse is a set of packages that work together because the “share common data representation and API design”.

Once you have data in a tidy format-- meaning each variable is a column, each observation is a row, and each type of observational unit forms a table-- you can go to town on your data set with the tidyverse!

We had you download the tidyverse in step one of this tutorial, but if you have skipped that you need to run the below command in your R console. If you’re prompted to download more software, make sure to do that!

# get tidyverse from CRAN

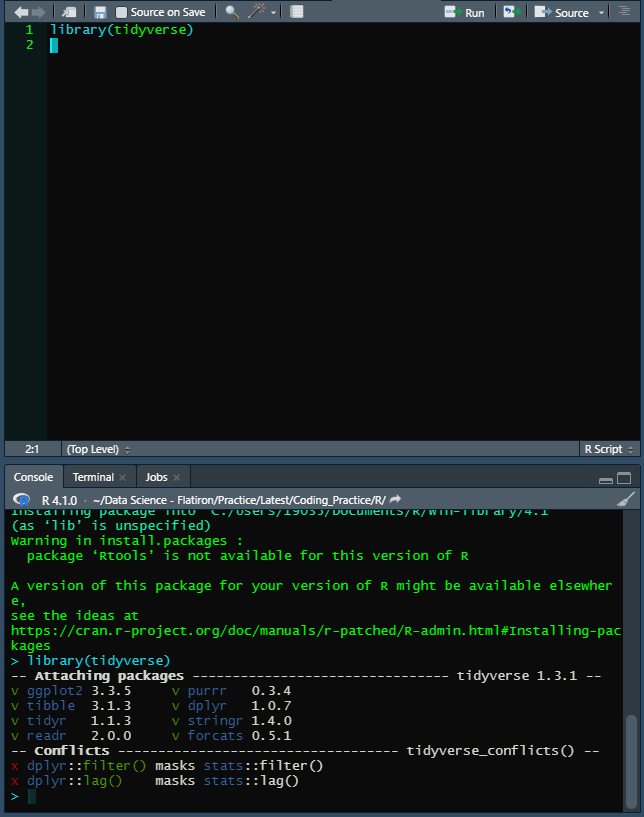
install.packages("tidyverse")

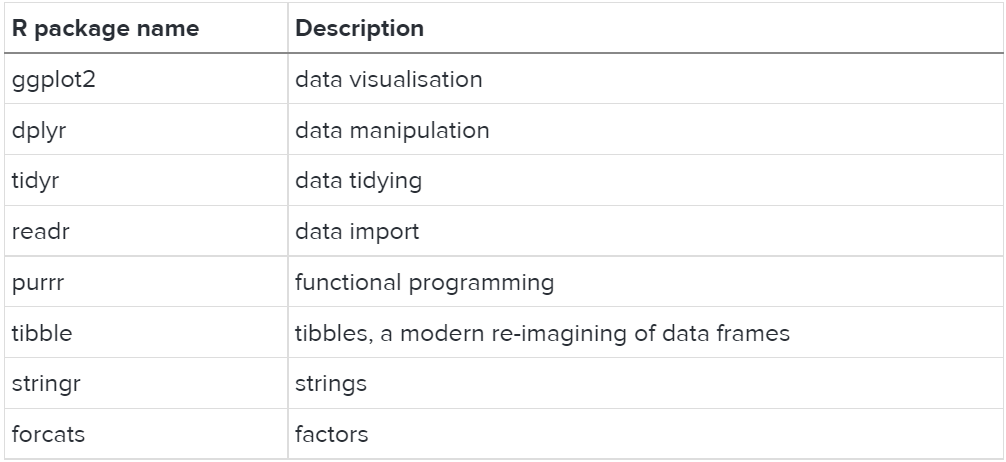
Remember, just like pip or conda, you only have to use install.packages() once unless you are updating.

Then once installed, you have to call the software to use it, in our case this will be

library(tidvyerse)

When you do this, you end up getting the [core tidyverse packages](https://tidyverse.tidyverse.org/). We’ve reproduced the table from the link below here.





There is also tidymodels, which for learning purposes will be the “sci-kit learn” of the tidyverse.

We won’t get to everything, but if you do want to do a proper dive into the tidyverse, your best place to go learn is [R for Data Science](https://r4ds.had.co.nz/).

#### Five Verbs

Let’s now continue to explore the tidyverse with the dplyr package. Here we’ll learn about five verbs that you will probably use most frequently in your data cleaning, as well as the pipe operator.

