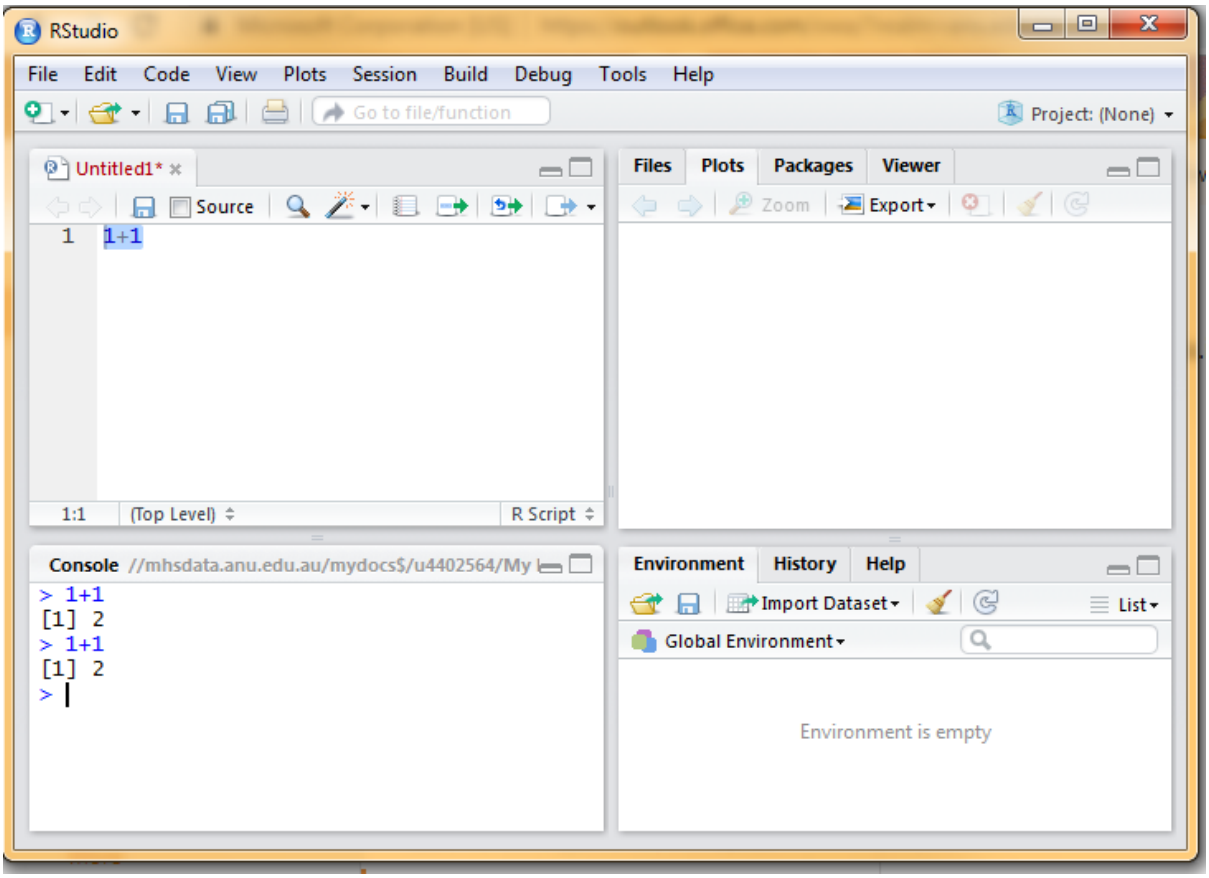
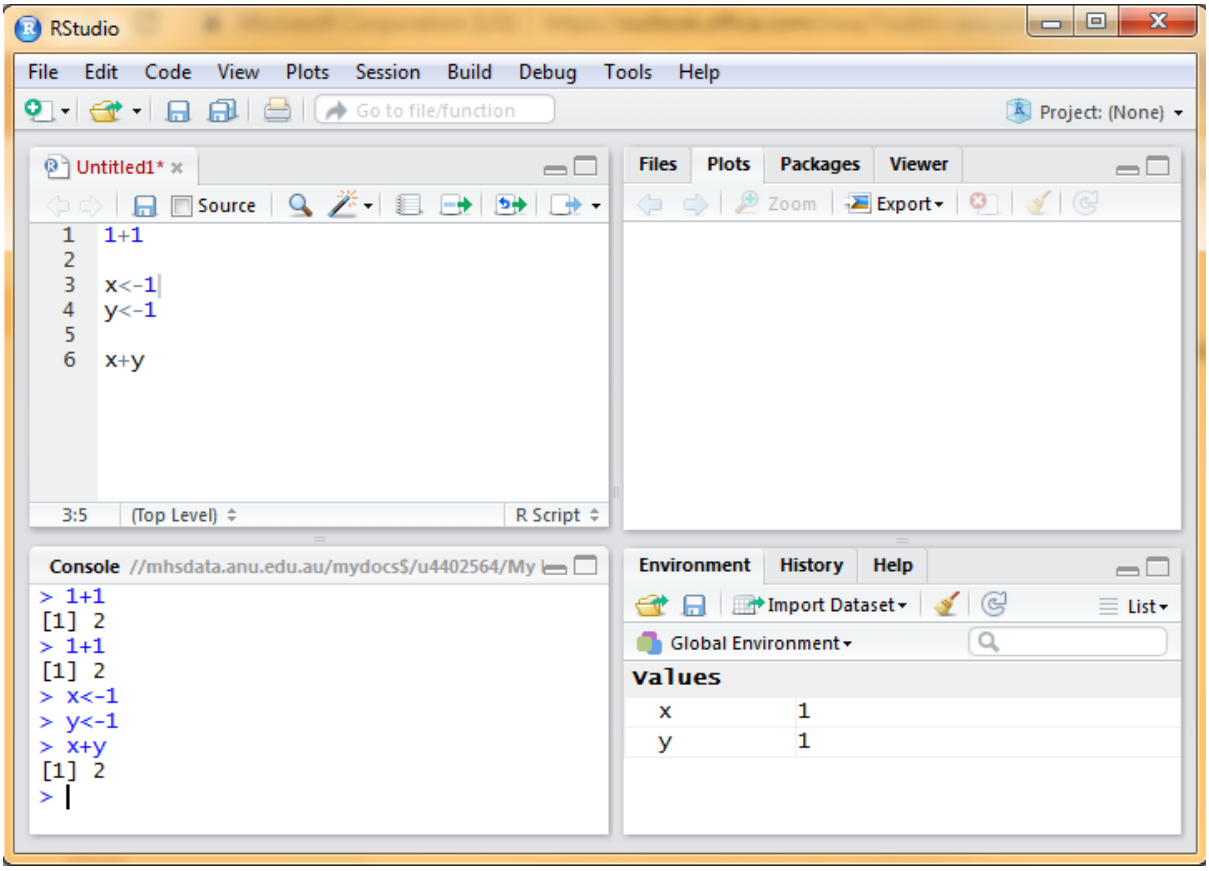
Basic R orientation

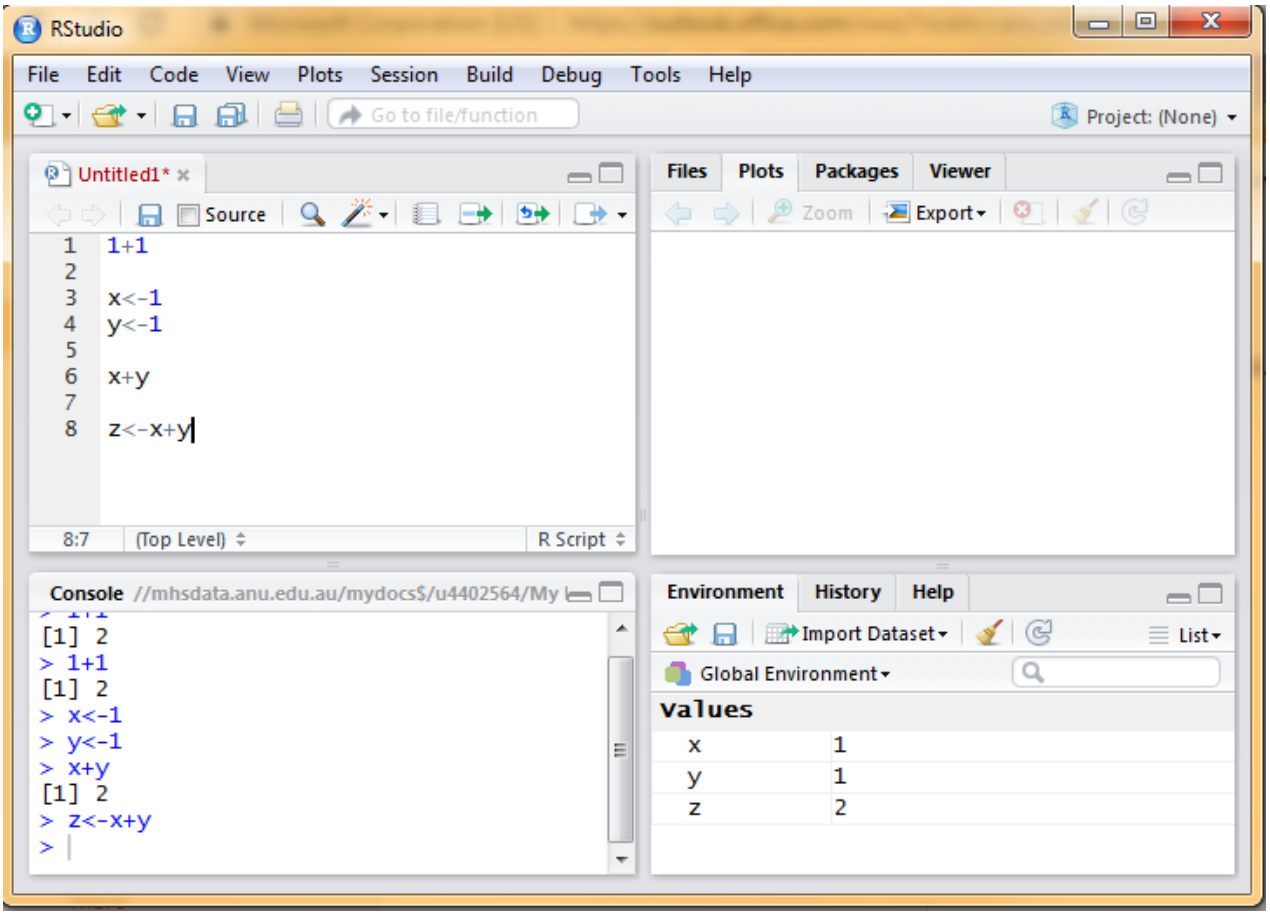
You can run the same things either directly in the console (below), or the script window (above). Anything you run in the script window (highlight it and press control + enter) will also show up in the console too.



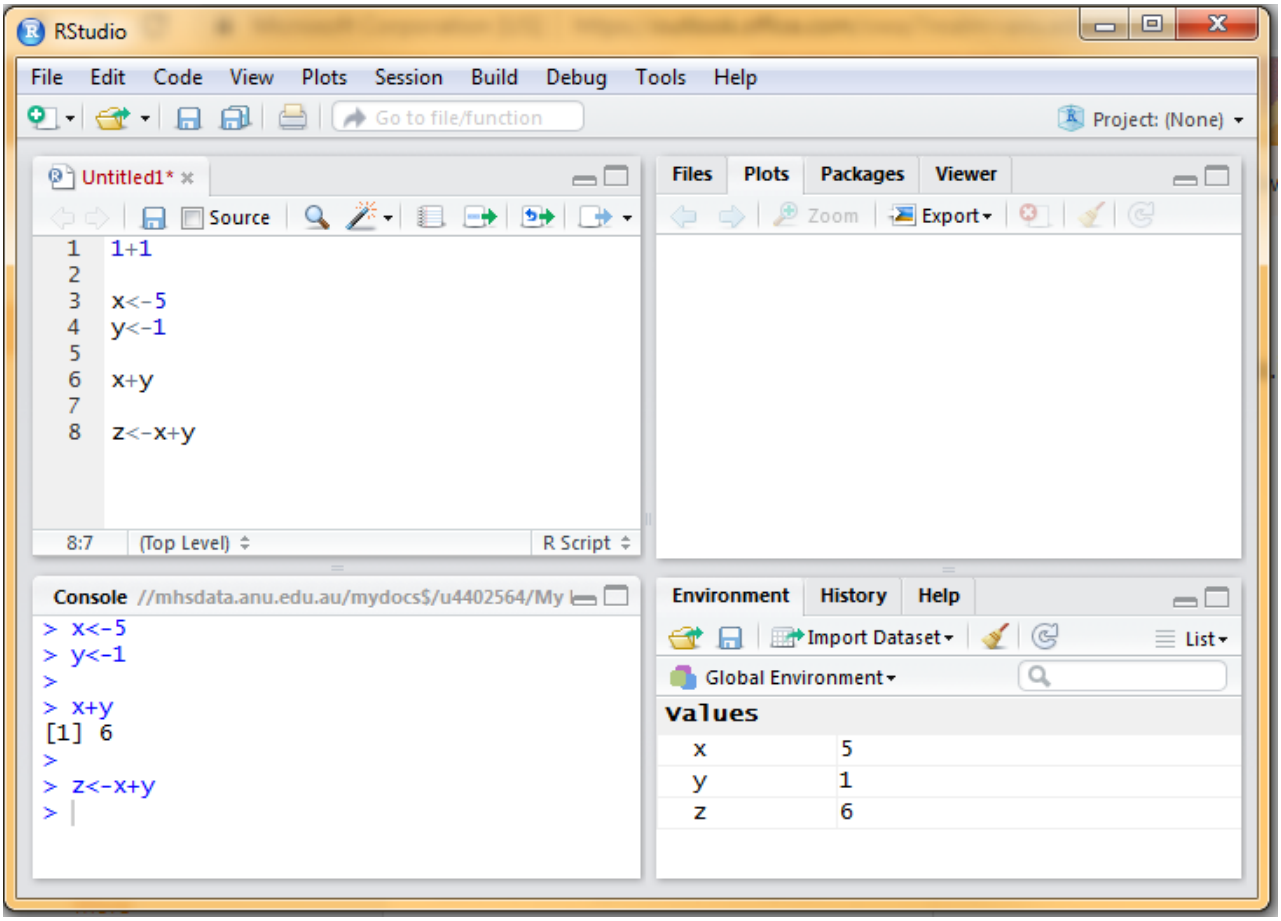
Variable assignment can be done via the <- operator (note they turn up in the global environment)



You can add variables together like they're numbers.



This means you can easily change one value earlier in the code to update your outcomes.

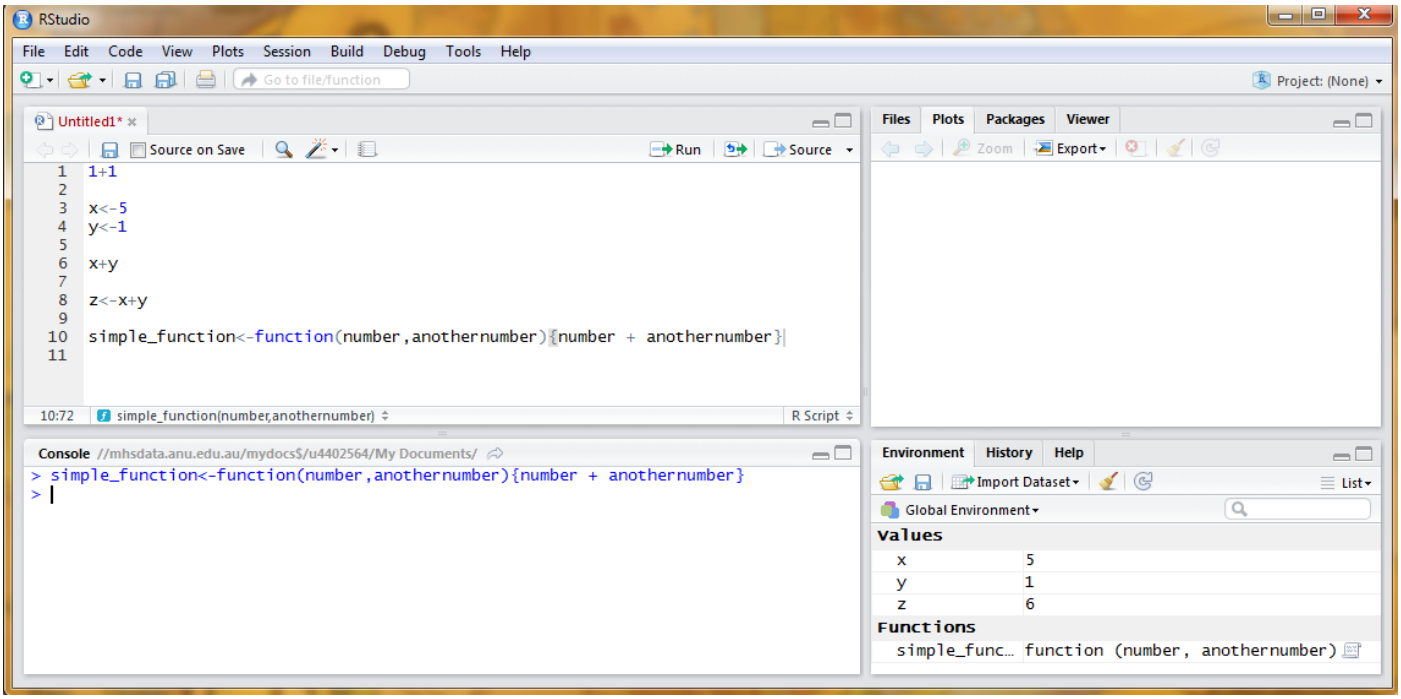


Functions

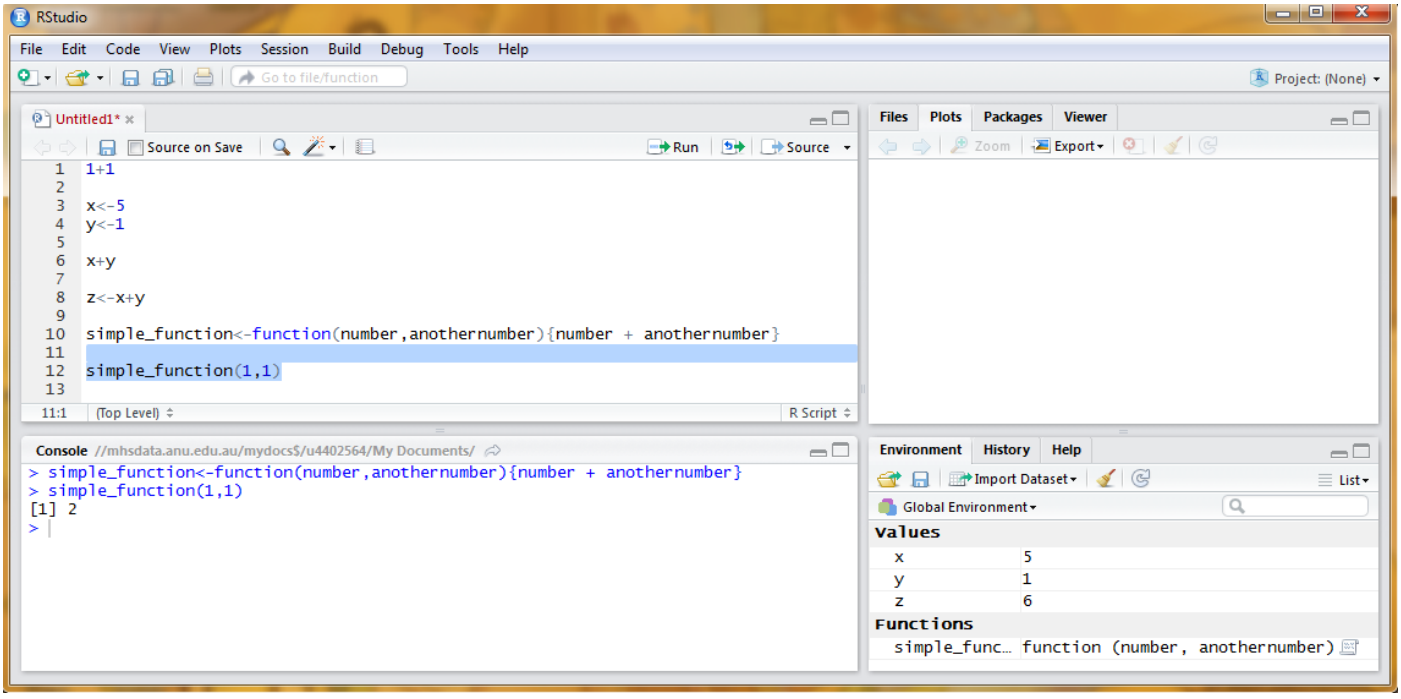
All functions are basically of the form:

```
function_name <- function(input1, input2, ... ){
        statements
        return(output)
        }
```

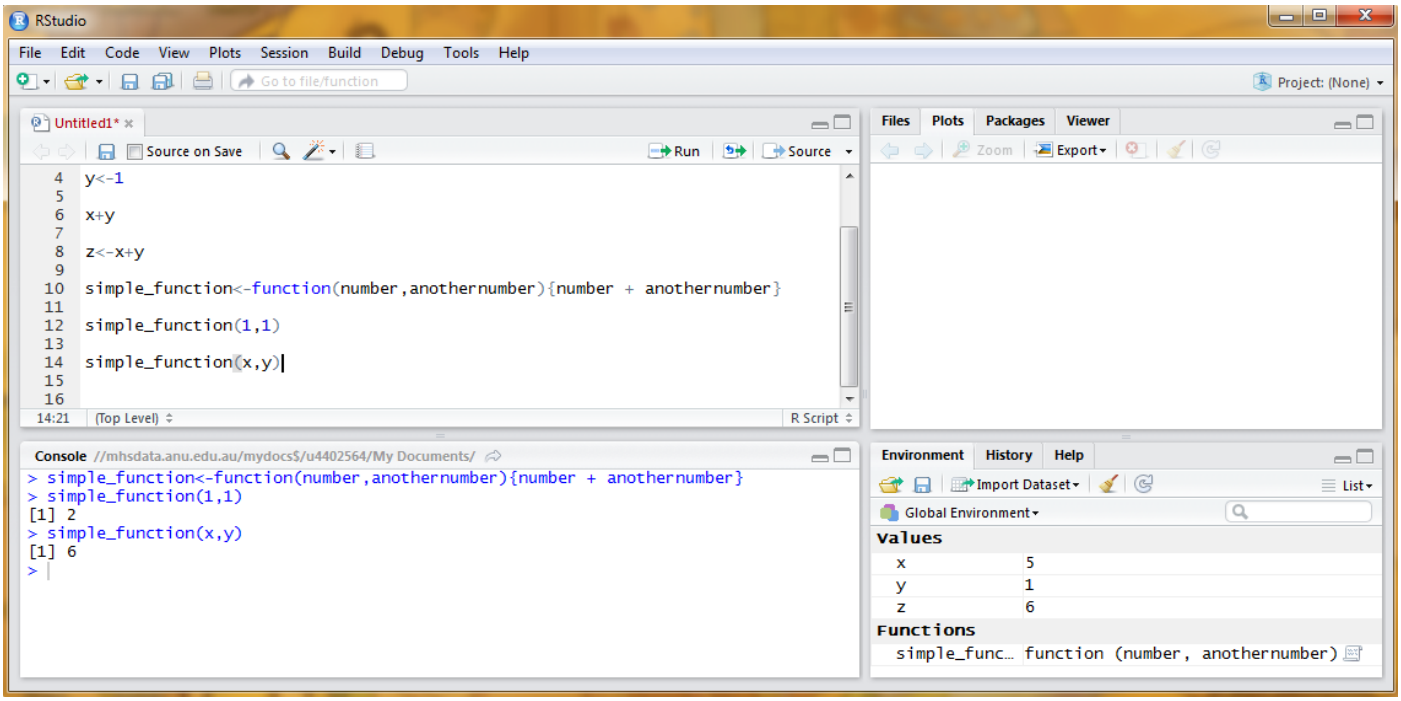
Write a simple function for adding two numbers.