This first bit of code that you need to run in RStudio grabs a few different libraries we will be using then we import in the tips.csv data set as the object tips as a special type of data frame in the tidyverse called a tibble.

library(dplyr) # for manipulating data

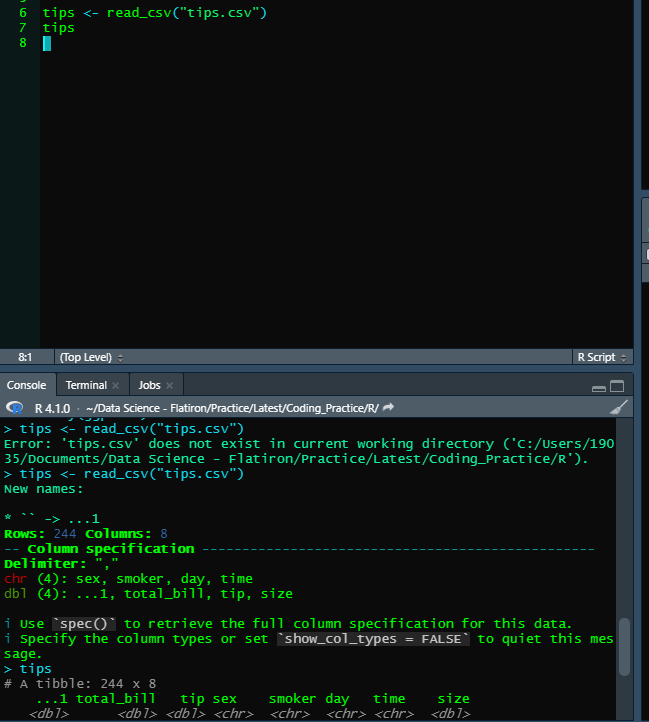
library(readr) # for getting data

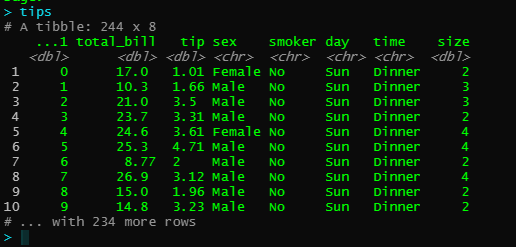
library(ggplot2) # for plotting data

tips <- read\_csv("tips.csv")

# You can change where this is output above in "Settings (by knit) > Chunk Output in Console"

tips

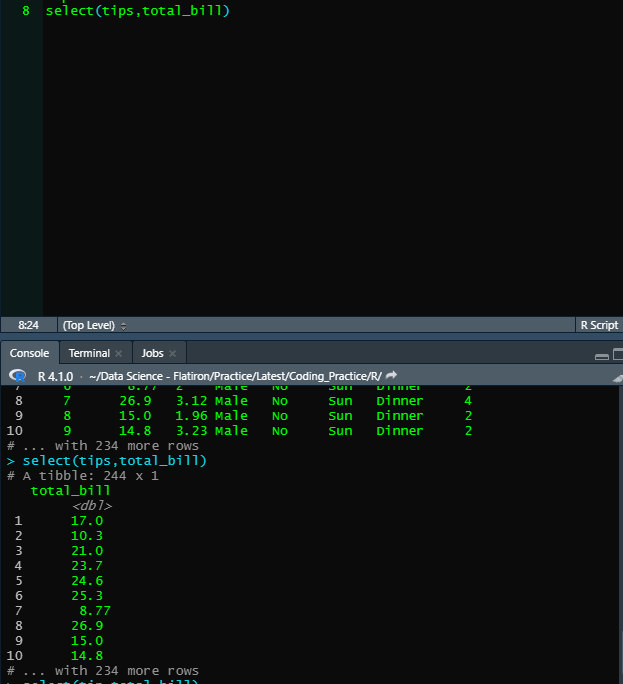


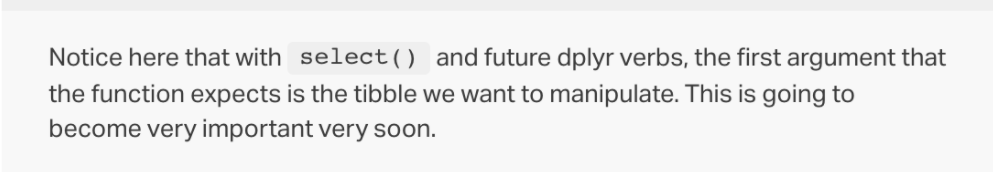


The first verb we will use is select() which lets us choose which columns we are interested in working with.

select(tips, total\_bill)

The select() command takes two arguments here. The first argument is the data frame that we want to begin to manipulate.

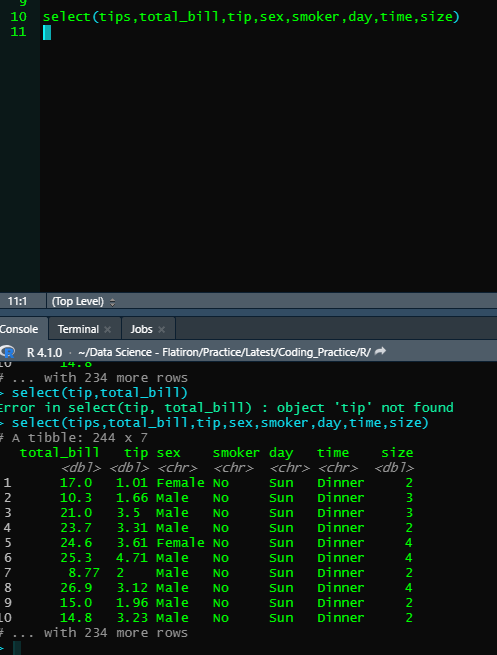




The second argument that select() will take are what columns we want to … select. In this example, we only get the column total\_bill.

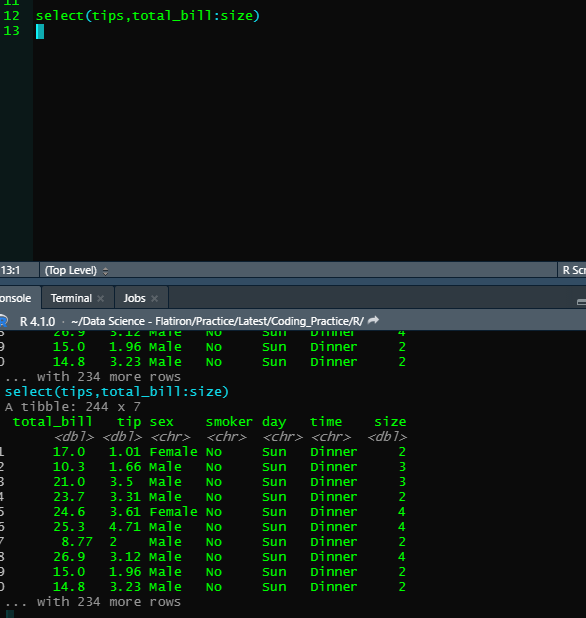
Now we often don’t just want one column, sometimes we want more than that! We can just keep adding whatever columns we need with more arguments.

select(tips, total\_bill, tip, sex, smoker, day, time, size)



Now, this is a lot to type out. The cool thing about dplyr is that you can use some of the same operators you would use on numbers on columns. The call below gives us the exact same output as before but is a bit more concise.

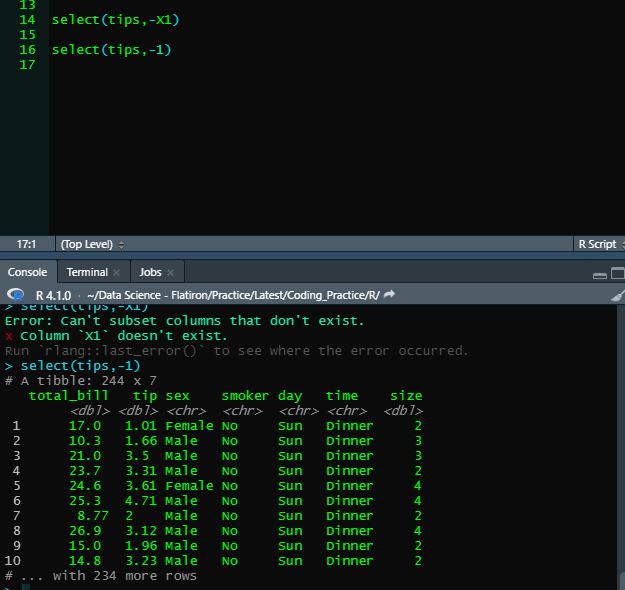
select(tips, total\_bill:size)



But maybe the reason that we are doing this is just because we want to drop that X1 column that came for the ride because we’ve imported this pandas data frame. Instead of saying what columns we do want, it might be easier to just say what we don’t want.