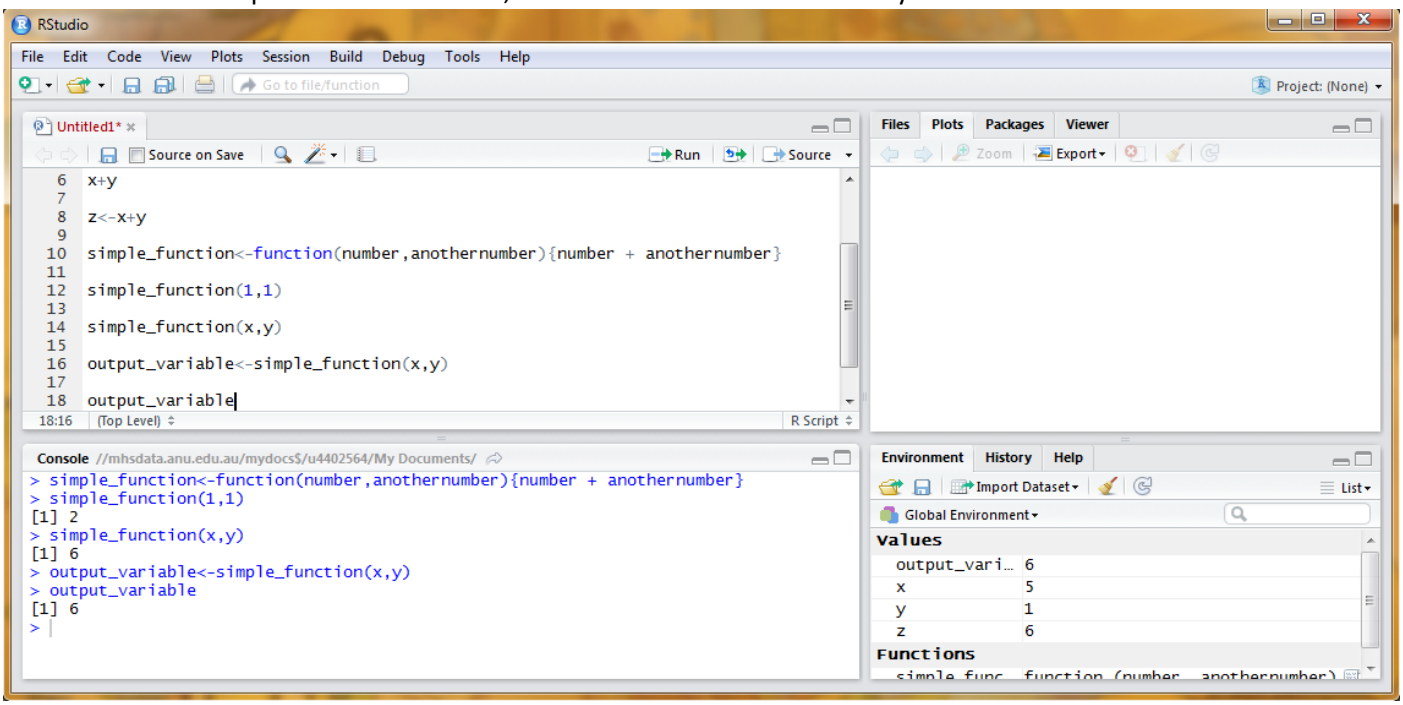
... and call it.



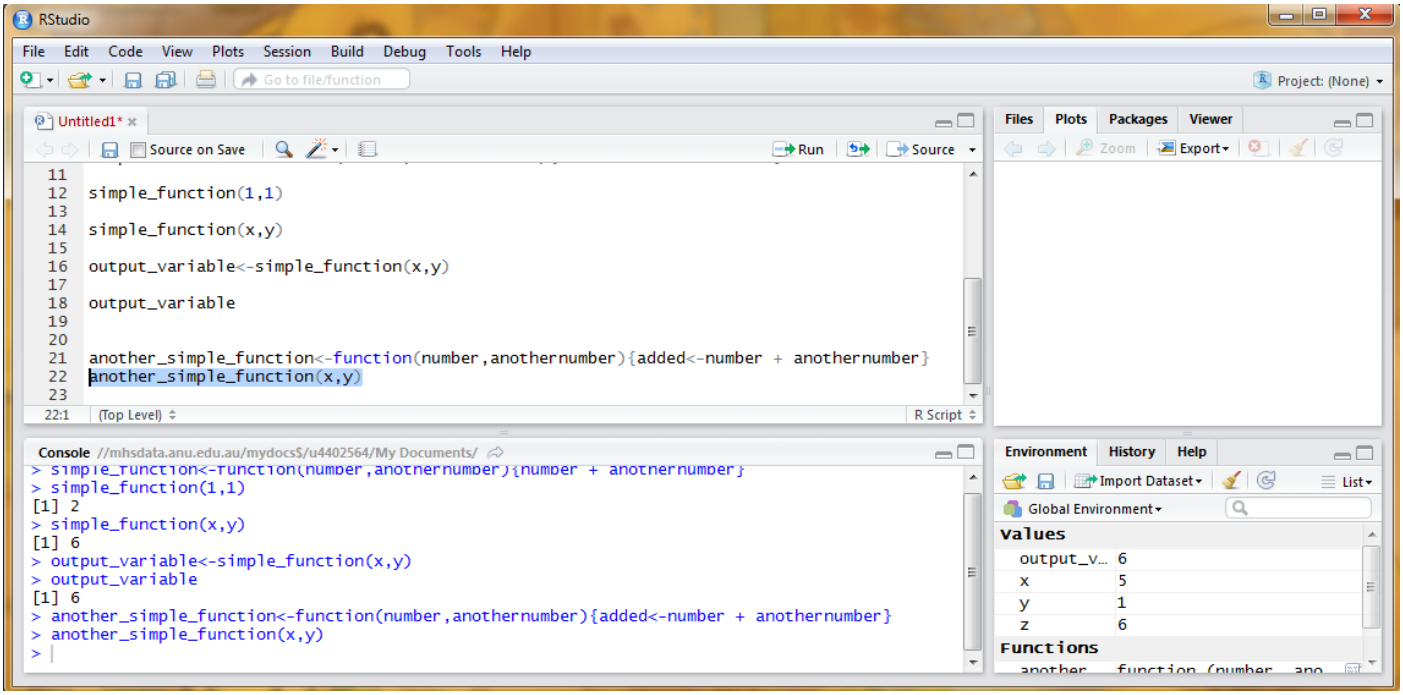
These functions can also be used with those variables you specified before.



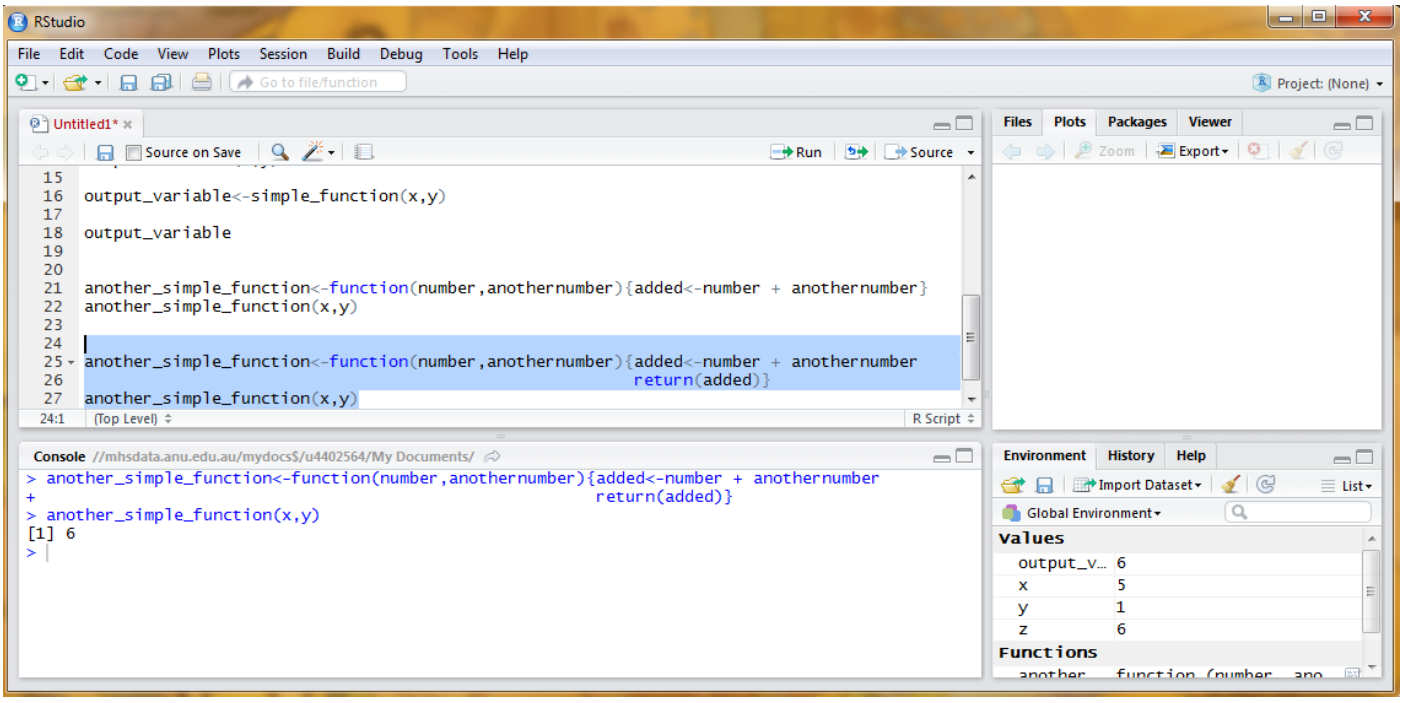
You can save the output as a new variable, which can then be called like any other variable.



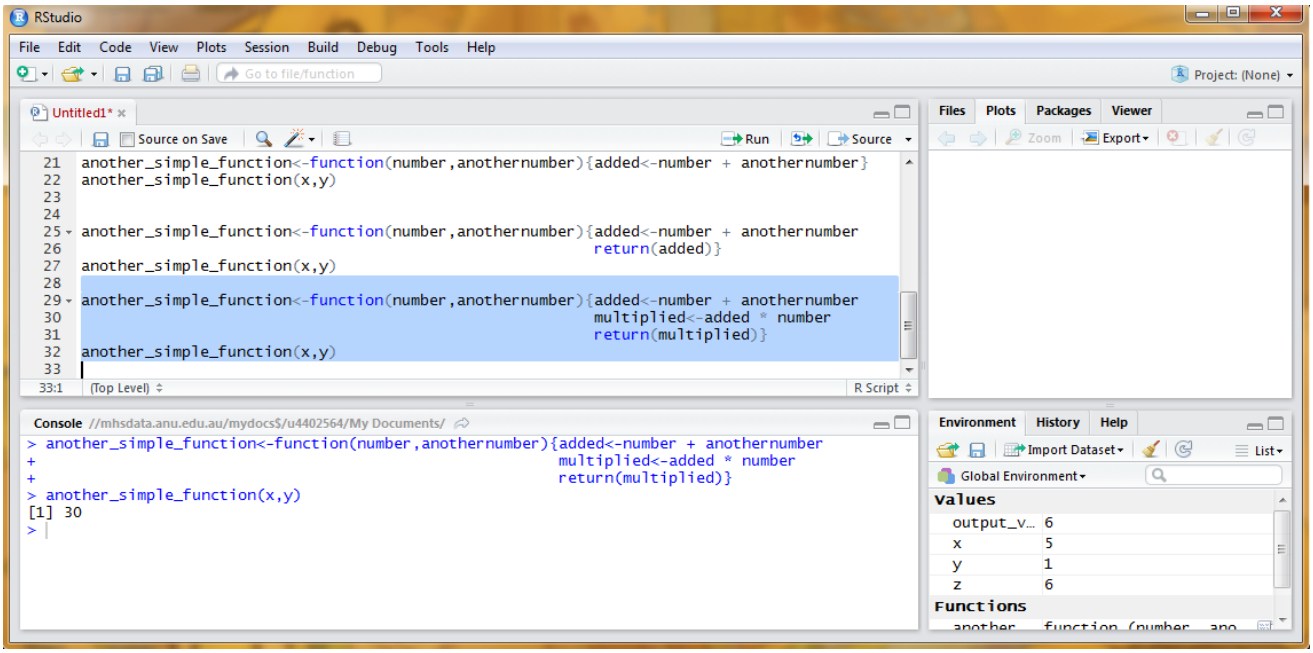
You can assign the operation inside the function to a variable, but unless you tell the function to return it you won't get anything out.



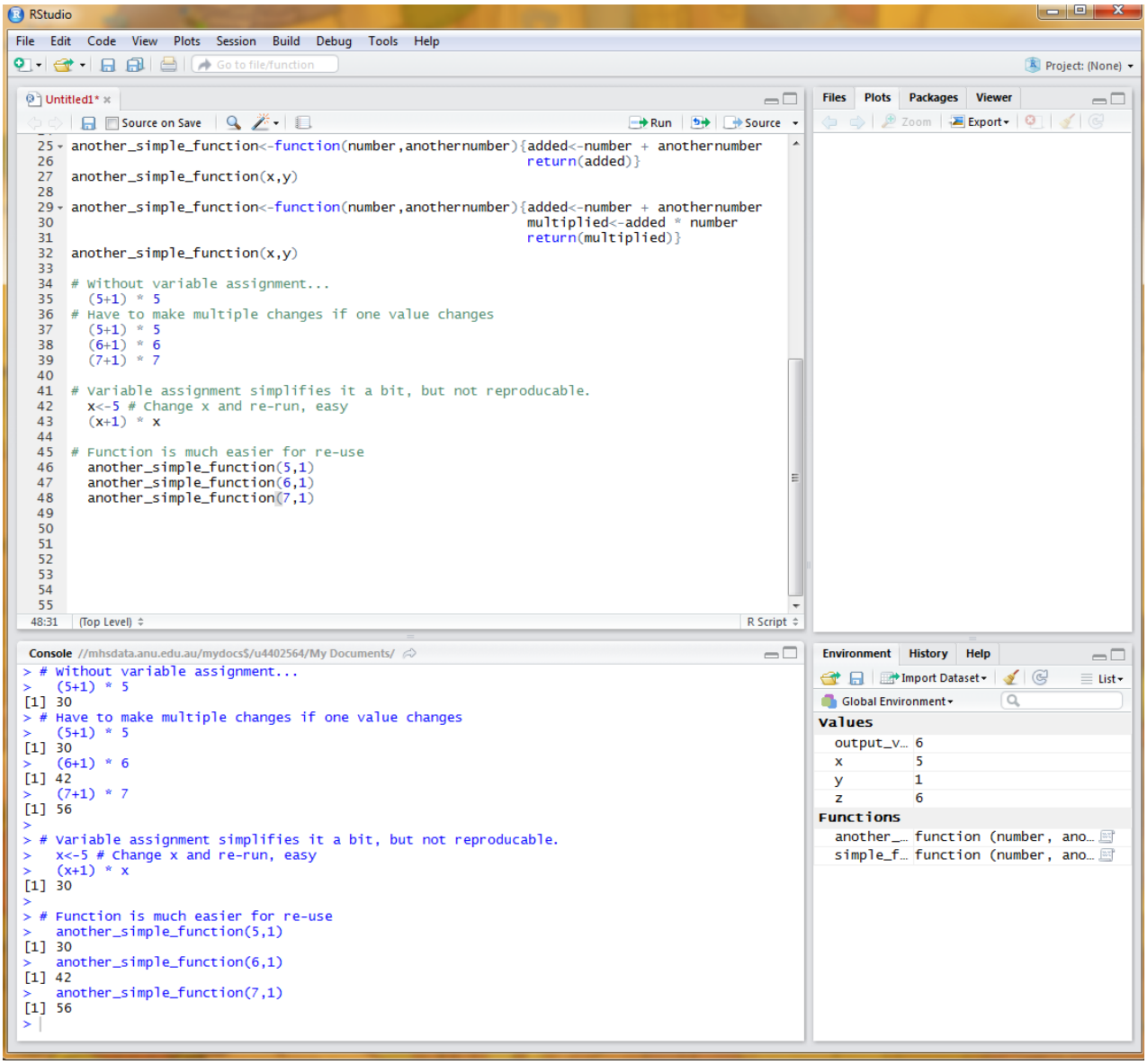
Need to add a return() argument.



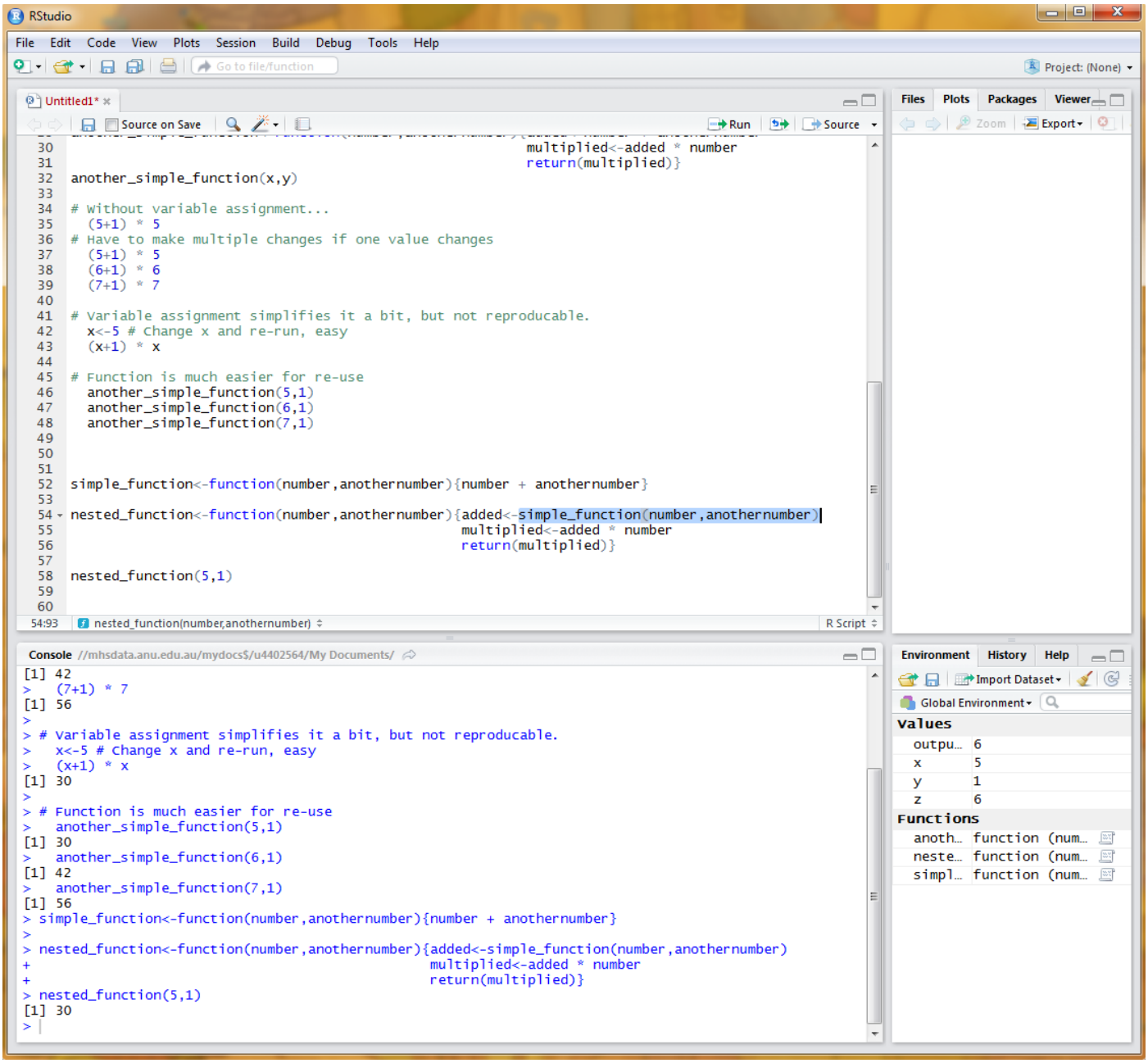
The key thing you can do in functions is adding multiple operations



You can see how functions are the pathway to complex, reproducible operations :D

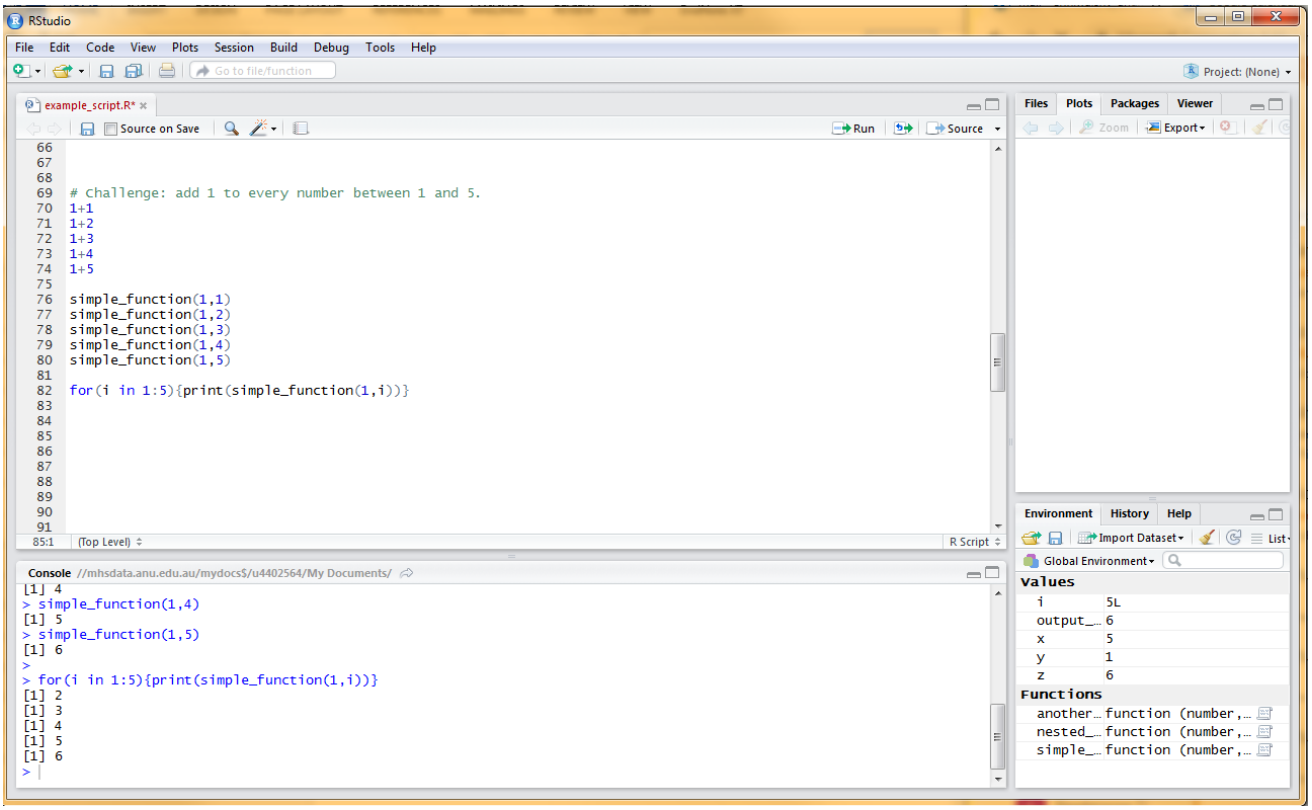


Also note, you can nest functions inside other functions for convenience.