select(tips, -X1)

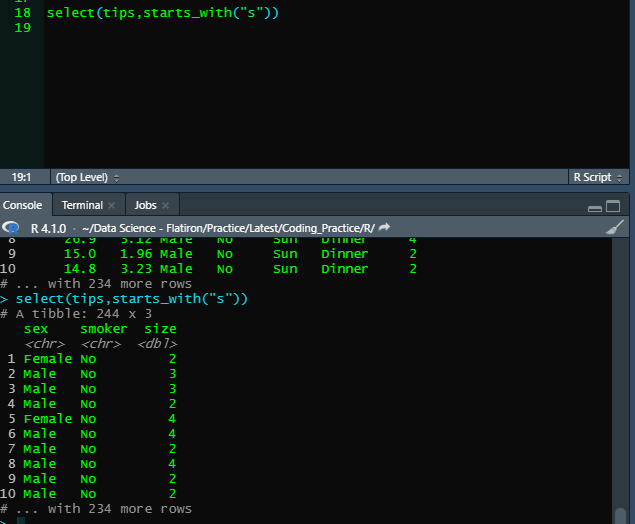
Again, we see the same output!



Now, this is all pretty basic, but there are a couple of other cool things you should know about. For example, the select() function also has an argument starts\_with() that takes a character!

select(tips, starts\_with("s"))

And of course if there’s a starts\_with(), there’s also an ends\_with().



If you’re going to be using dplyr a lot, it’s worth [reading a bit of the documentation on it](https://dplyr.tidyverse.org/reference/select.html) as some functions like pivot\_longer(), which is used to take wide data to long format data, use select() syntax.

So if select is for columns, the what do we have for rows? It’s a function called filter().

filter() works very much like select() in that as a function, the first argument it expects is the data frame where you want to pick specific rows to subset out.

It can deal with both numeric and character input as seen below as well as conditional operators!

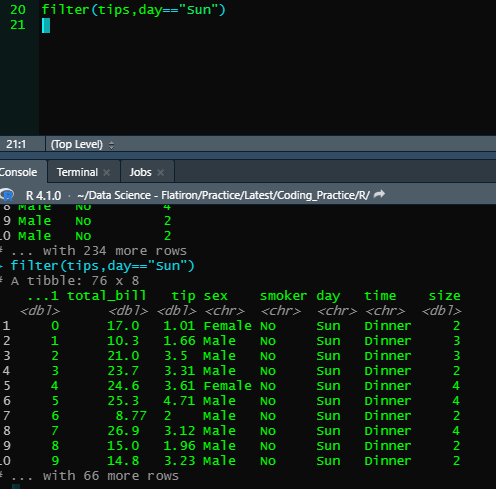
Try to run the code below and see what it does.

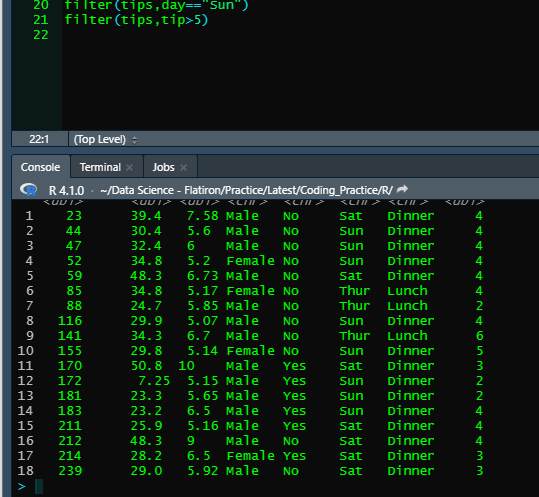
filter(tips, day == "Sun")

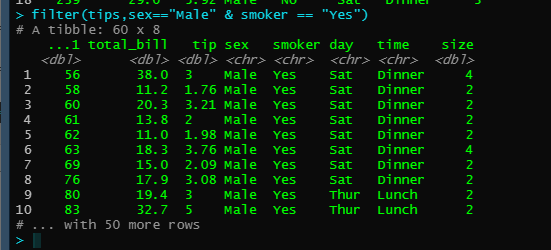
filter(tips, tip > 5)

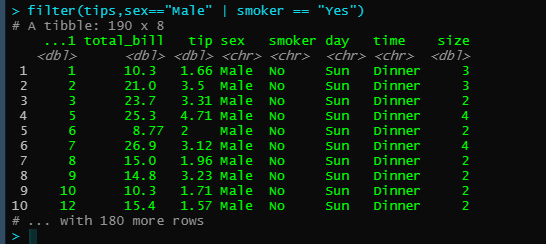
filter(tips, sex == "Male" & smoker == "Yes")

filter(tips, sex == "Male" | smoker == "Yes")









So now we know how to pick apart a data frame based on either columns or rows, but what if we want to make new variables on the fly, just like in pandas?

For that, we are going to use the mutate() function.

Let’s return to our example from before where we wanted to convert the data in our tips.csv data set from USD to GBP. Right now, the conversion rate is that 1 USD is about .82 GBP.

Again, the way the syntax works is that the first argument of mutate() expects the data frame that we are going to manipulating. The next argument is the creation of a new variable. In this case, we are creating a new variable called gbp\_total and then using the = operator to tell mutate() what we want that new variable to be. Here all we need to do is then take the variable we want to manipulate and multiple it by .82. Since R does element-wise execution, what’s happening under the hood is every element in the column total\_bill gets multiplied by .82 and is added to a new column of the original data frame. We’ll also do this for the tip variable. Notice that we just separate the two different variables within mutate() with a comma to say these are separate arguments.

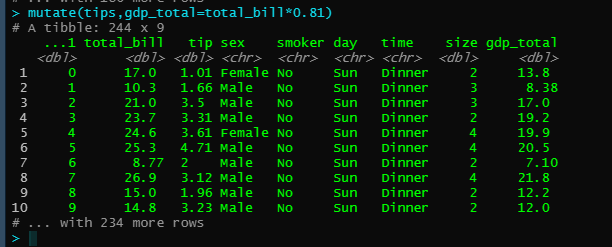
mutate(tips, gbp\_total = total\_bill \* 0.81)

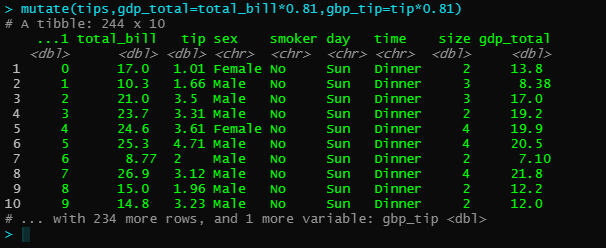
# R doesn't care about spacing!!

mutate(tips,

gbp\_total = total\_bill \* 0.81,

gbp\_tip = tip \* 0.81)





Up until this point, we have been doing some very basic commands.

What if we want to get a bit more sophisticated? Let’s say we only wanted a few columns and rows selected based on some selection criteria. For example, let’s only look at the columns smoker and total\_bill\_gbp, but only where gbp\_tip is more than a fiver (£5).

Now if we were doing this like we have been above, it might look something like this.

my\_first\_subset <- select(tips, smoker, tip)

my\_second\_subset <- mutate(my\_first\_subset, gbp\_tip = tip \* .82)

my\_data\_frame\_i\_actually\_wanted <- filter(my\_second\_subset, gbp\_tip >= 5)

my\_data\_frame\_i\_actually\_wanted

That’s OK, but kind of a pain to read. We have to keep track of intermediate variables, figure out what to name them, and doing this makes us kind of bound to this order of code.

Since this is something we do a lot of in data science, there’s actually a really slick way of getting around this problem with something called the pipe operator %>% from the magrittr package.

Let’s first look at the code above re-written with the pipe operator, then see if we can figure out what it’s doing.

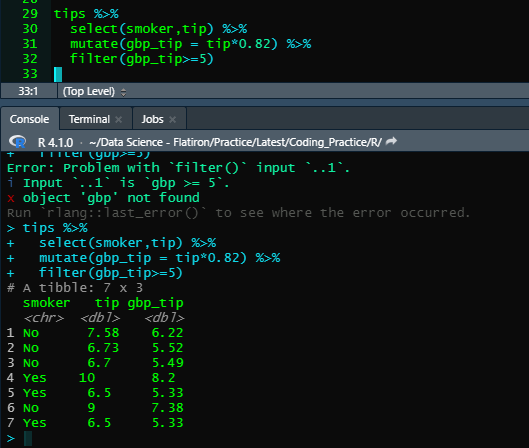
tips %>%

select(smoker, tip) %>%

mutate(gbp\_tip = tip \* .82) %>%

filter(gbp\_tip >= 5)

This is a lot more compact and hopefully easier to read!