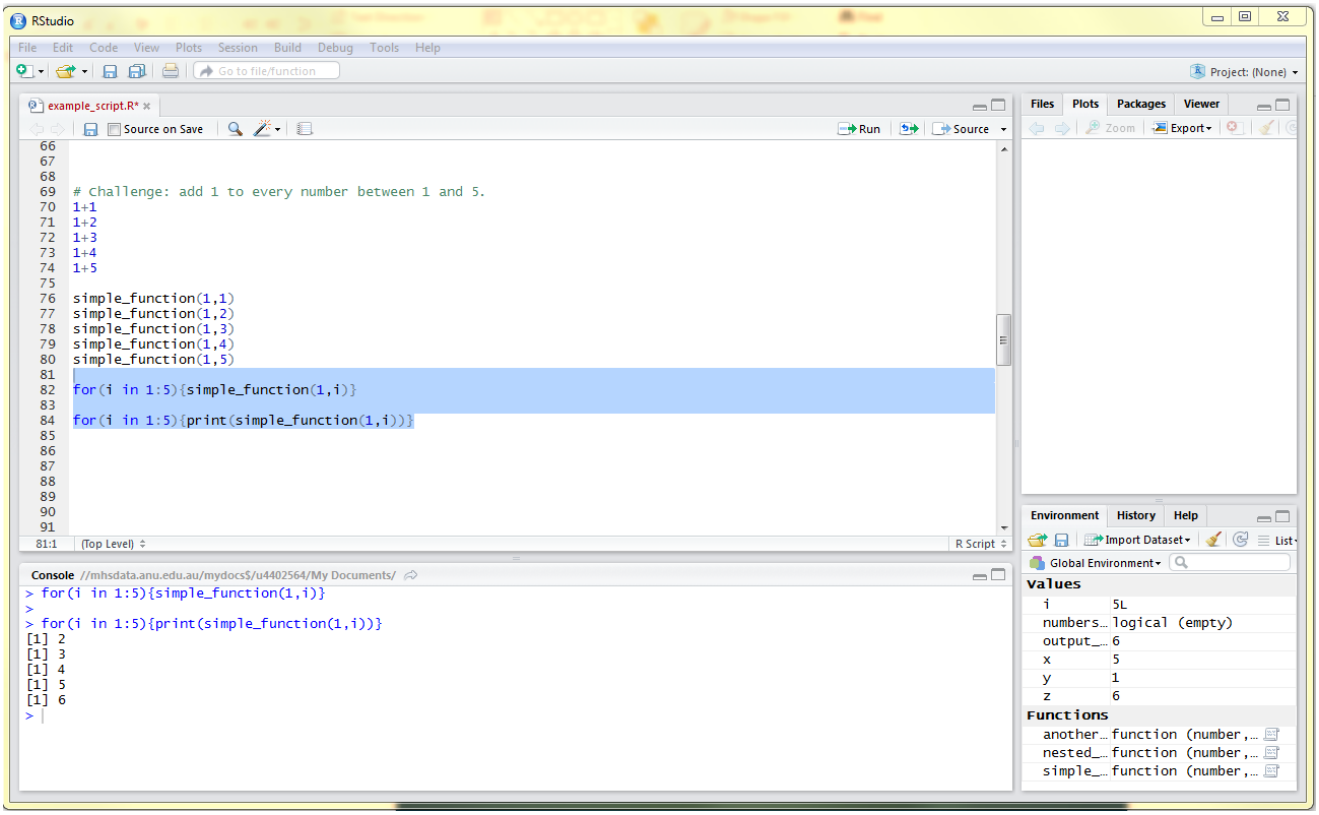


Loops

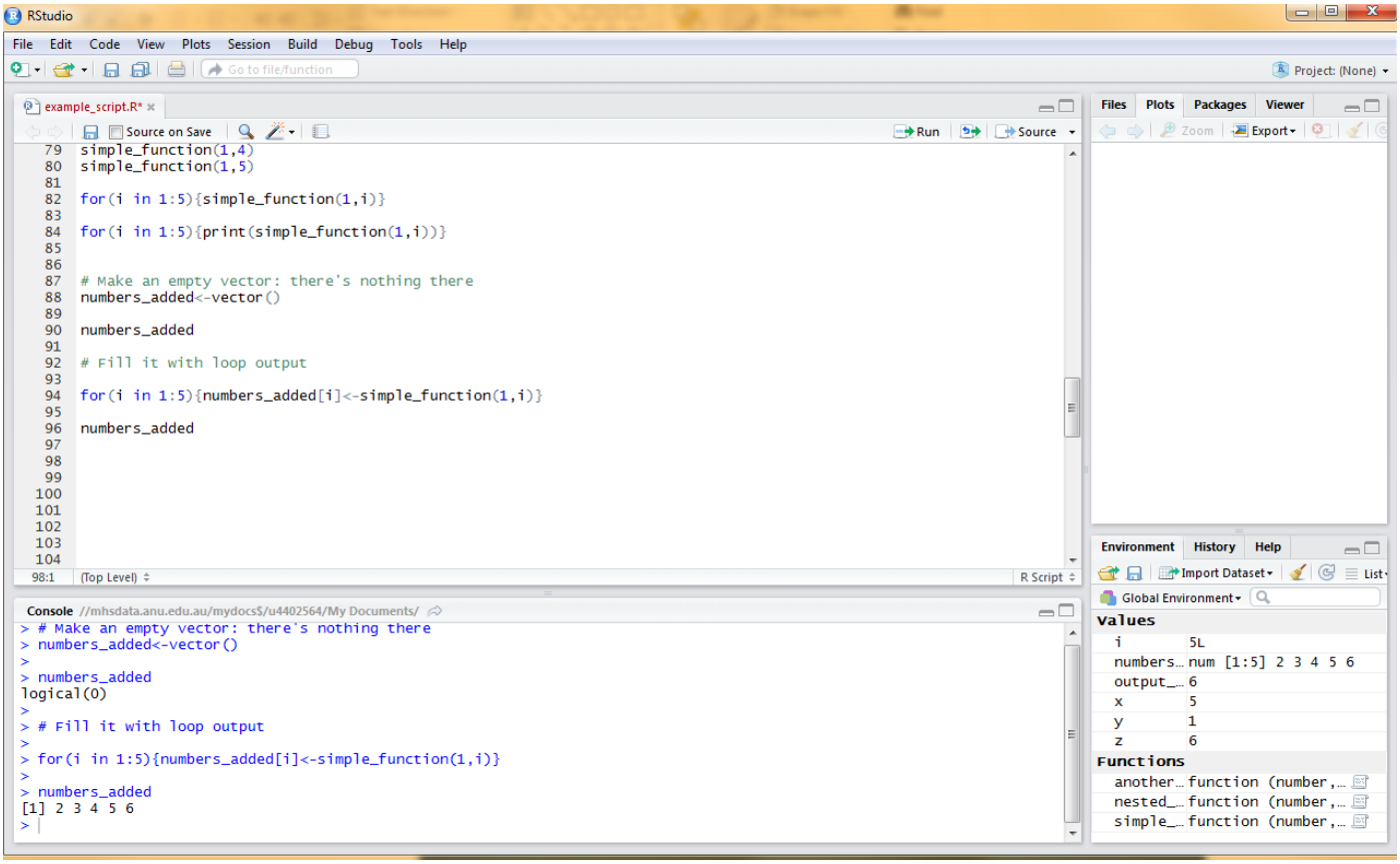
A loop is a sequence of instructions that is continually repeated until a certain condition is reached, of the generic form `for(iteration in number of times) {do something}`. Typically, we use a lowercase `i` to refer to the specific iteration. So, if we want a loop to run 5 times, we specify `i` in 1:5, and then allow the second argument to our `simple_function` to be `i` - this means each time the loop progresses, that `i` will become 1, 2, 3,4, then 5.



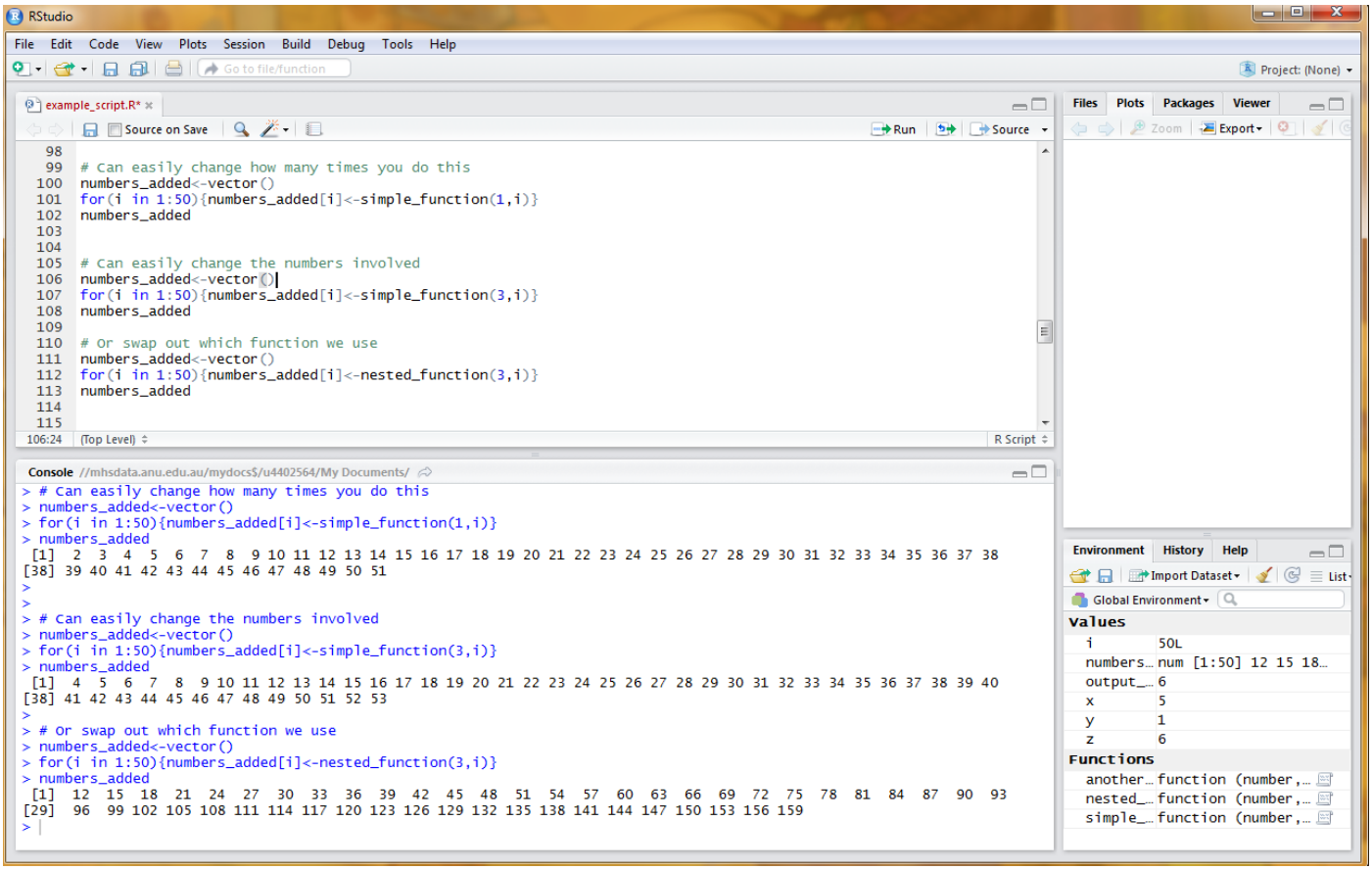
Like a function, it will not return anything if you don't tell it to specify an output. So, you need to either tell it to print out using the `print()` function (this prints the output to console)



Or fill up an empty variable. You build an empty variable by assigning it an empty vector value, used as `vector()`.

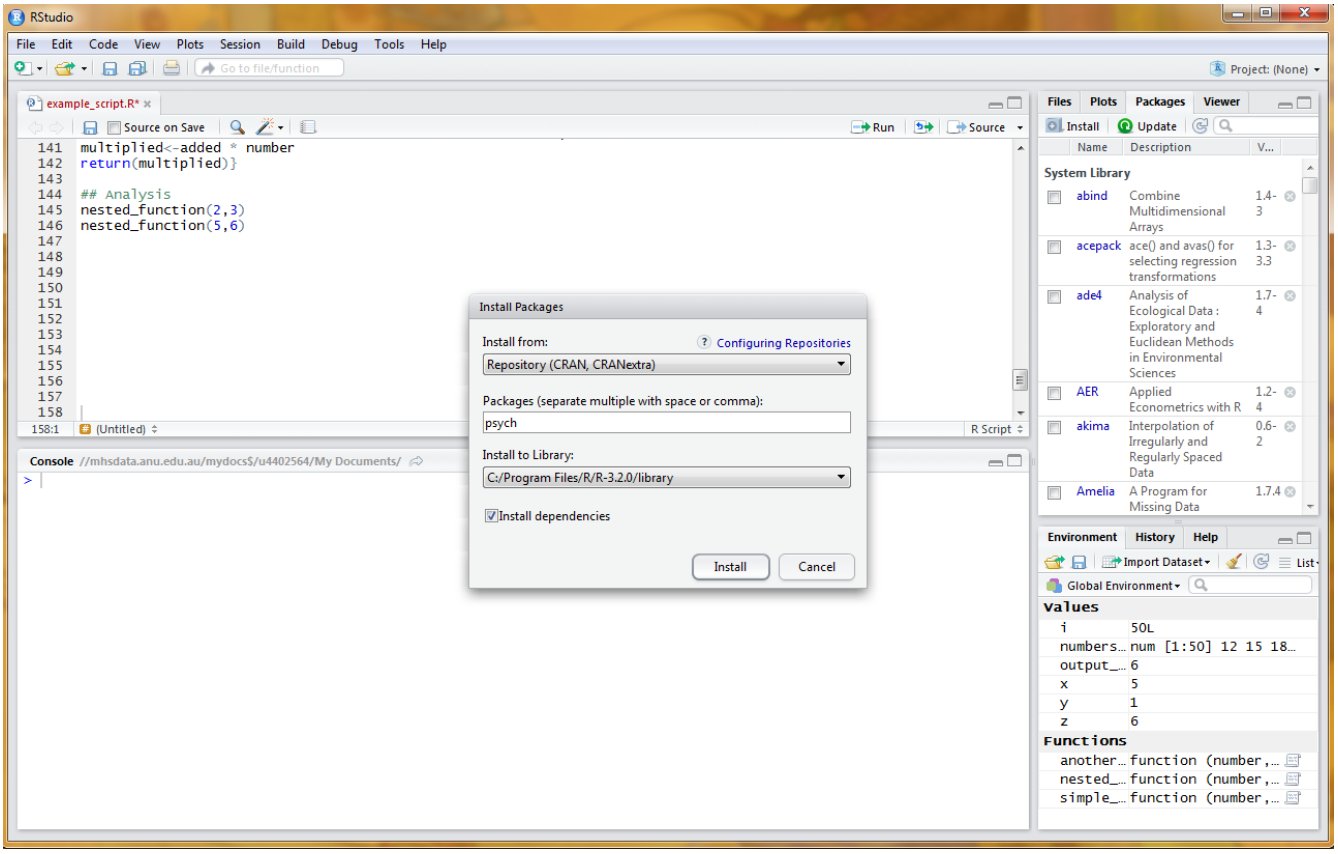


Now we have basic loop, you can see how easy it is to change the parameters. Changing the number in the `for` statement changes how many times the loop runs. You can change the number you pass to the loop. And, of course, you can change the function inside the loop.

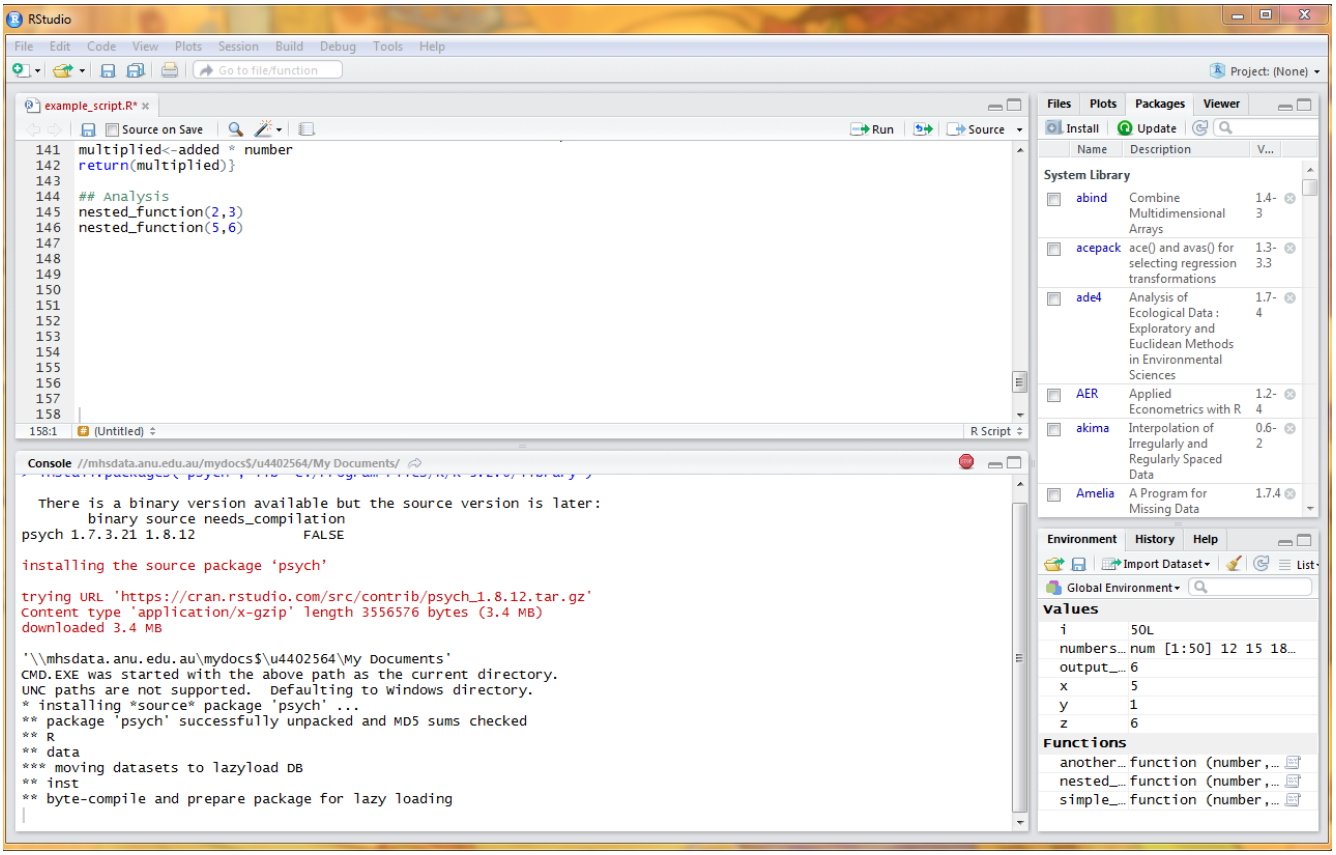


Packages

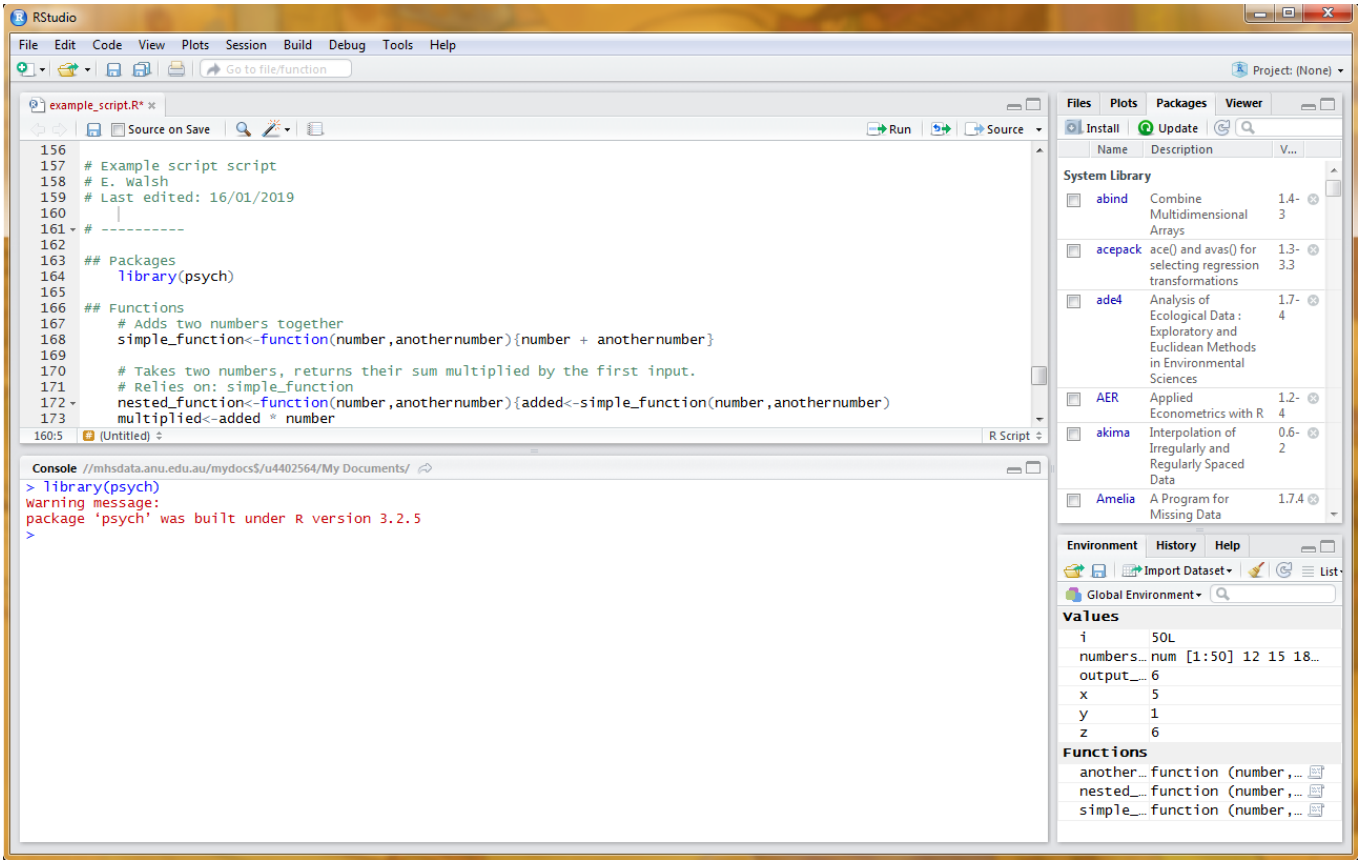
You can think of packages as collated collections of functions other people have written. To install, in Rstudio go to 'packages', 'install', then type the name of the package you want. For example we're going to install 'psych', it has basic descriptive statistics.



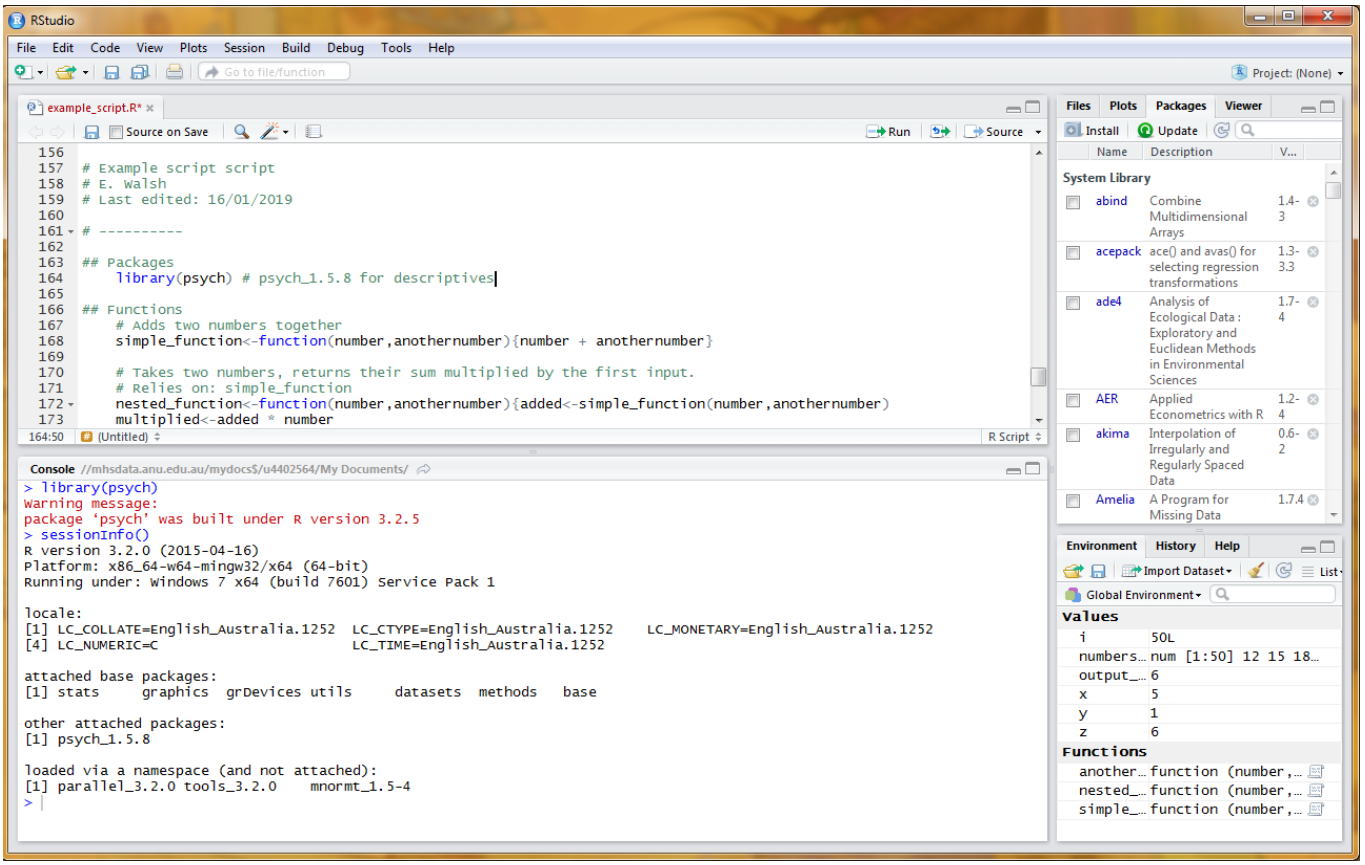
Console will show it loading.



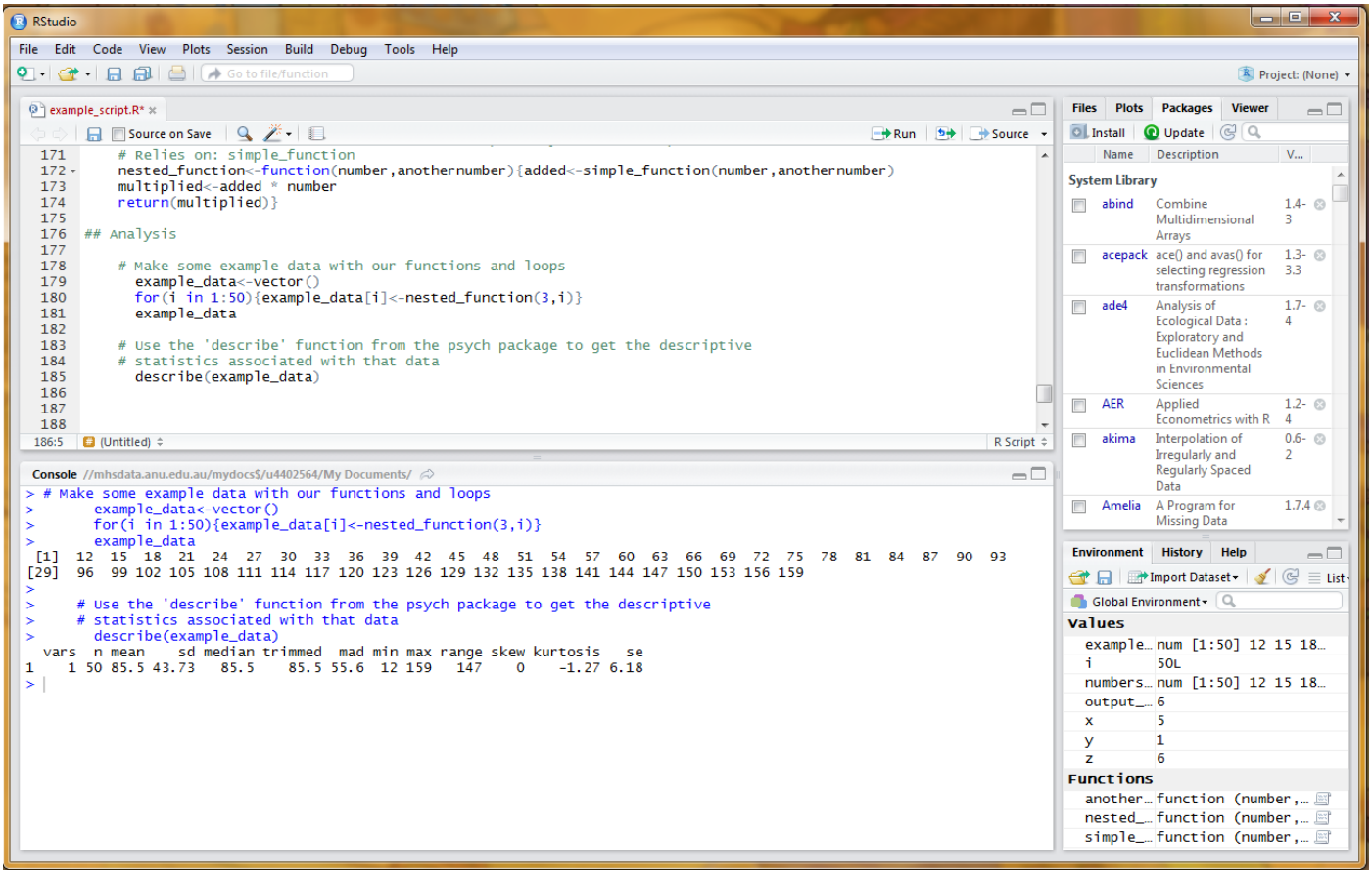
Once you have it installed (only need to do it once) you load it (ideally at the top of the script) using the call `library(psych)`.



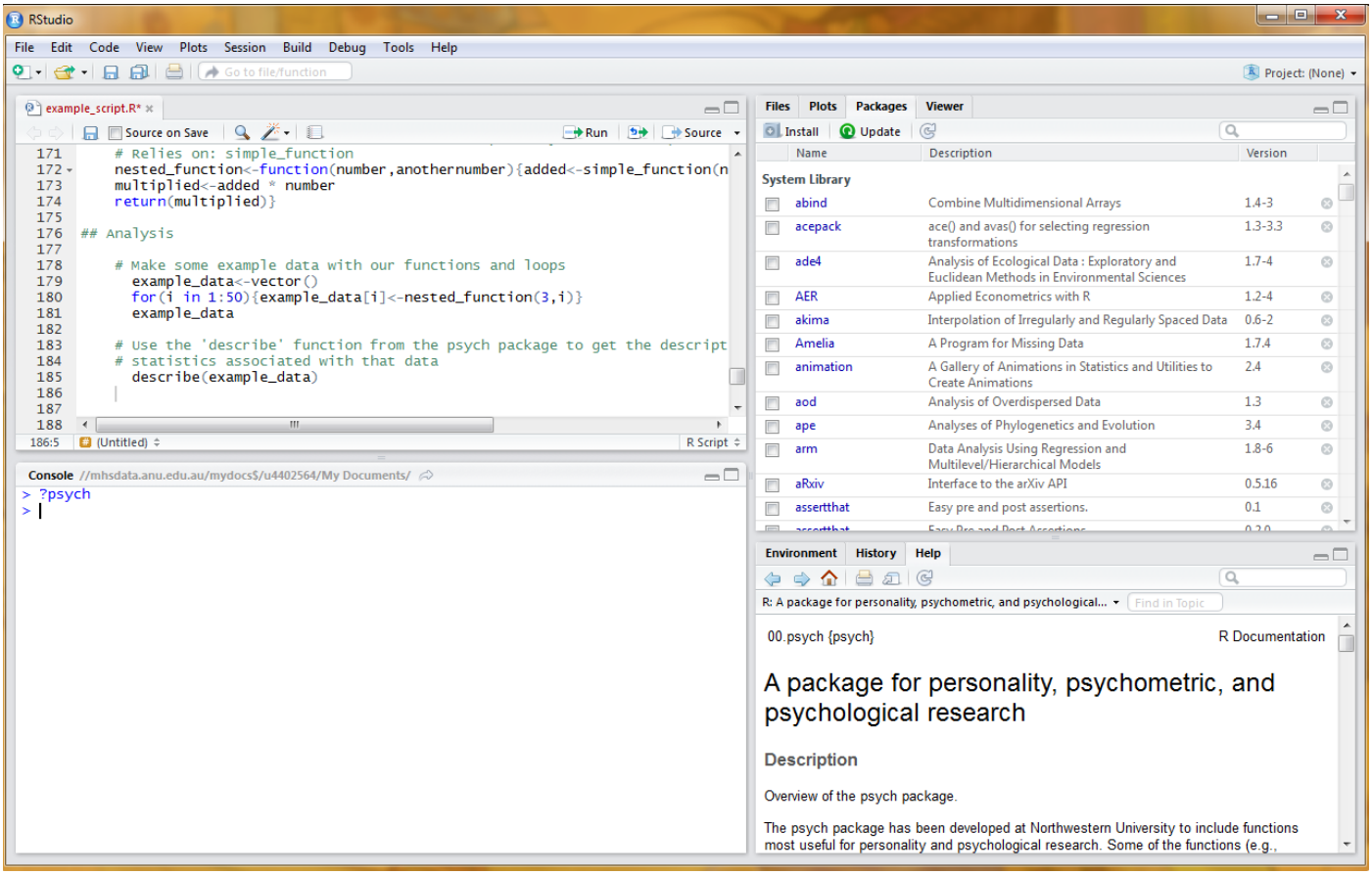
Package versions can change all the time, and you may well end up loading a lot of them, so it is best practise to include (a) what it was for, and (b) version information in the comments near the package. You can obtain this information by typing `sessionInfo()` in the console.



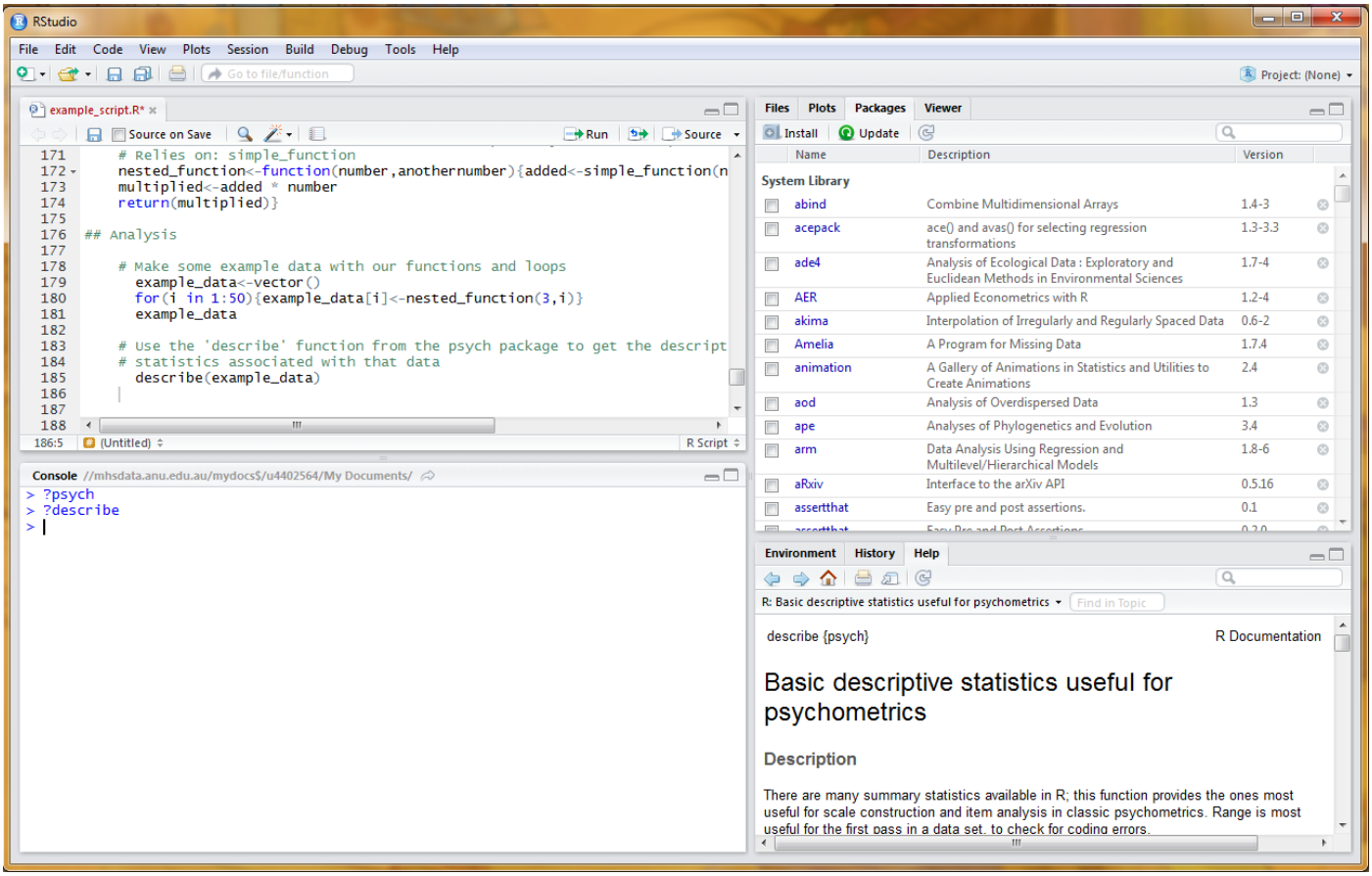
Now we can use all the functions that package has to offer. For example, let's use the `describe()` function to grab some descriptive statistics.



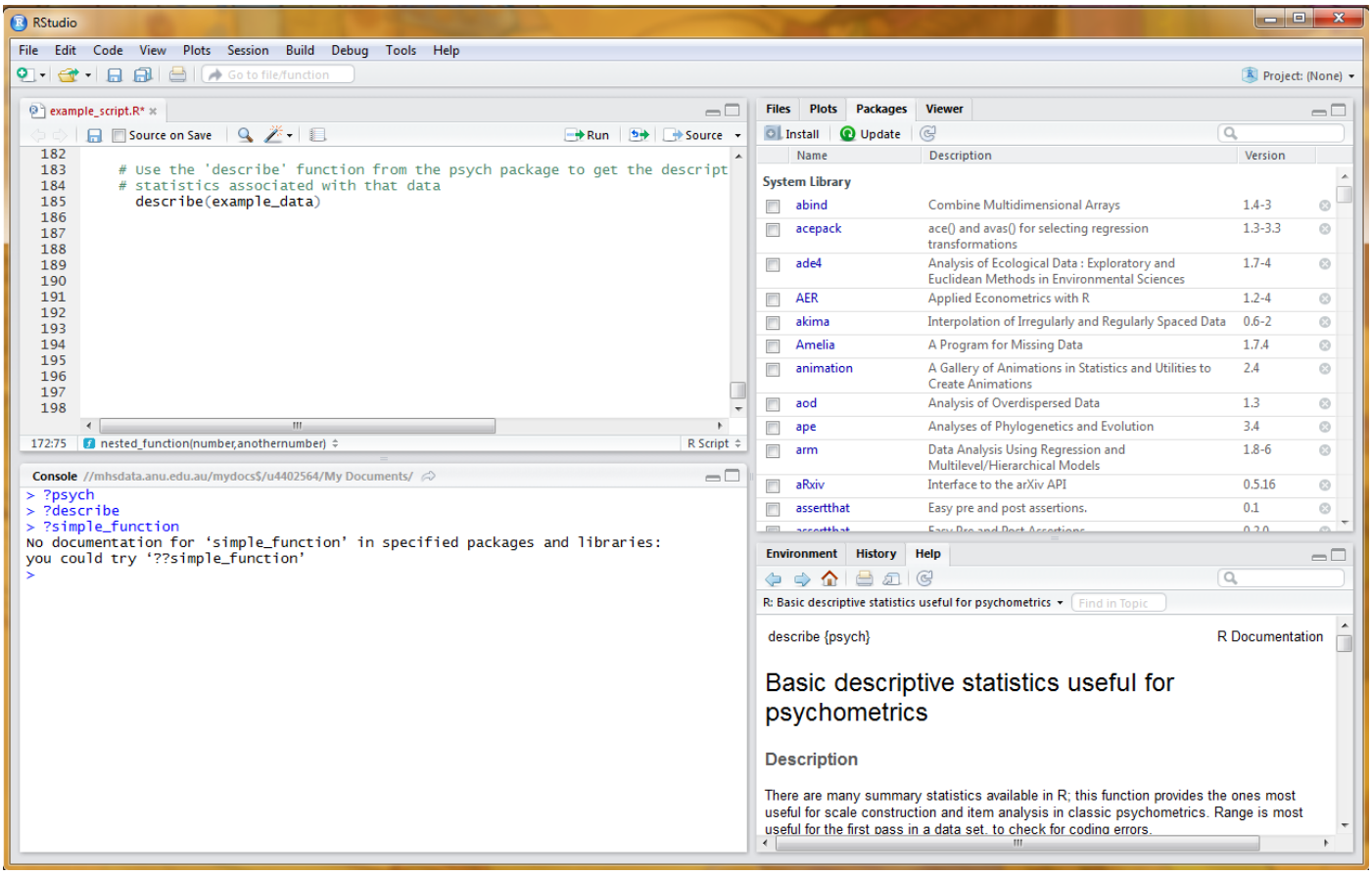
Information about packages can be found online, but also in R. To find help, use a question mark and the package's name, like the `?psych` example here. Help comes up in the 'help' window.



You can do the same for functions within a package, such as `?describe`.

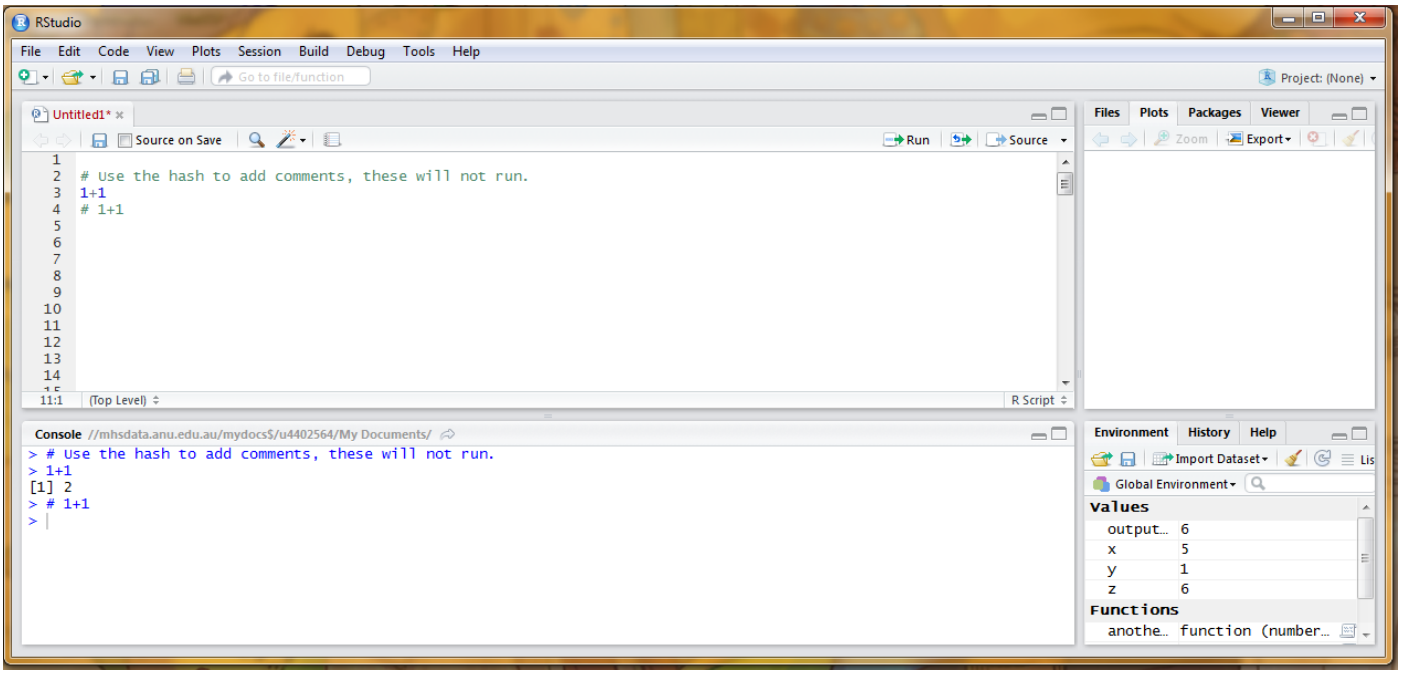


Of course, this doesn't work with the function you made up, as you've not written any documentation for it.

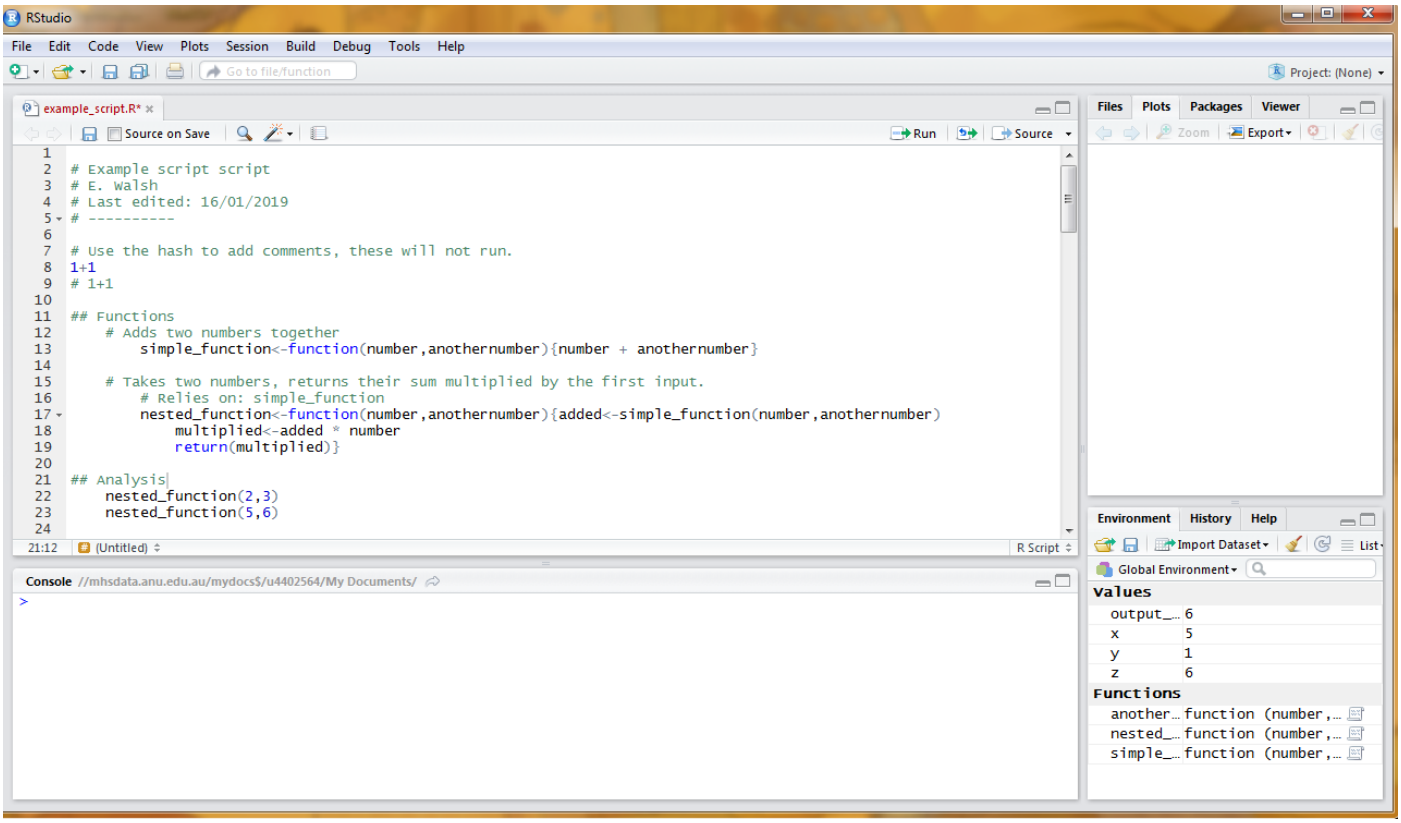


Scripts

First - you can use the # to add comments. These don't produce output in the console window, and are extremely useful for keeping track of precisely what you're doing.

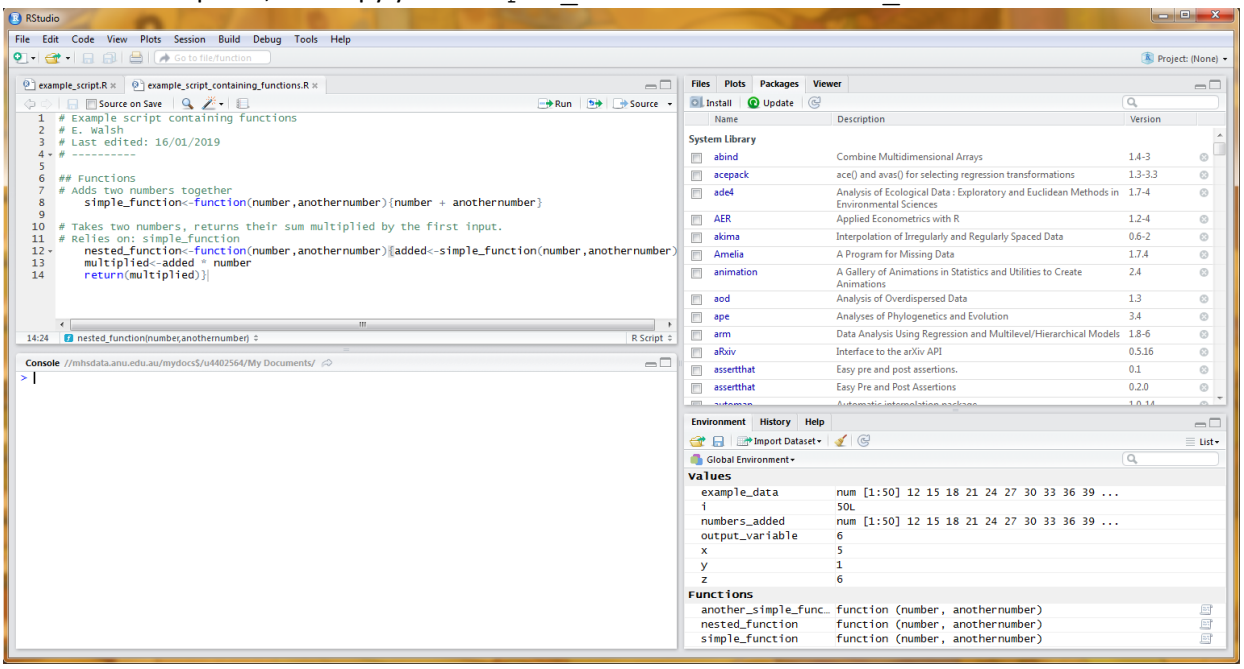


Good practise: lead with who wrote the script it, what it is for, and when you last meaningfully edited it. Then setup (functions up the top), and then your analysis. Use comments to remind you what you were doing, and indentation for readability. … Future you will be very grateful to past you.

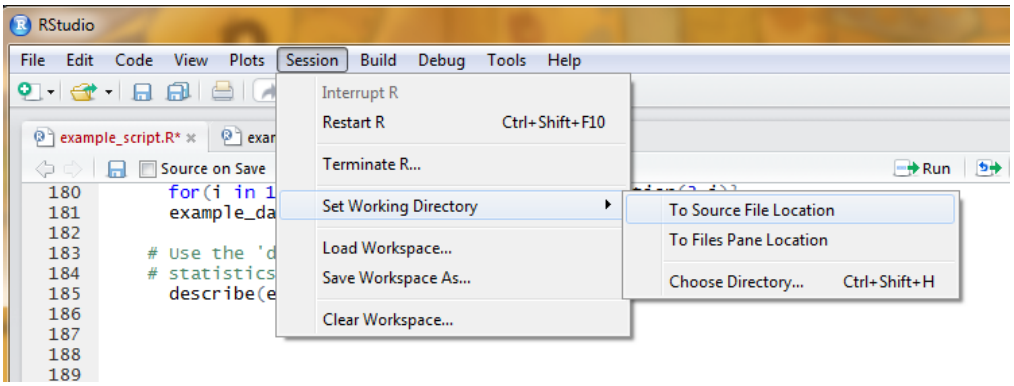


A common usage scenario is saving functions in a script.

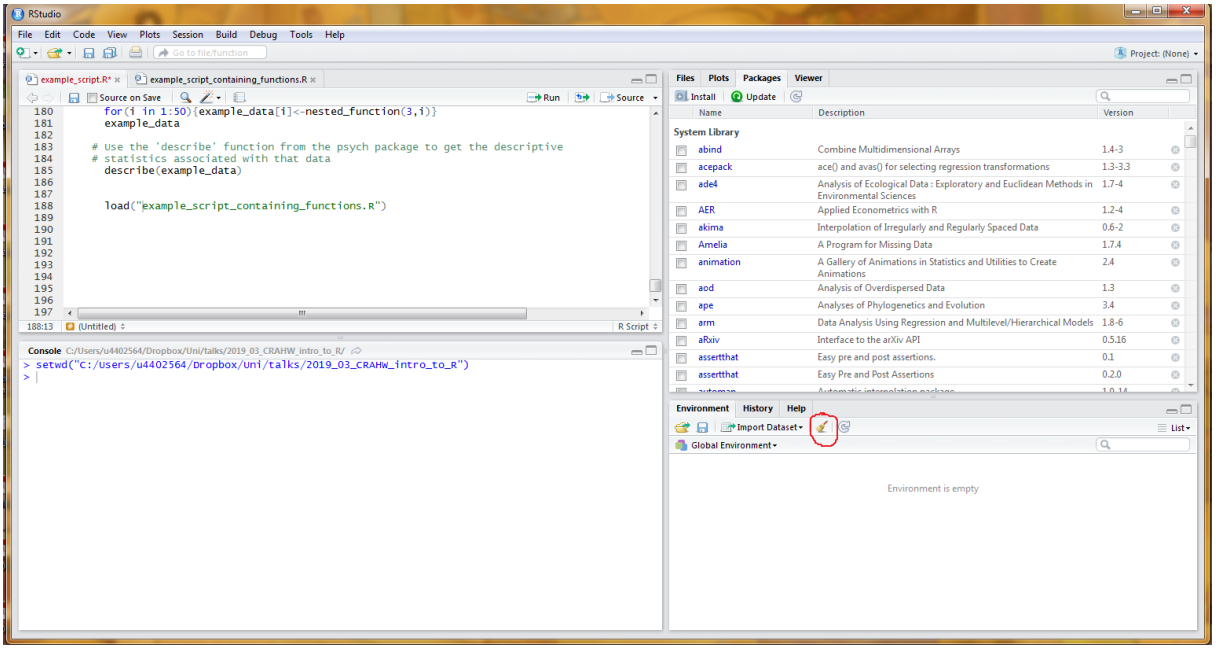
Make a new script file, and copy your `simple_function` and `nested_function` functions into it.



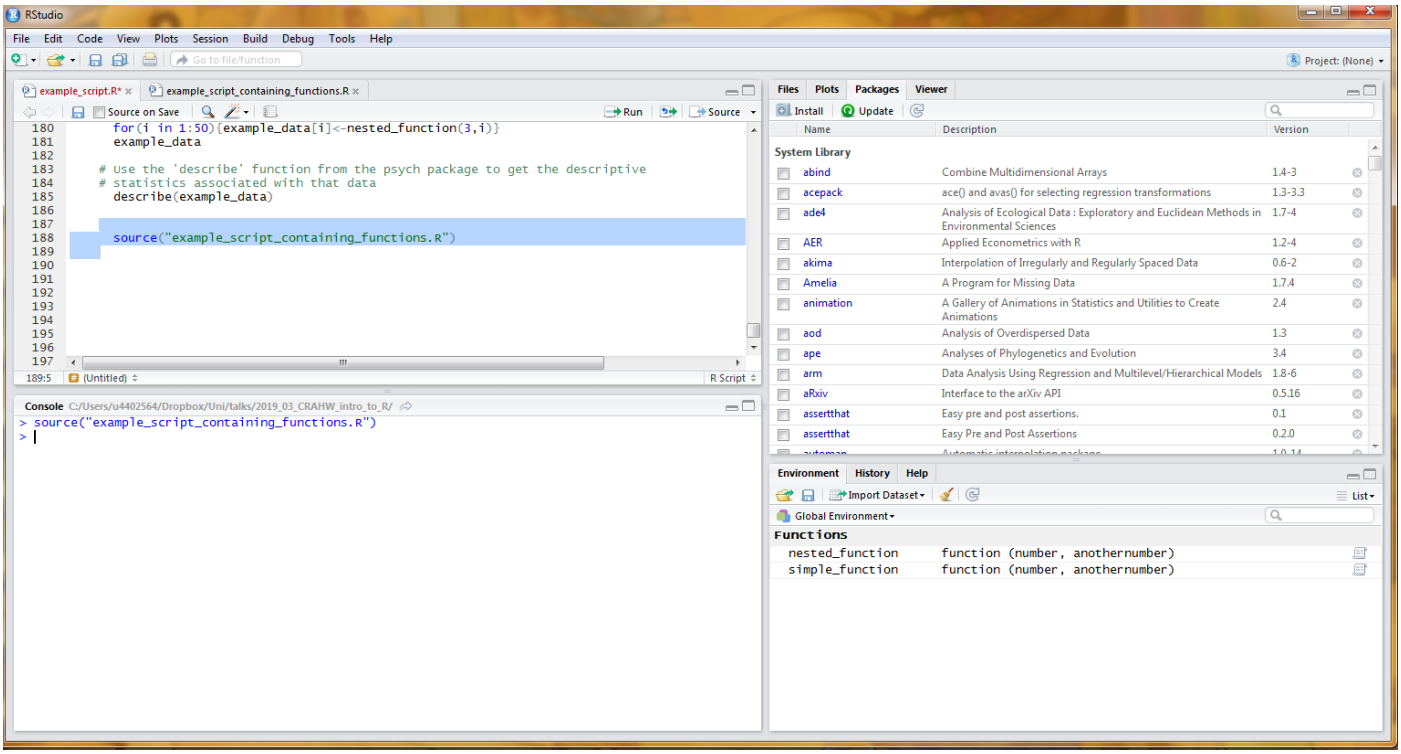
Go back to your first script, set the working directory.



To show this works, we're going to clear the environment. This tells R to forget all of the values and functions it currently knows: this does not clear away functions belonging to packages.



Now, we 'source' our script. See how the functions now show up in the environment.

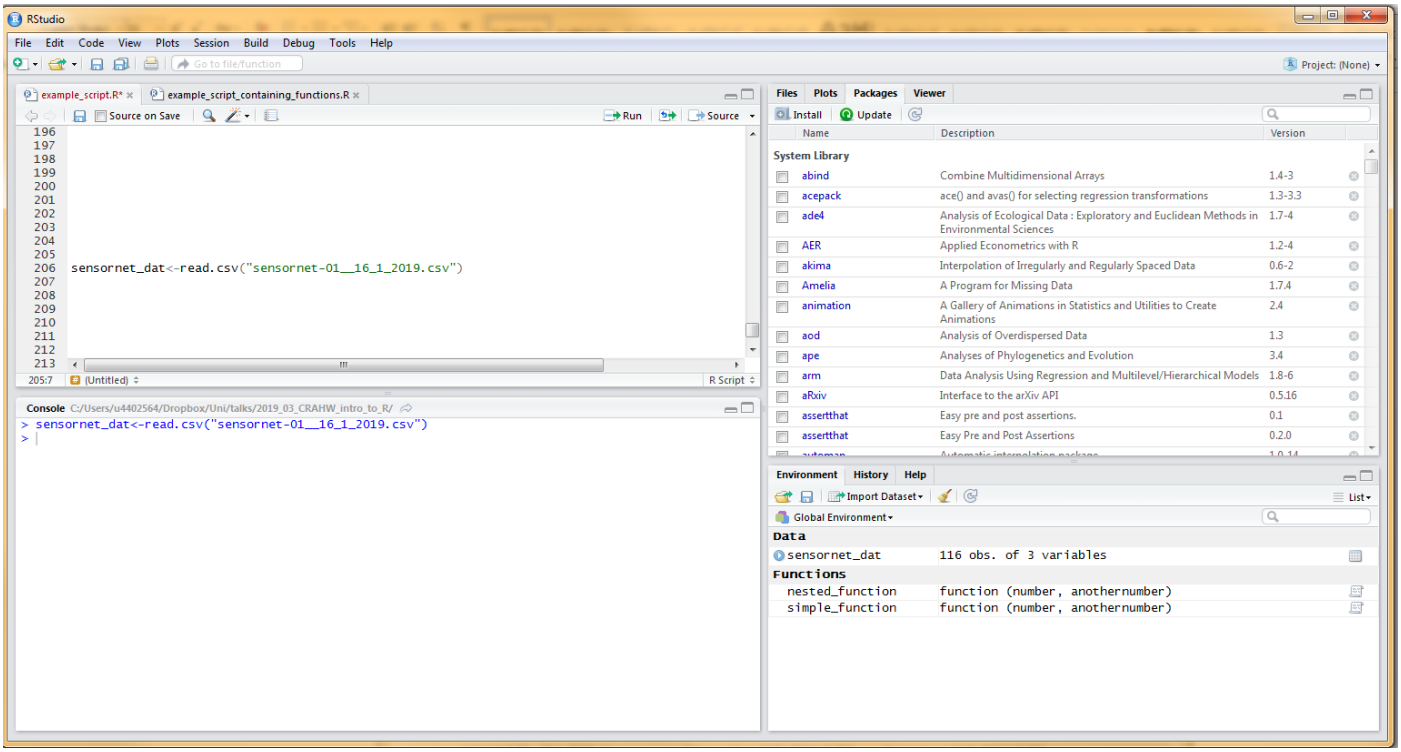


Importing and working with datafiles

Very much like scripts: make sure you have set your working directory so R knows where to look for your data files.

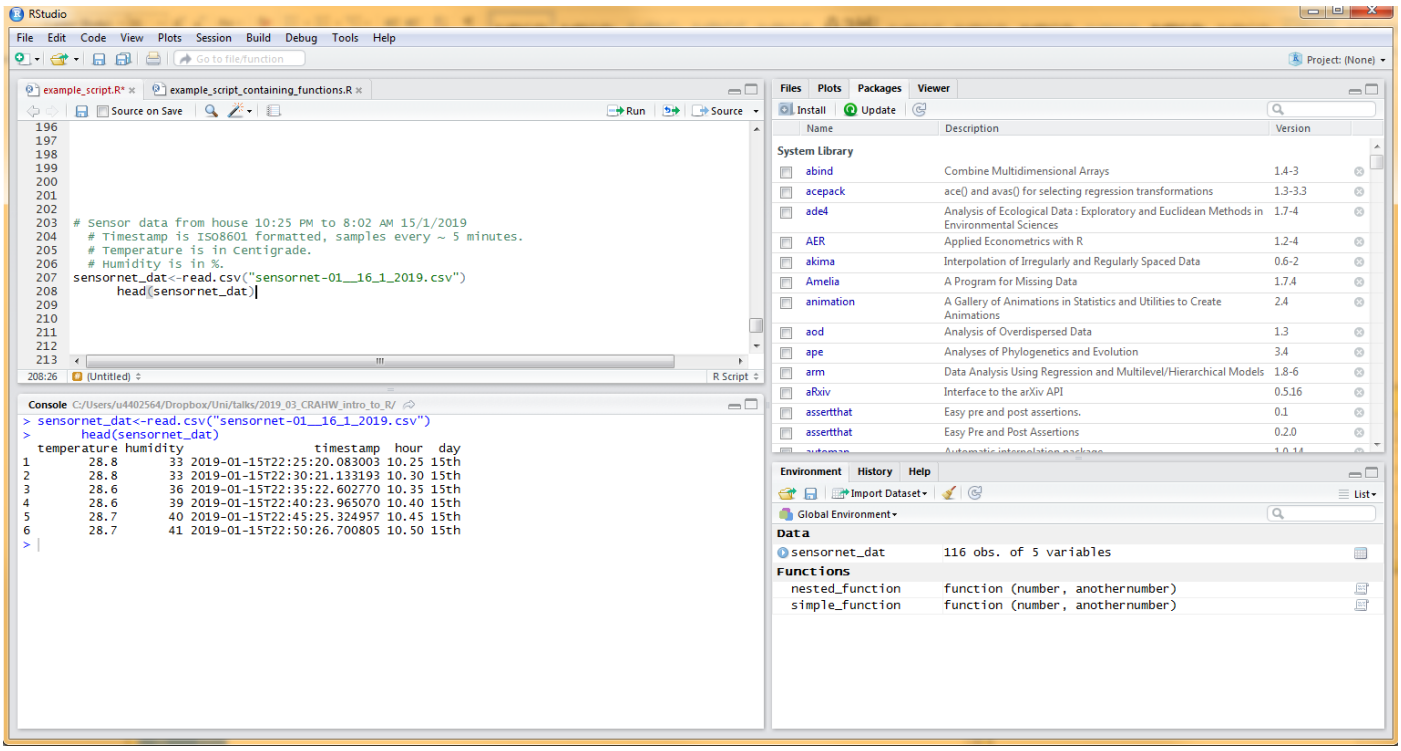
Assign the data to a variable: if you don't it just prints in console. Here it is a .csv so we're using '`read_csv`'; can also read excel and SPSS files with the help of a package (too much detail for today).

It is best practise to append `_dat` to the variable name, so you always know which variables are data files.

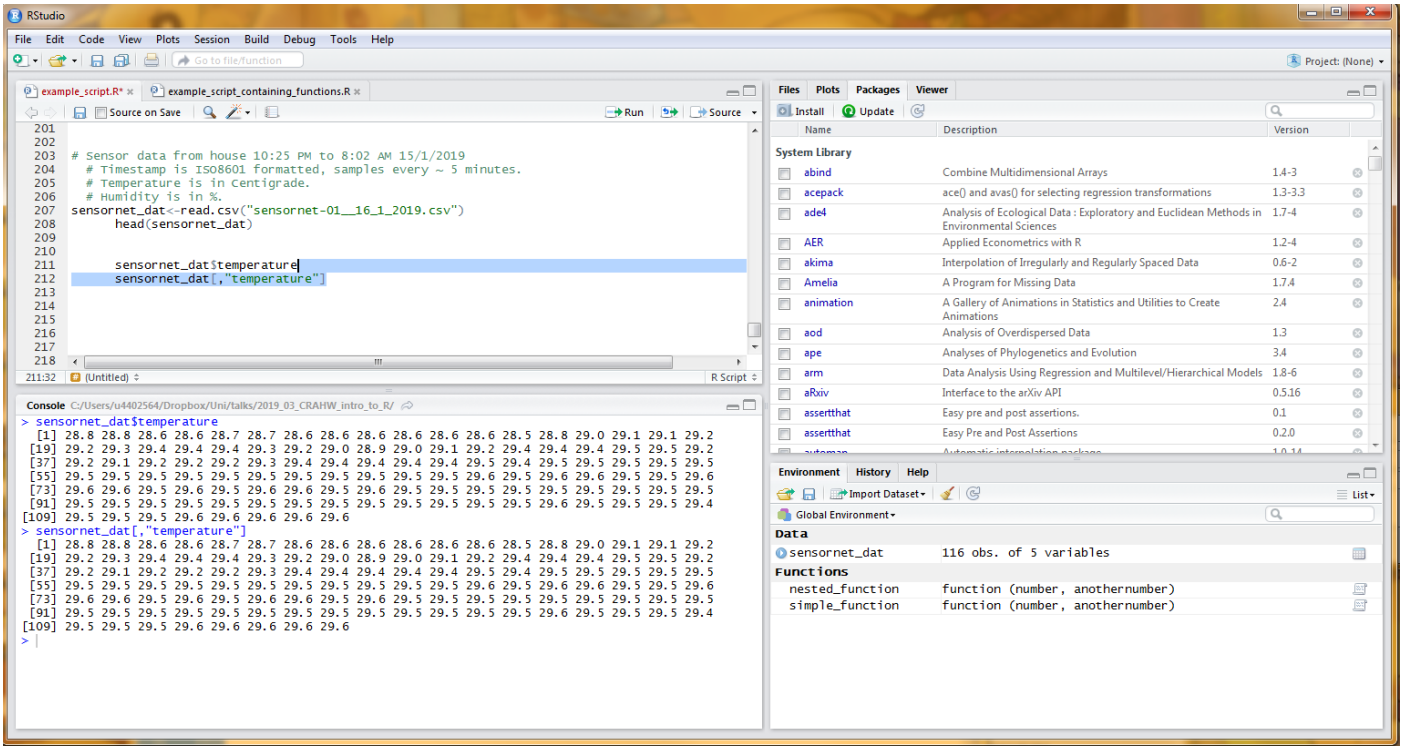


ALWAYS use the 'head' function to look at the first few rows of the data file to ensure it has loaded correctly.

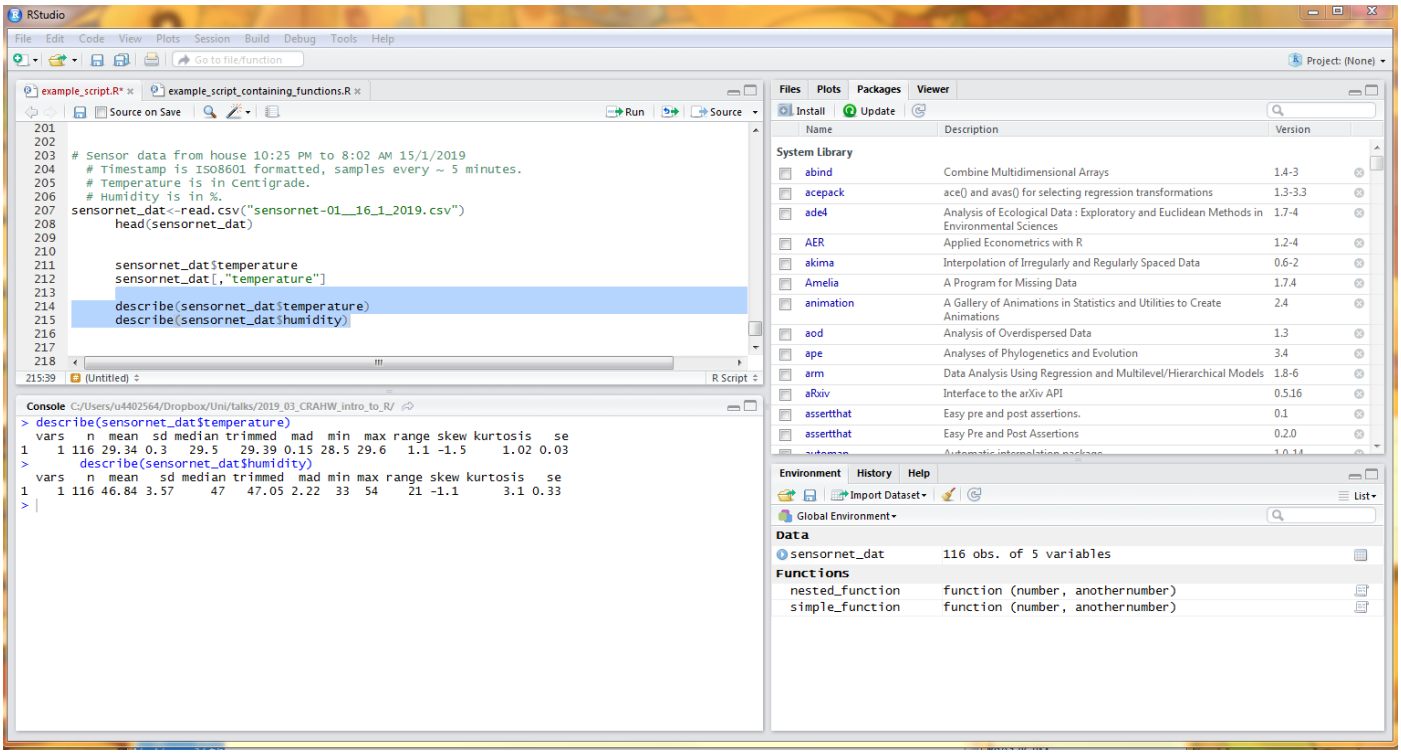
Also, it is best practise at this point to include descriptions of the dataset and each variable in comments for future reference.



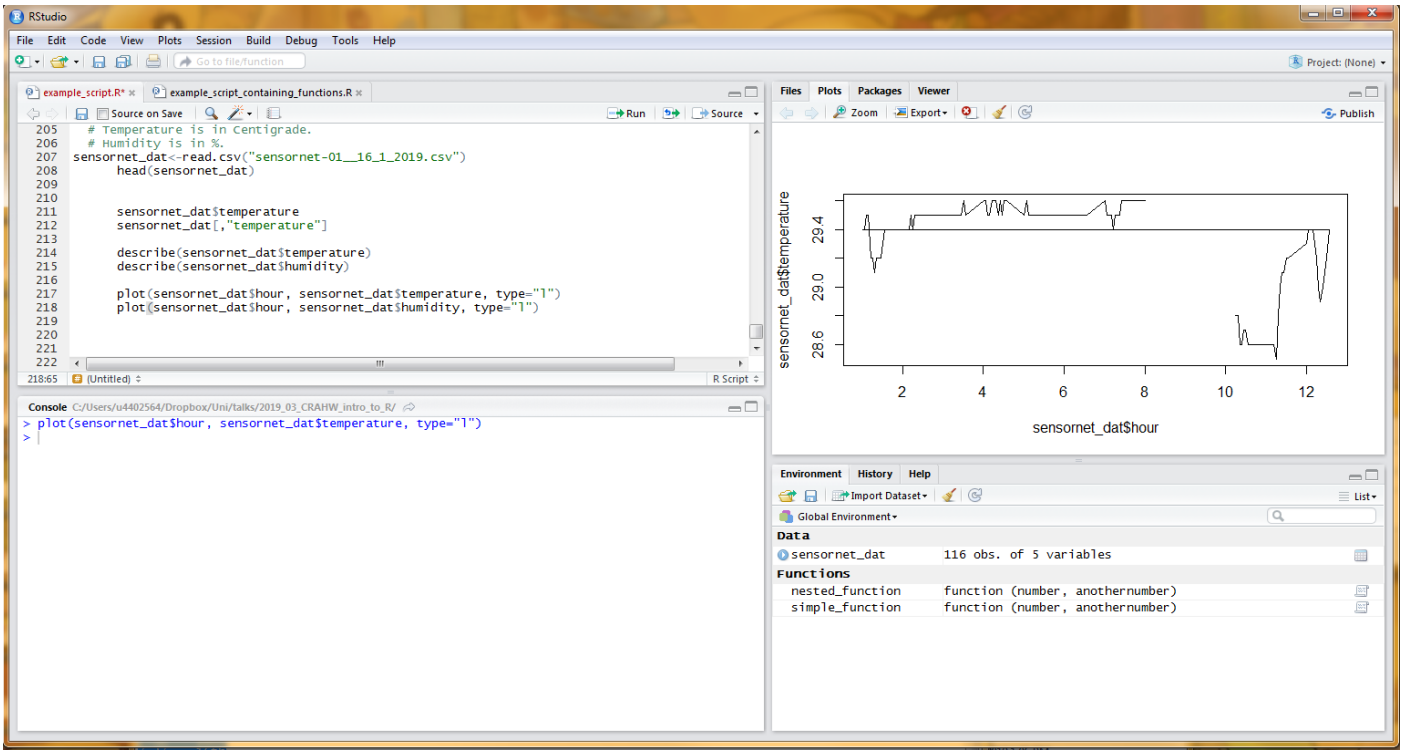
There are two ways to view a variable in a dataset; either using the $ sign, or as a named index. They're both the same, for now we'll use the $ sign because it is easiest.



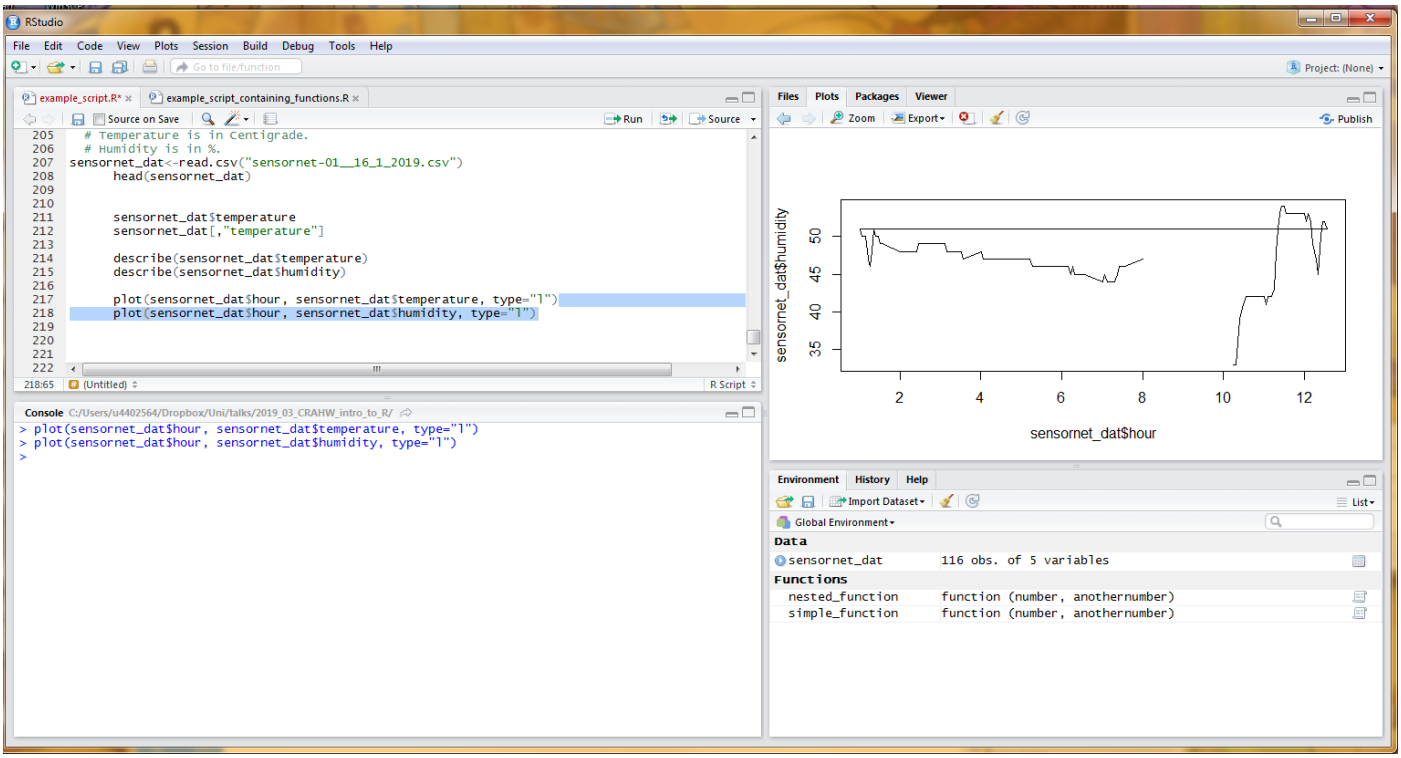
To get a feel for the data, let's start by looking at the descriptive statistics for temperature and humidity using the `describe()` function.



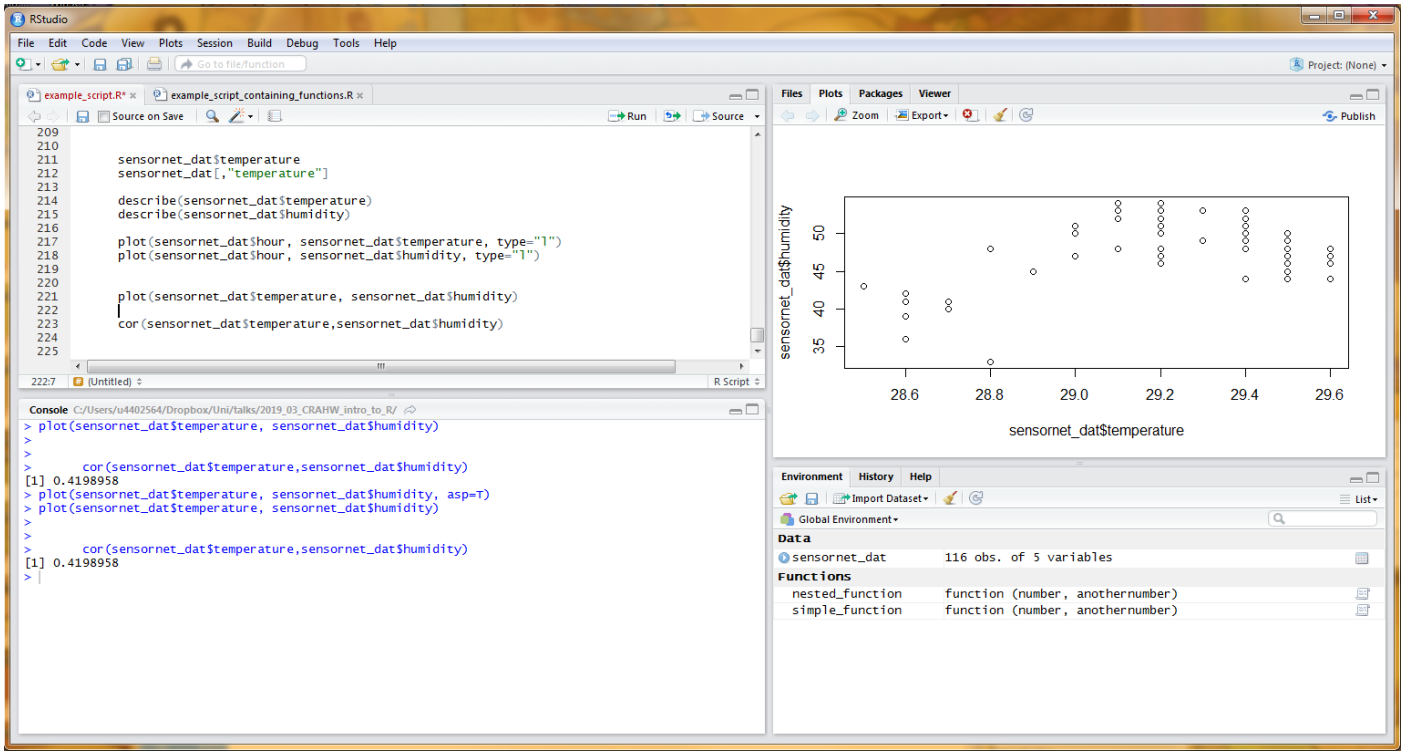
Now, let's plot temperature and humidity by hour. There are many amazing plotting packages in R, such as ggplots, but for now let's just use base graphics. We're using the `plot()` function, with the two variables we want to plot, then the additional argument `type="l"` to get a joined up line. Here is the plot for temperature…



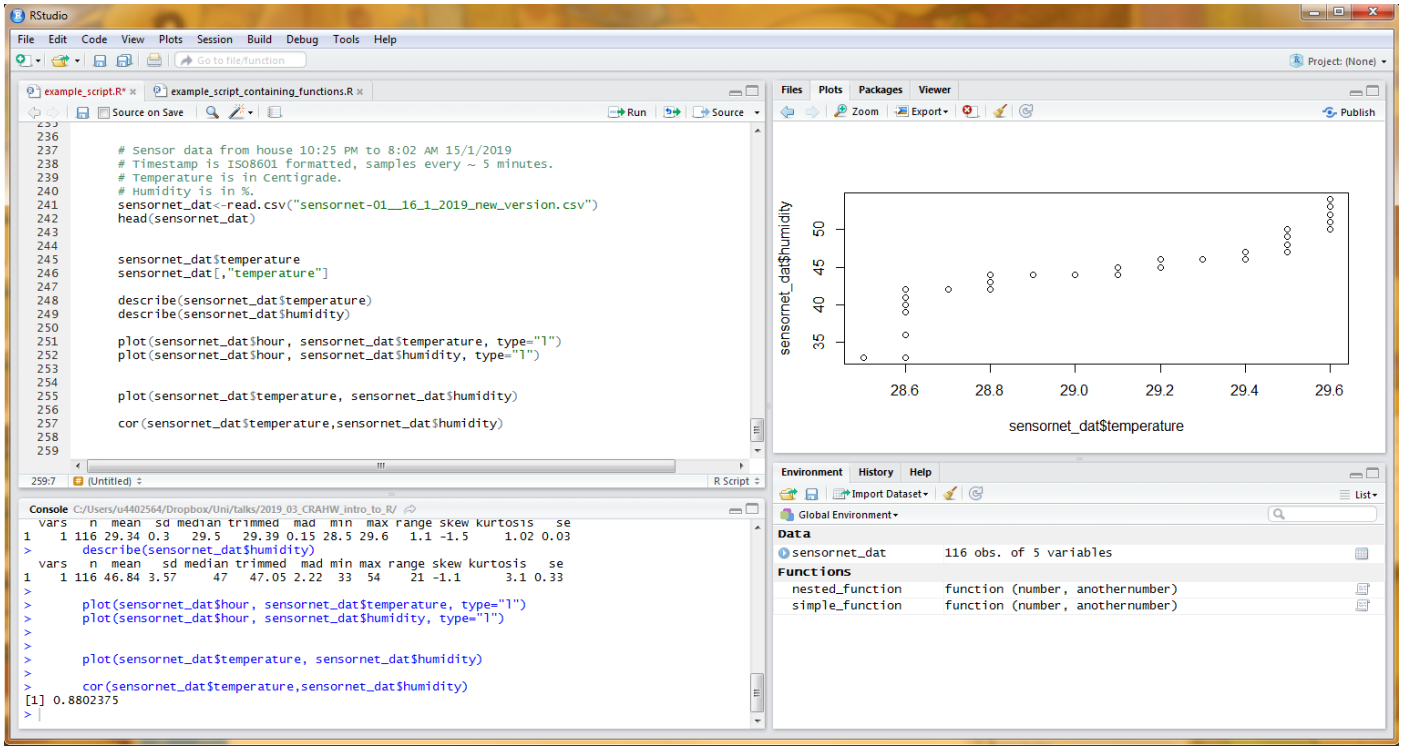
And here is the one for humidity.



Then we can run whatever analysis we want. For example, we might look at the correlation here using `cor()`.

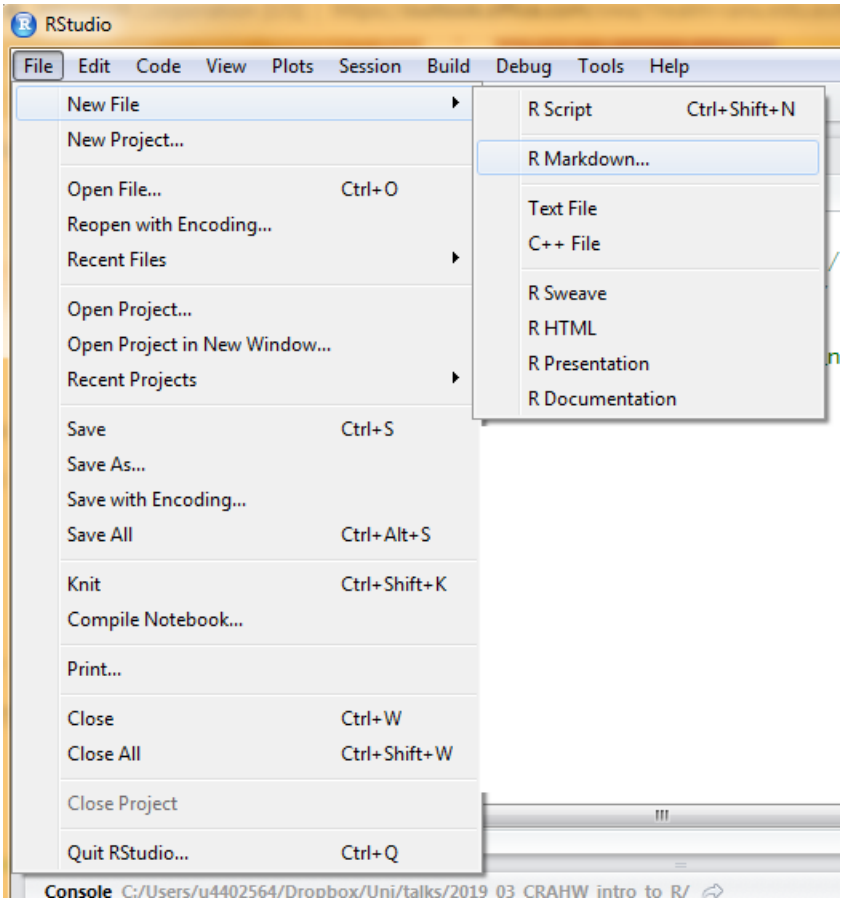


But, say we made a mistake with the sensornet data; turns out there's an updated datafile. Here's where the reproducibility comes in: we can just go back to where we loaded the file, change the name to the new one, then re-run everything else in our script below. Everything is updated, with very minimal effort!

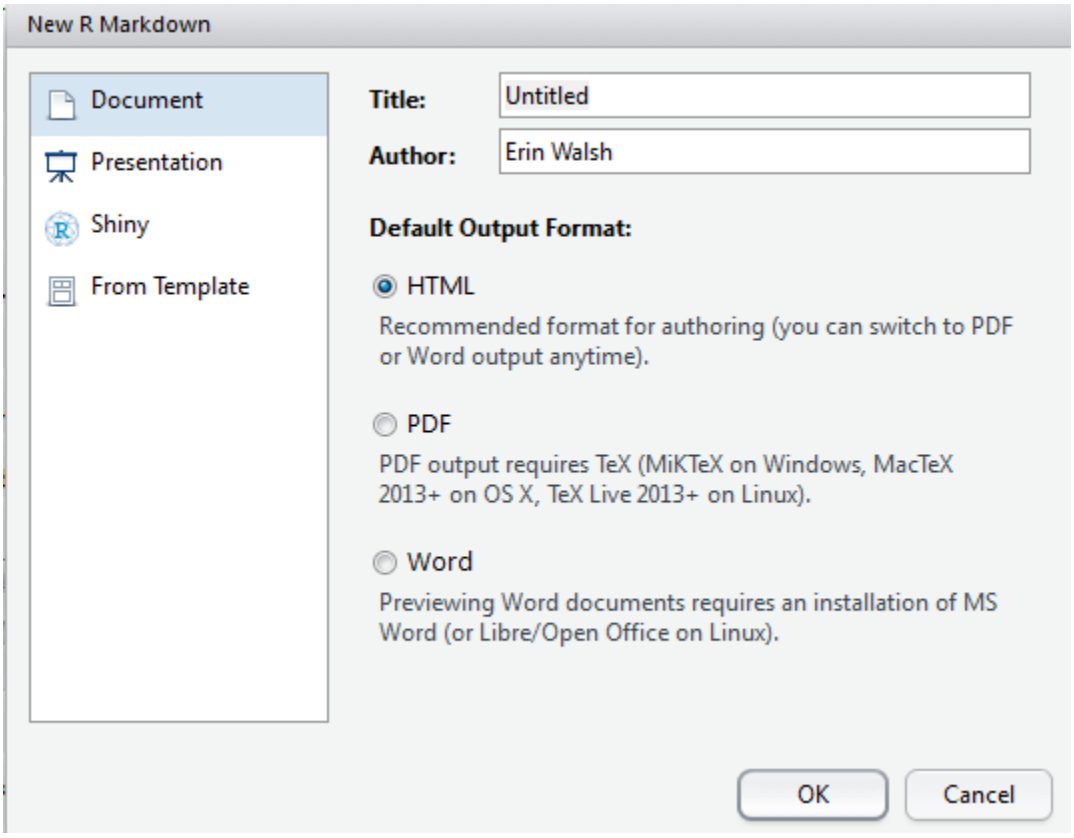


RMarkdown

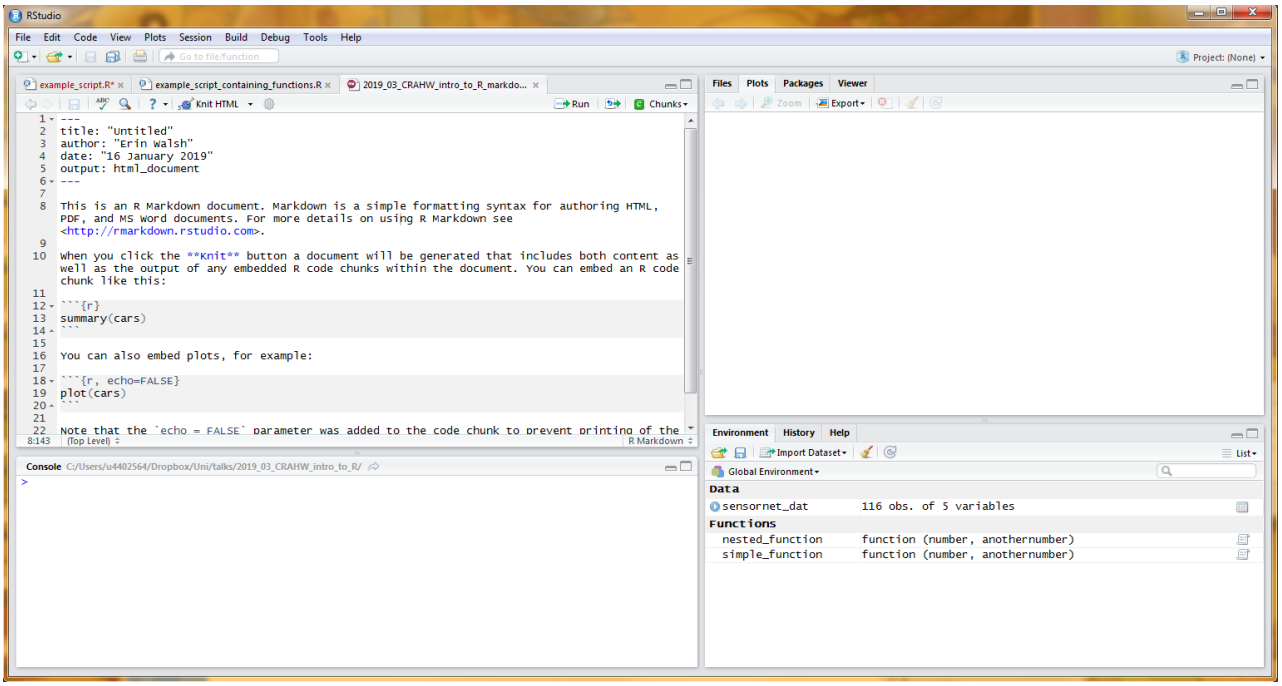
We'll start with a new RMarkdown file.



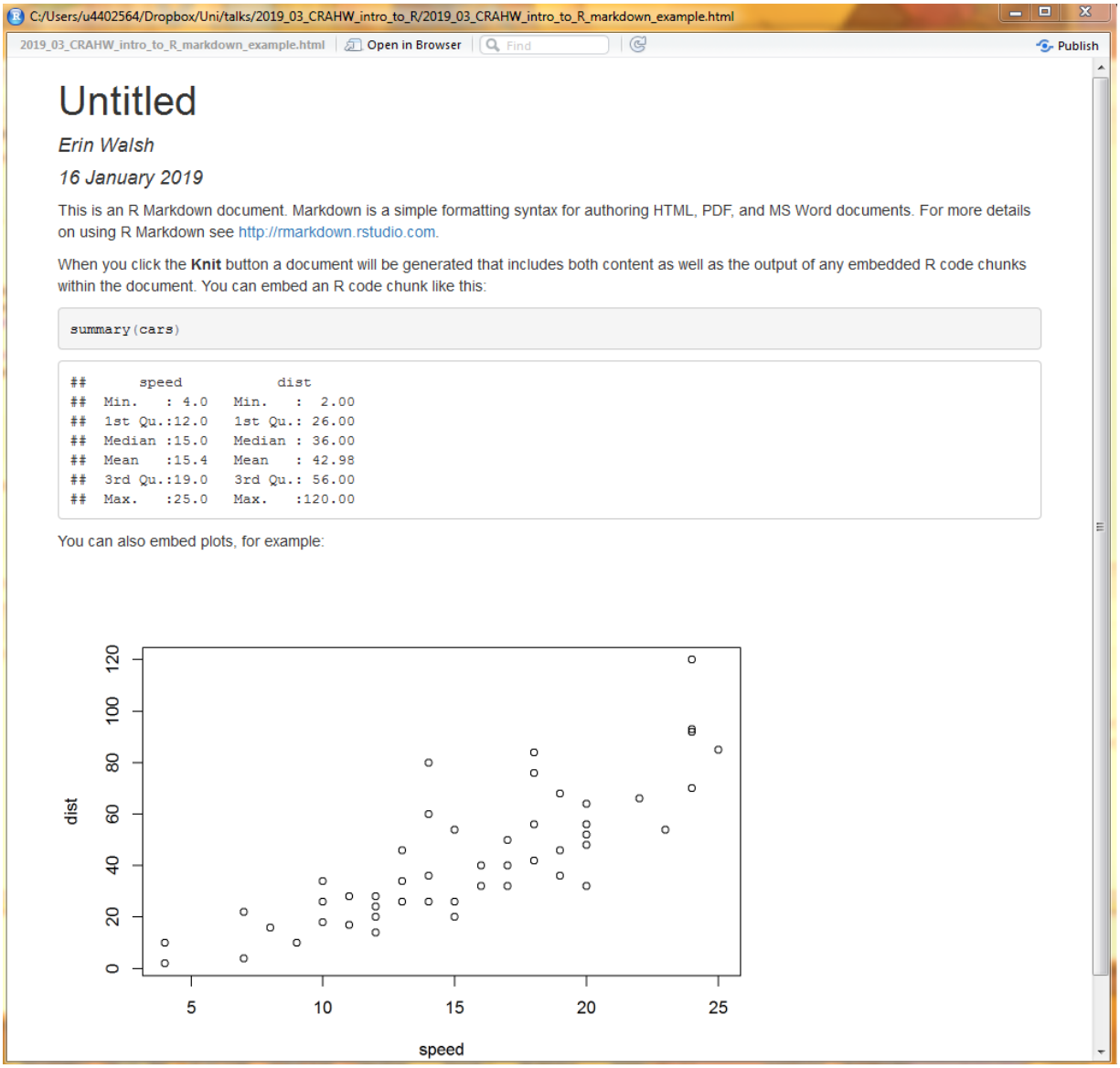
The next step should look something like this – should be fine to leave it on defaults.



They helpfully include information in the default document. Save this document with any name you'd like.

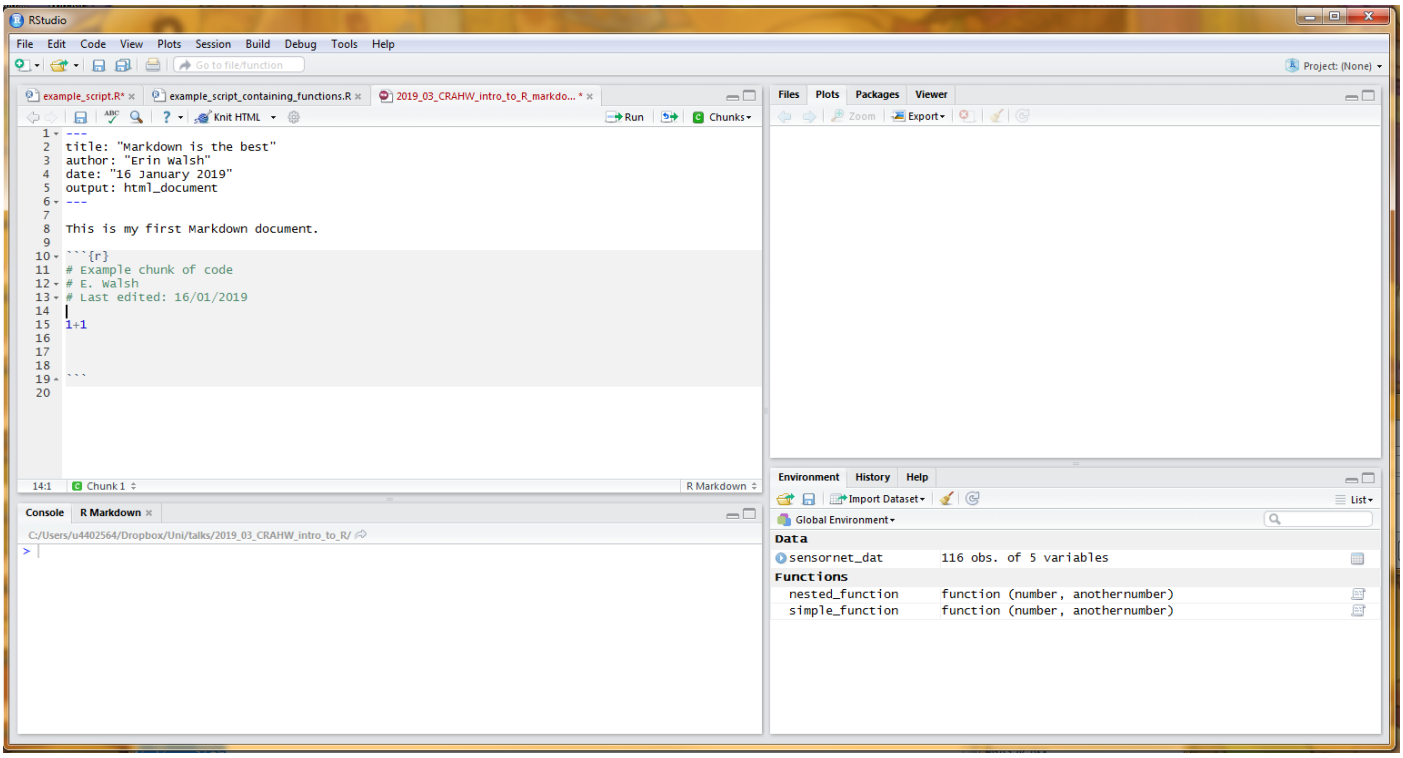


Click 'Knit html', and wait a moment. The console will show some code, and then a new window will pop up. It will look something like this:



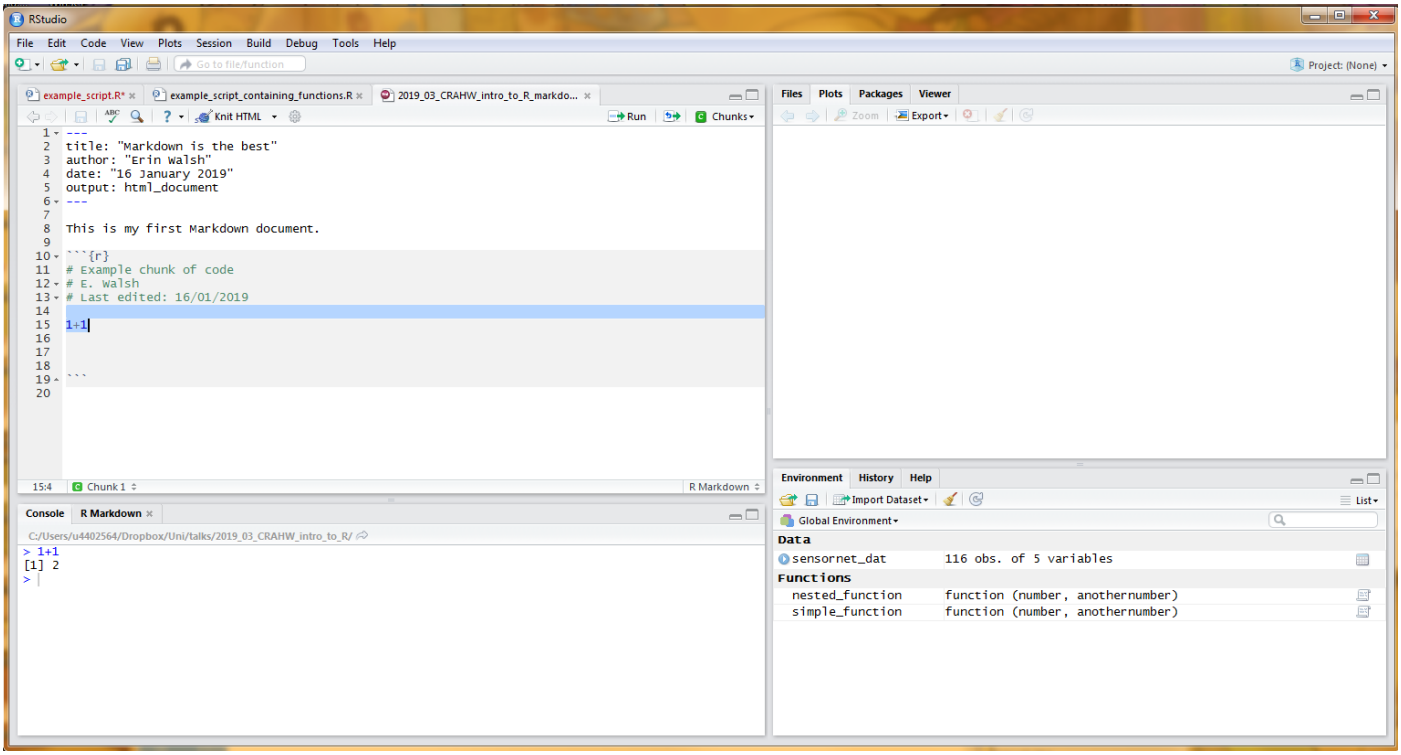
You will also see it as an html file in the same folder as the underlying script.

Markdown has two parts: the plain text area (in white) and the code chunk (in grey). Let's edit the document by removing most of the example text, changing the text at the top, and adding something simple to the code chunk. It should look something like this:

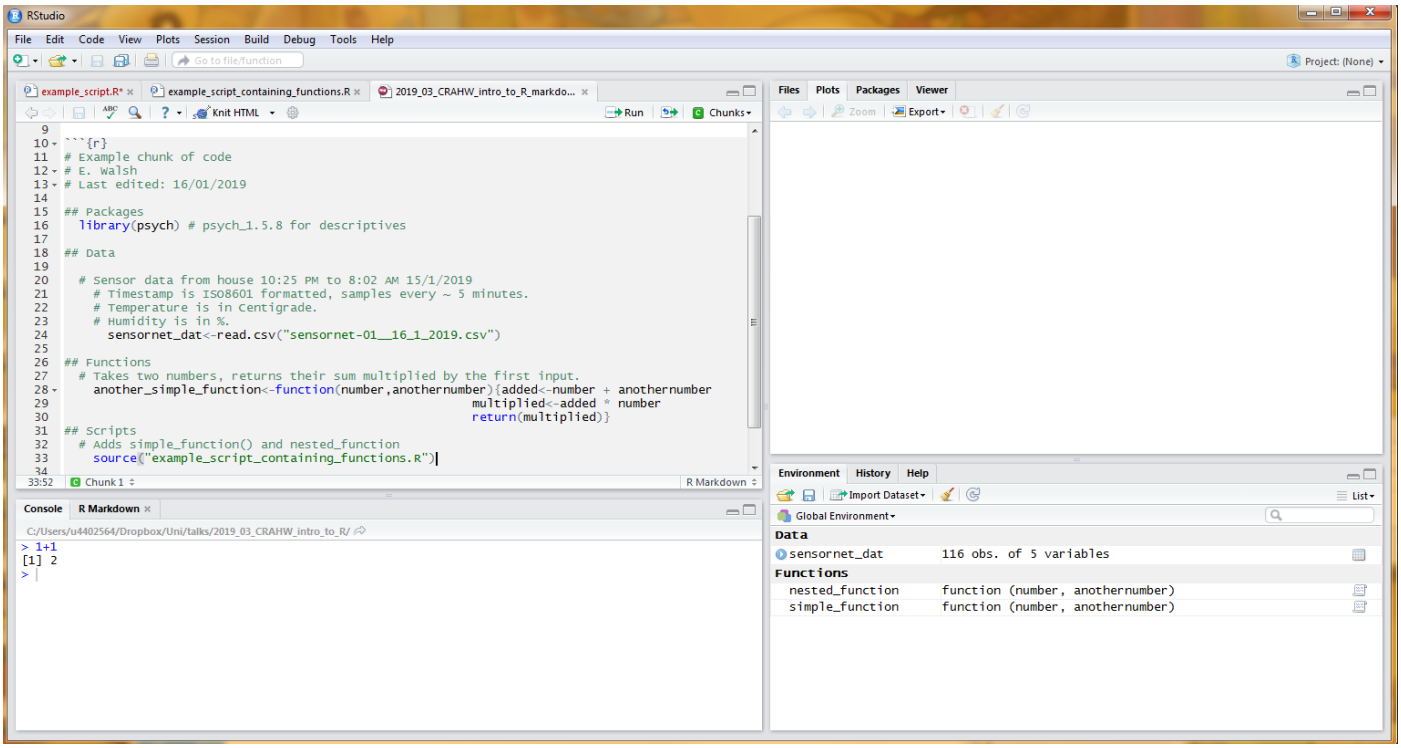


Knit this document and you should end up with something like this:

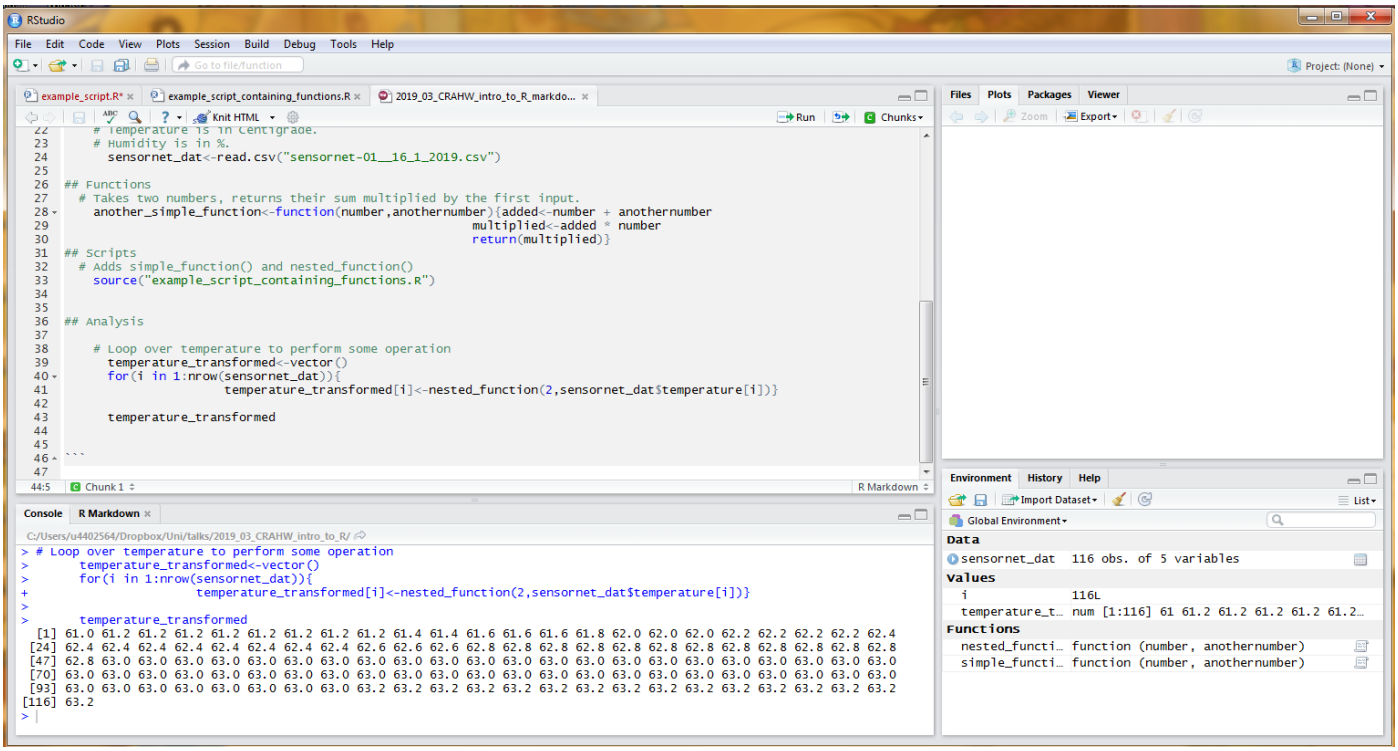
You don't need to compile it every time just to see the results. The grey code chunks work just like the normal script window does, so you can run things as you work (thanks to R's hybrid nature).



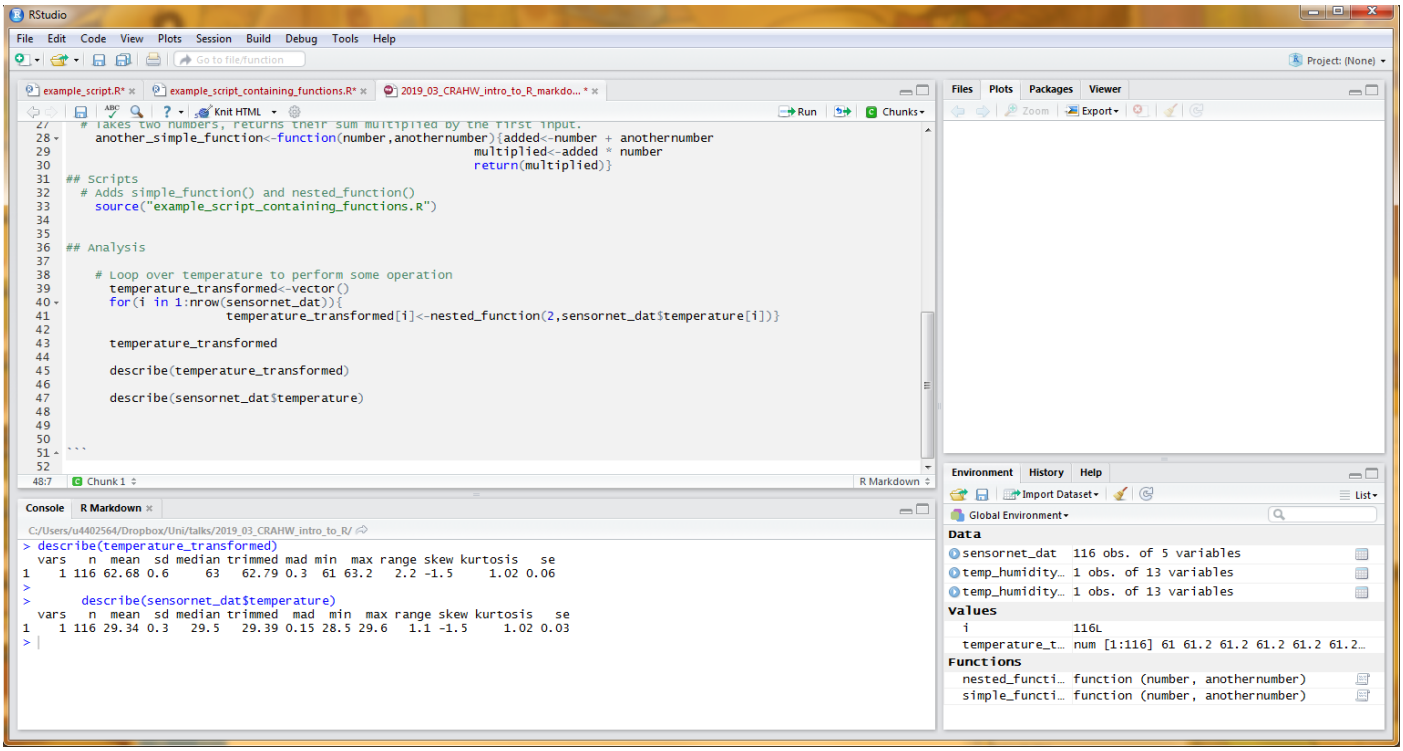
Now for the fun part: we can do anything and everything in the code chunks that we did for the scripts. Even importing scripts! Let's load some packages, data and scripts…



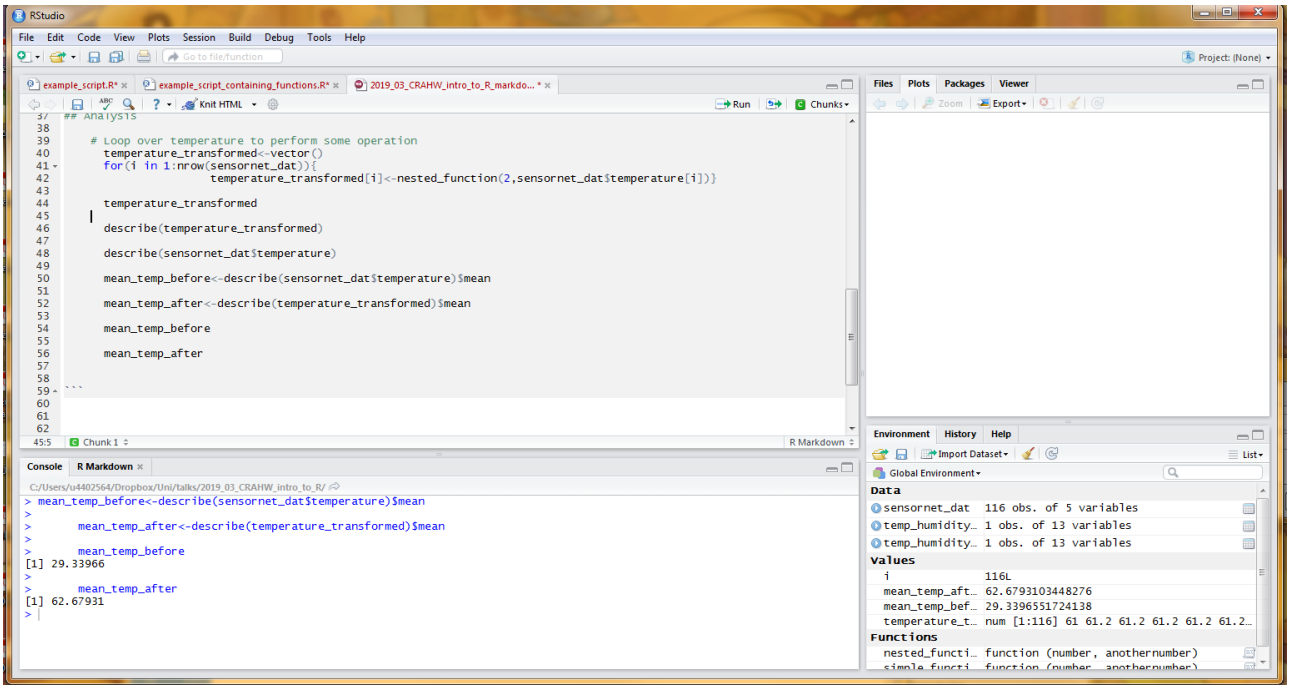
…And run one of our functions from the script we saved over temperature



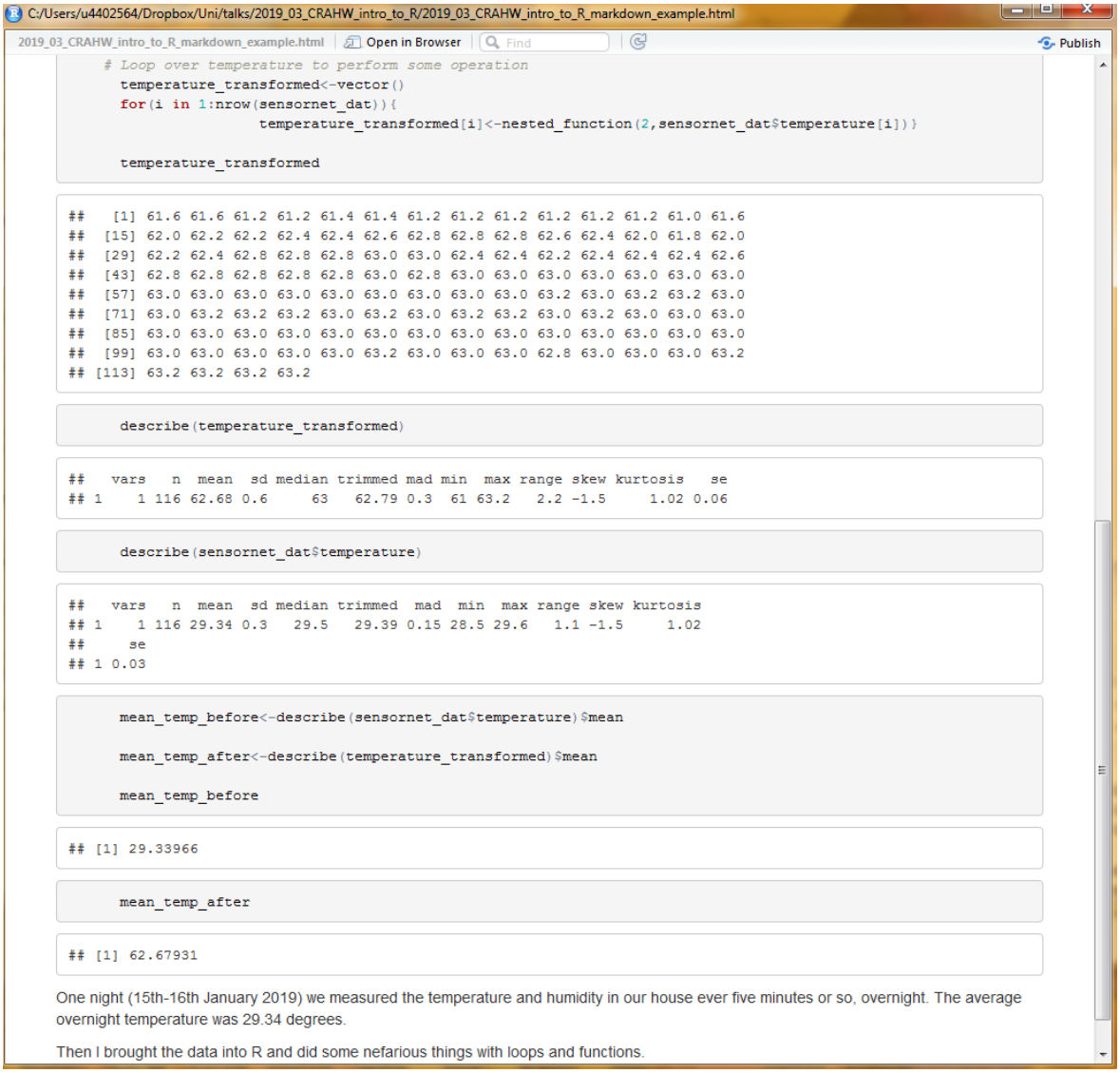
Let's see what happens to the descriptive statistics before and after:



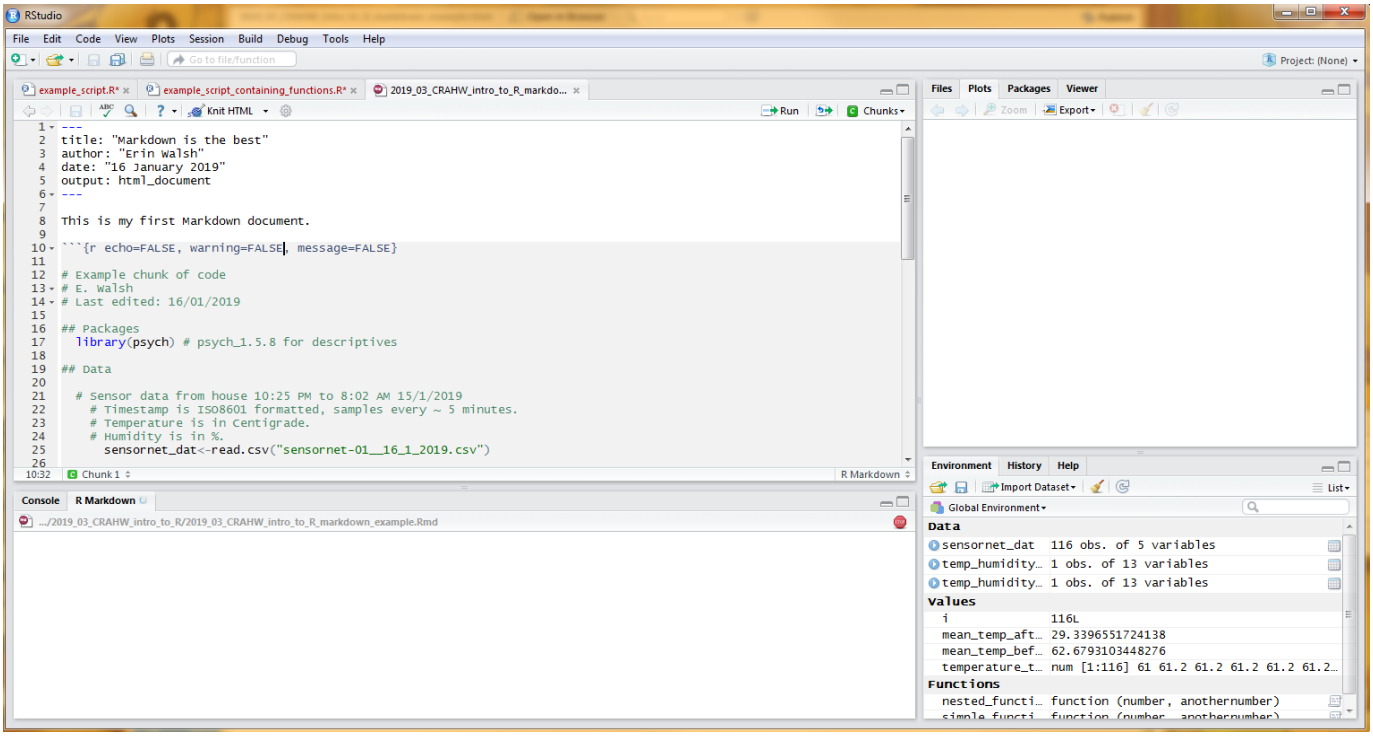
Ok, let's save the mean.



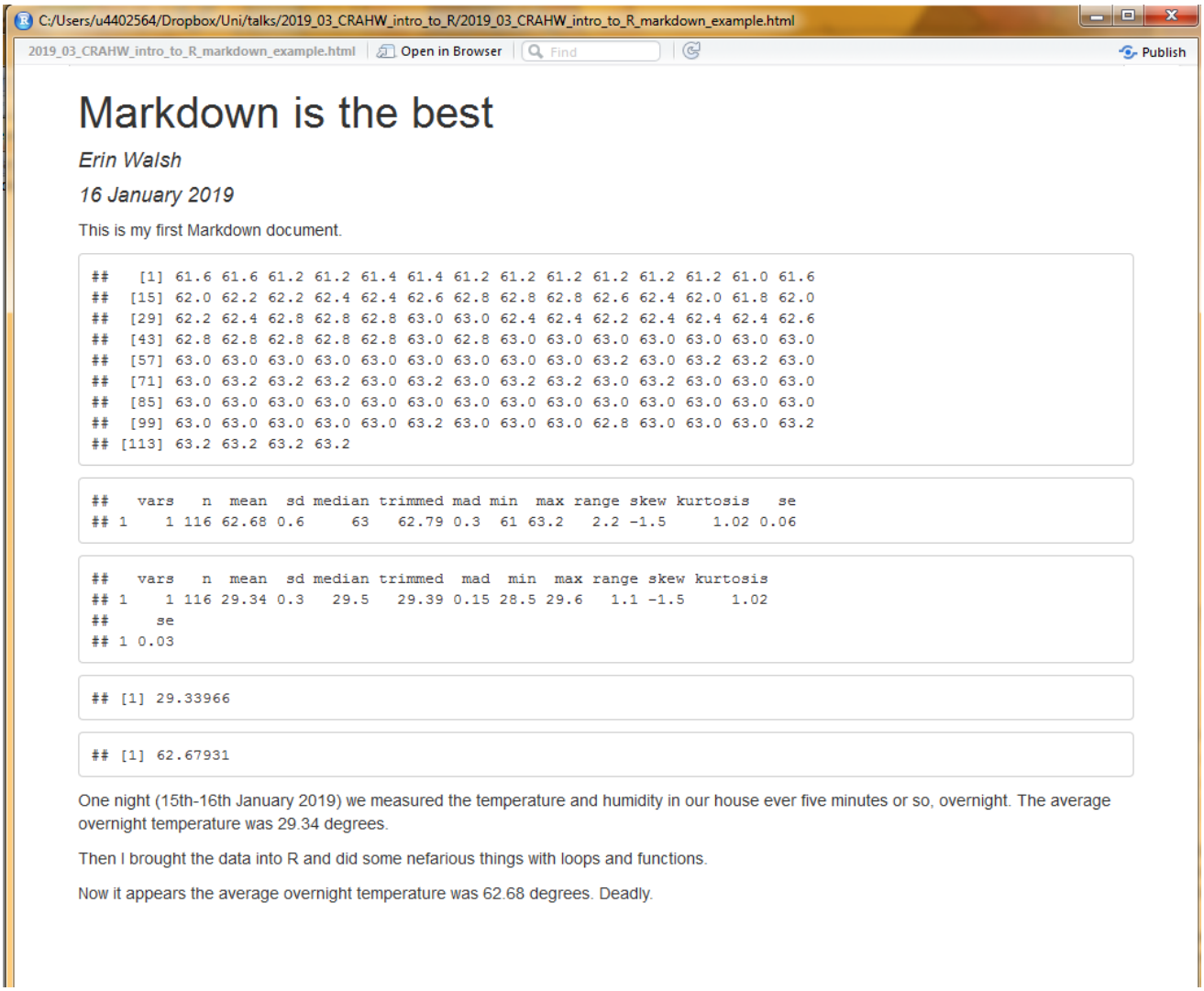
Right now, knitting this is rather underwhelming – you just end up getting a (nicer) depiction of the analysis.



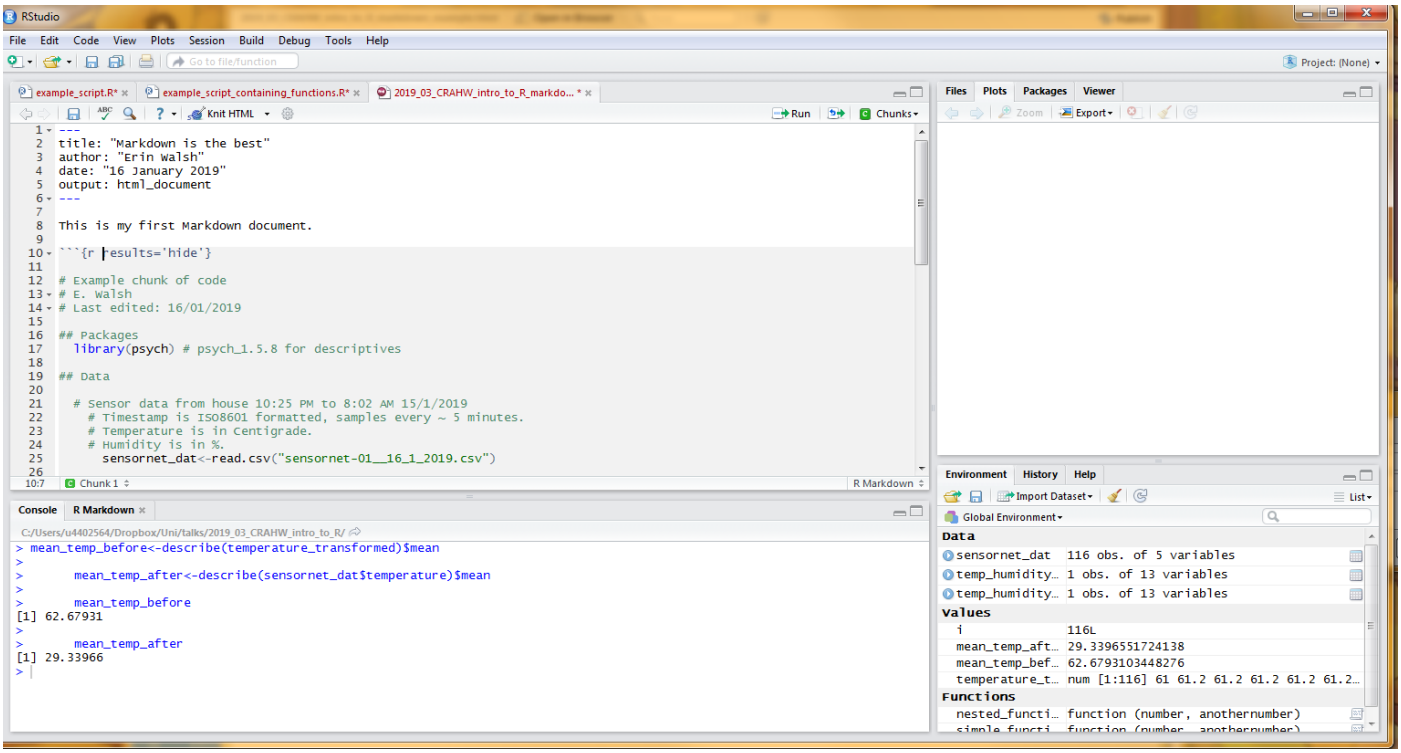
So, let's tidy things up. First, let's hide the underlying code, and just show the output. It will still be run every time markdown is run, but doesn't show up. We do this by adding some arguments to the chunk header.



This gives us... Just the output.

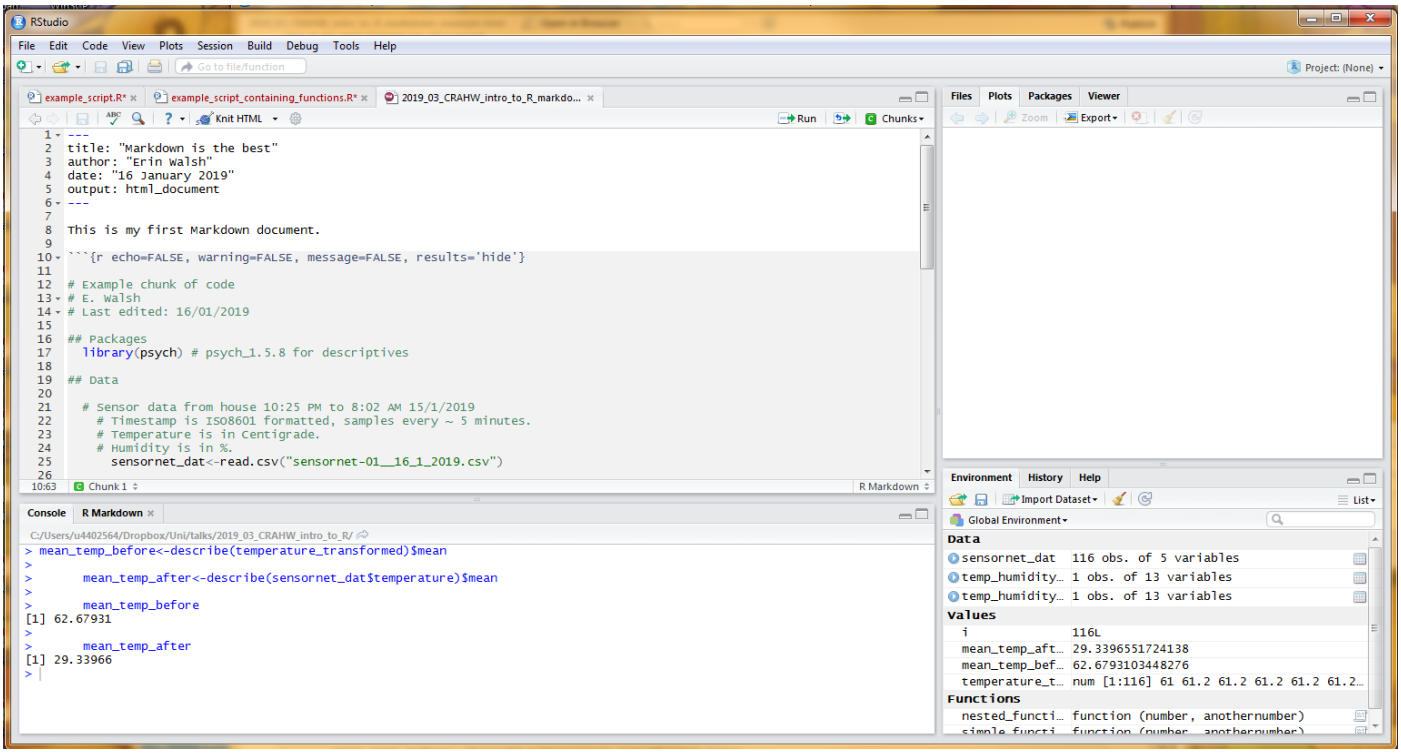


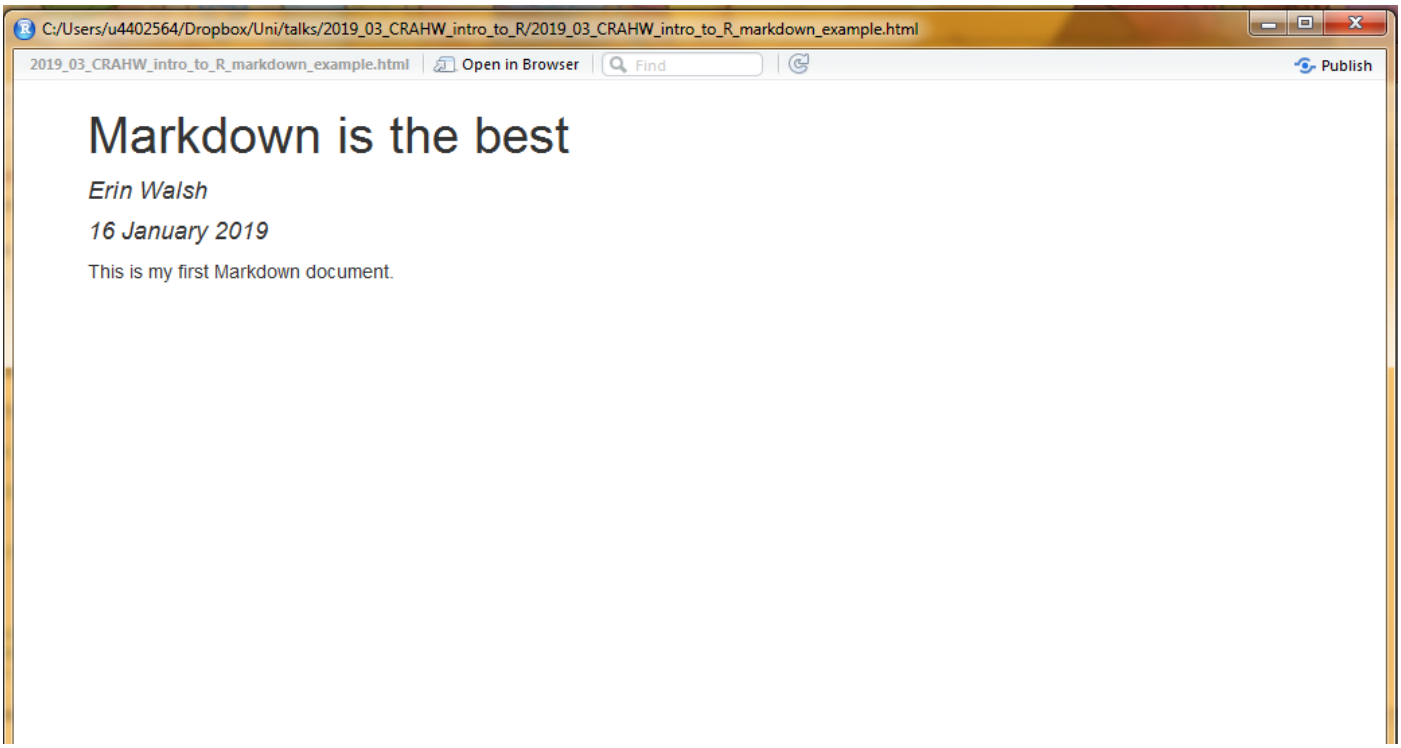
We can also turn off the output to just show the code:



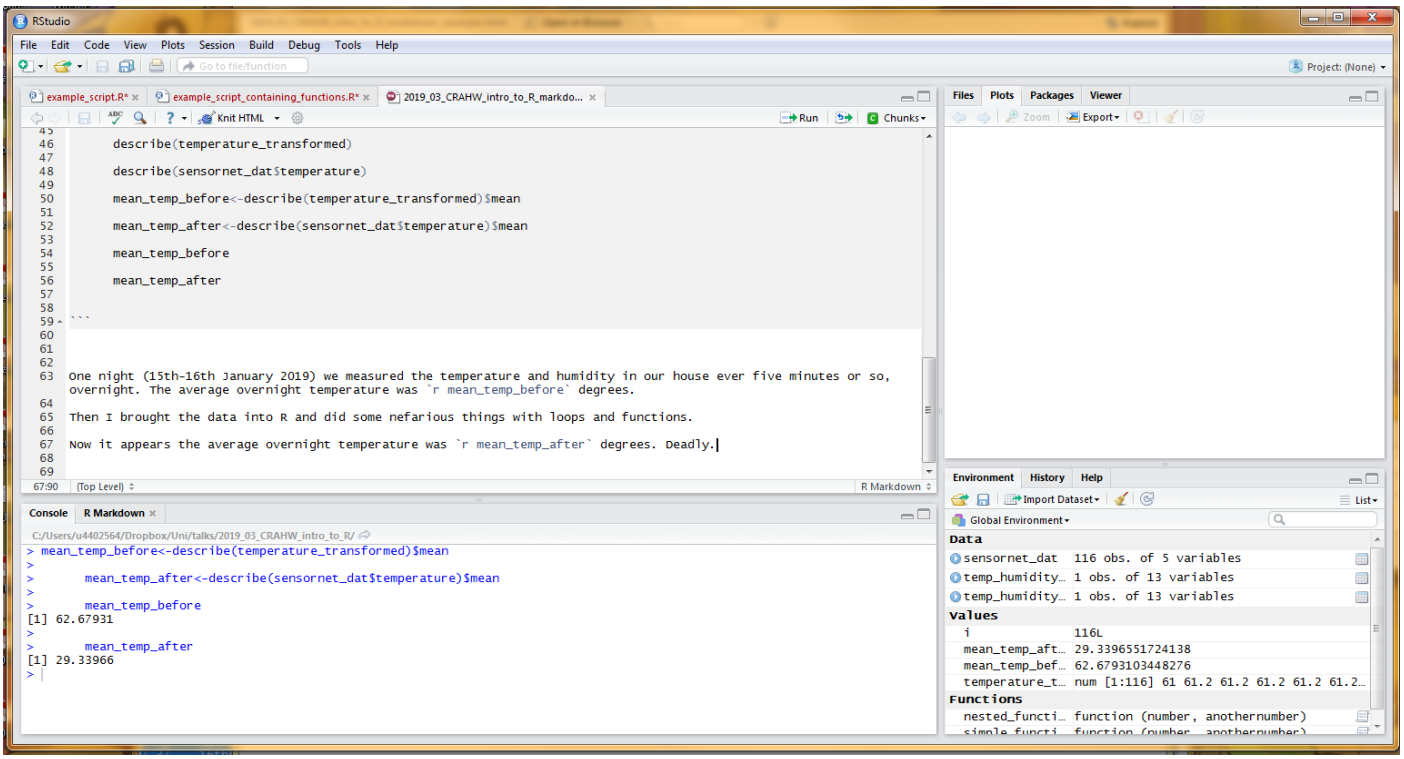

```
    # Humidity is in %.
      sensornet_dat<-read.csv("sensornet-01__16_1_2019.csv")

## Functions
  # Takes two numbers, returns their sum multiplied by the first input.
    another_simple_function<-function(number,anothernumber){added<-number + anothernumber
                                                            multiplied<-added * number
                                                            return(multiplied)}
## Scripts
  # Adds simple_function() and nested_function()
    source("example_script_containing_functions.R")


## Analysis

    # Loop over temperature to perform some operation
      temperature_transformed<-vector()
      for(i in 1:nrow(sensornet_dat)){
                      temperature_transformed[i]<-nested_function(2,sensornet_dat$temperature[i])}

      temperature_transformed

      describe(temperature_transformed)

      describe(sensornet_dat$temperature)

      mean_temp_before<-describe(temperature_transformed)$mean

      mean_temp_after<-describe(sensornet_dat$temperature)$mean

      mean_temp_before

      mean_temp_after
```

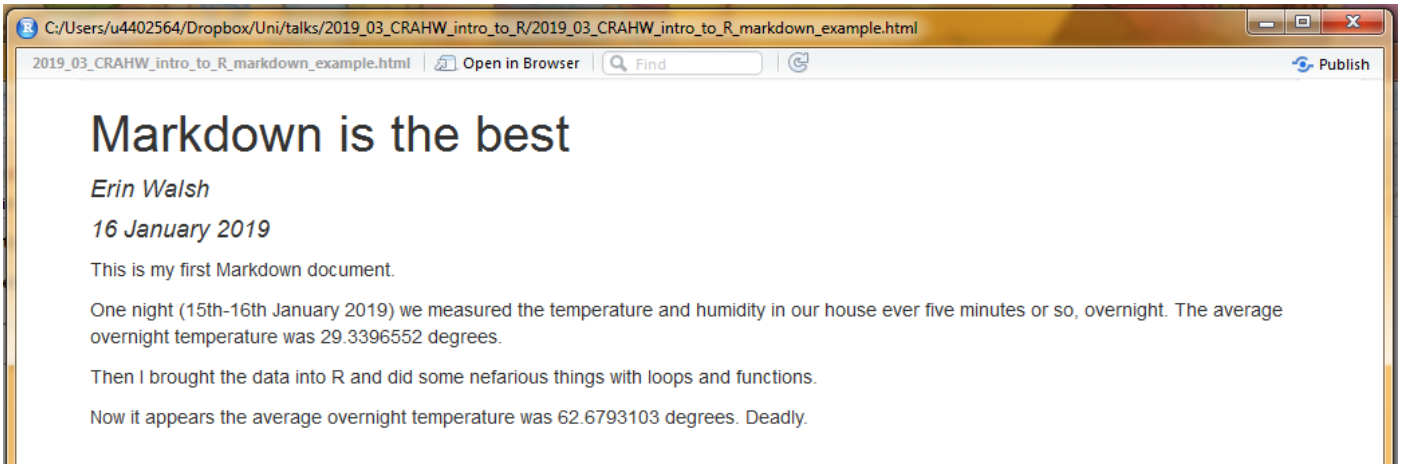
Or hide everything:



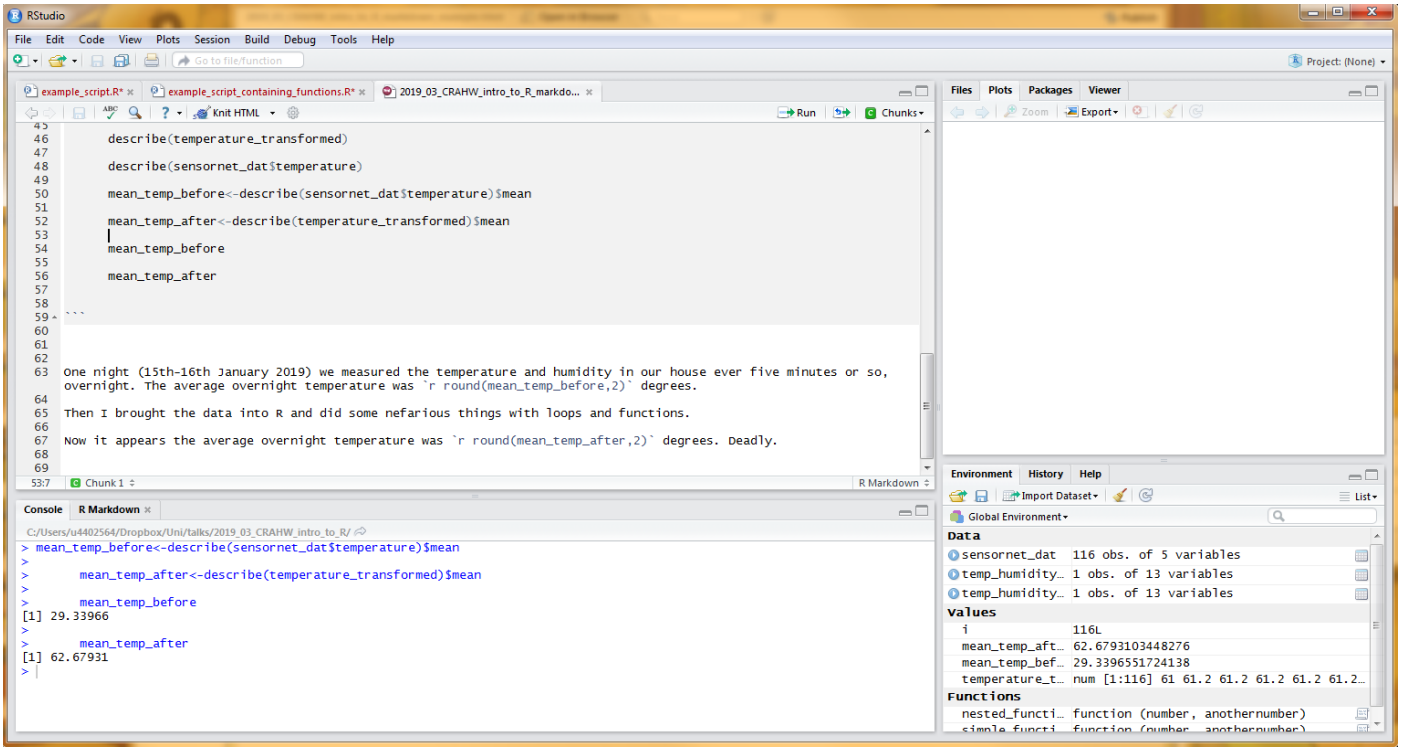


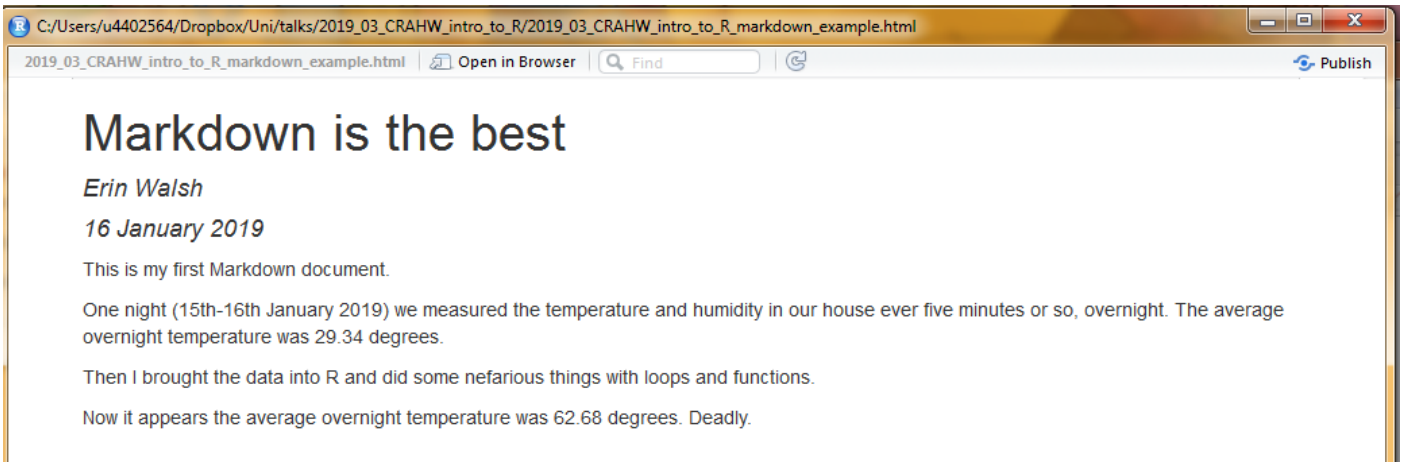
Why would we do this? Well, you can also pipe output directly into text. This essentially means you can write your results section in R, with the results dynamically piped in.





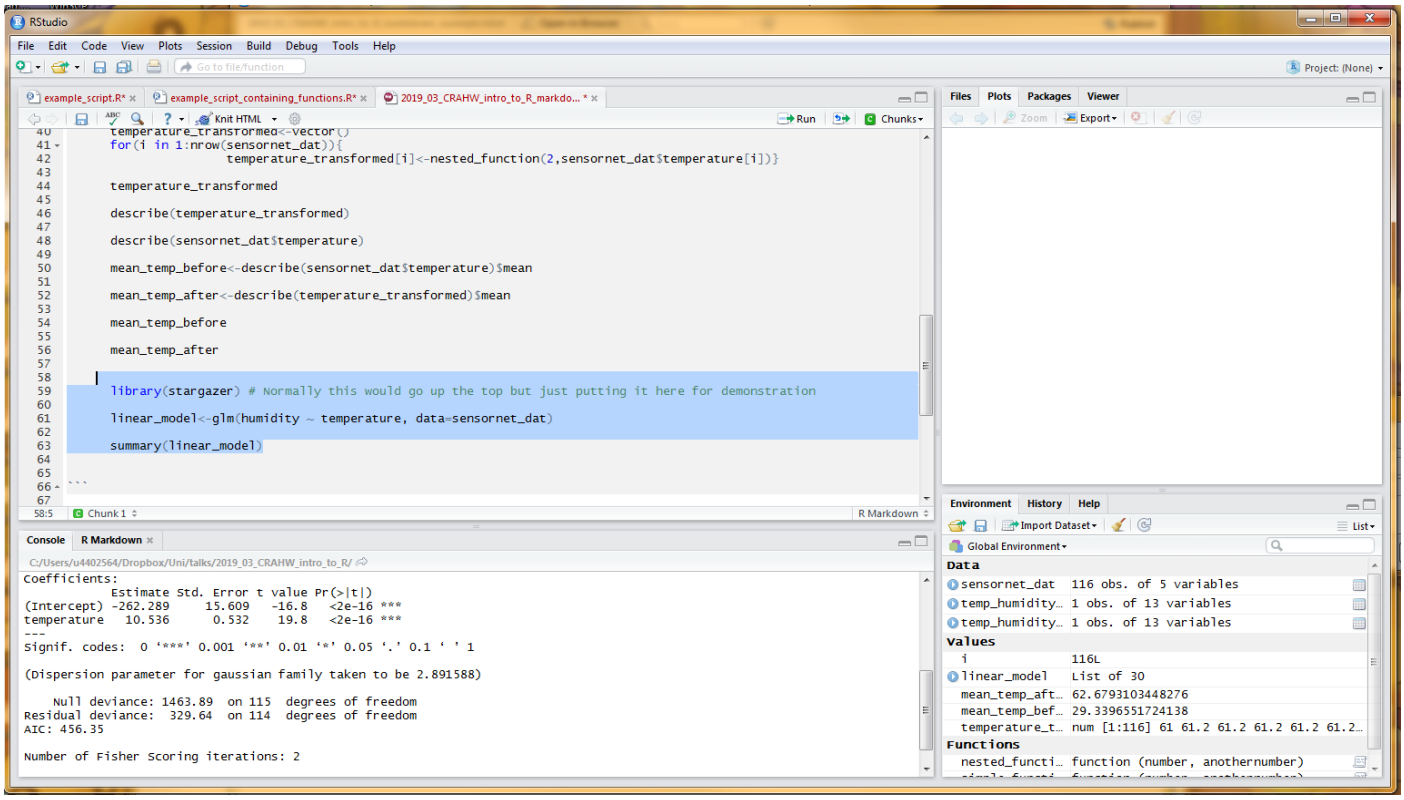
You can also do operations within he in-text r tags, such as fixing the rounding in the above text. In general I'd recommend doing this mostly in the code proper, as you don't want your text getting too messy.