So what’s going on here?

Well, remember what we said before about it being really important that the first argument of a dplyr verb wanting the tibble (data frame)? That’s where the secret sauce is, what is happening is that the first object, in this case tips is getting passed through the pipe and secretly becoming the first argument of the next line. You can think of it as the code reading:

“Start with the tips tibble and then select the smoker and tip columns”

tips %>% # and then !

select(smoker, tip)

Whatever tibble is created from this process then gets passed to the next pipe. The great thing about this is that you can join up as many pipes as you’d like! From here we can then build up much more complex commands.

For example, based on what we have learned thus far, can you figure out what the code below is doing?

tips %>%

select(total\_bill, tip, sex, smoker) %>%

filter(sex == "Male" & smoker == "Yes") %>%

mutate(gbp\_total\_bill = total\_bill \* 0.82,

gbp\_tip = tip \* 0.82)

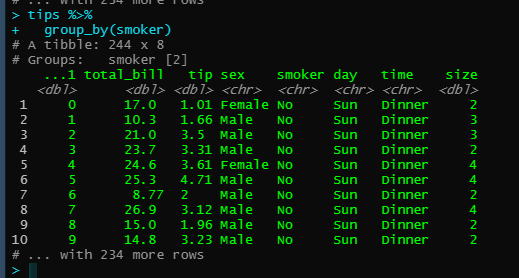
Most of what we have learned thus far is how to break down or add on to a tibble that we already have. But what if we want to start collapsing it down based on some parameters? For example, in our current state, we can’t figure out the average tip between smokers and non-smokers.

For that we need both the group\_by() and summerise() functions.

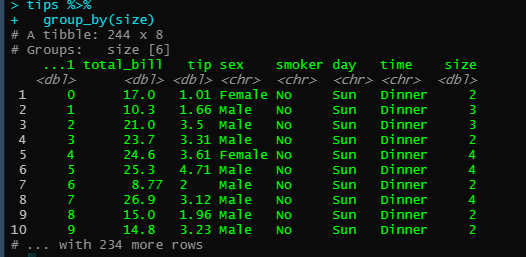
tips

tips %>%

group\_by(smoker)



To get an idea of what group\_by() is doing, run the two lines above and look for some difference in the output of your code. In the second bit of code, you’ll see that because you passed the group\_by() argument, the table is now split into two different groups as noted by the little Groups: smoker [2] in the output. Try and see what happens when you pass the size variable (a double) instead of a character variable!



So on its own, it’s not that helpful, but we can use this in conjunction with the summerise() function. The summerise() function works very much like mutate(). Look at the code below and see if you can notice any similarities between the code here and what mutate() did above.

tips %>%

group\_by(smoker) %>%

summarise(mean\_tip = mean(tip),

count = n())

Here we group our data set into two groups on the variable smoker, then figure out the average tip per group and call it a new variable called mean\_tip. We also make a new variable called count using the special function n().

Lastly, let’s arrange our output here so that those who tip more are on top! This isn’t the biggest problem in our little, chopped down data set here, but by this point, you can imagine why this next command might be helpful.

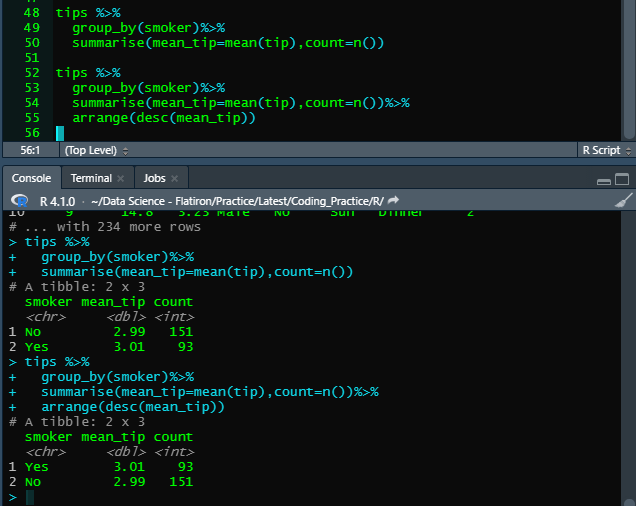
tips %>%

group\_by(smoker) %>%

summarise(mean\_tip = mean(tip),

count = n()) %>%

arrange(desc(mean\_tip))



For ascending, don’t mention anything:

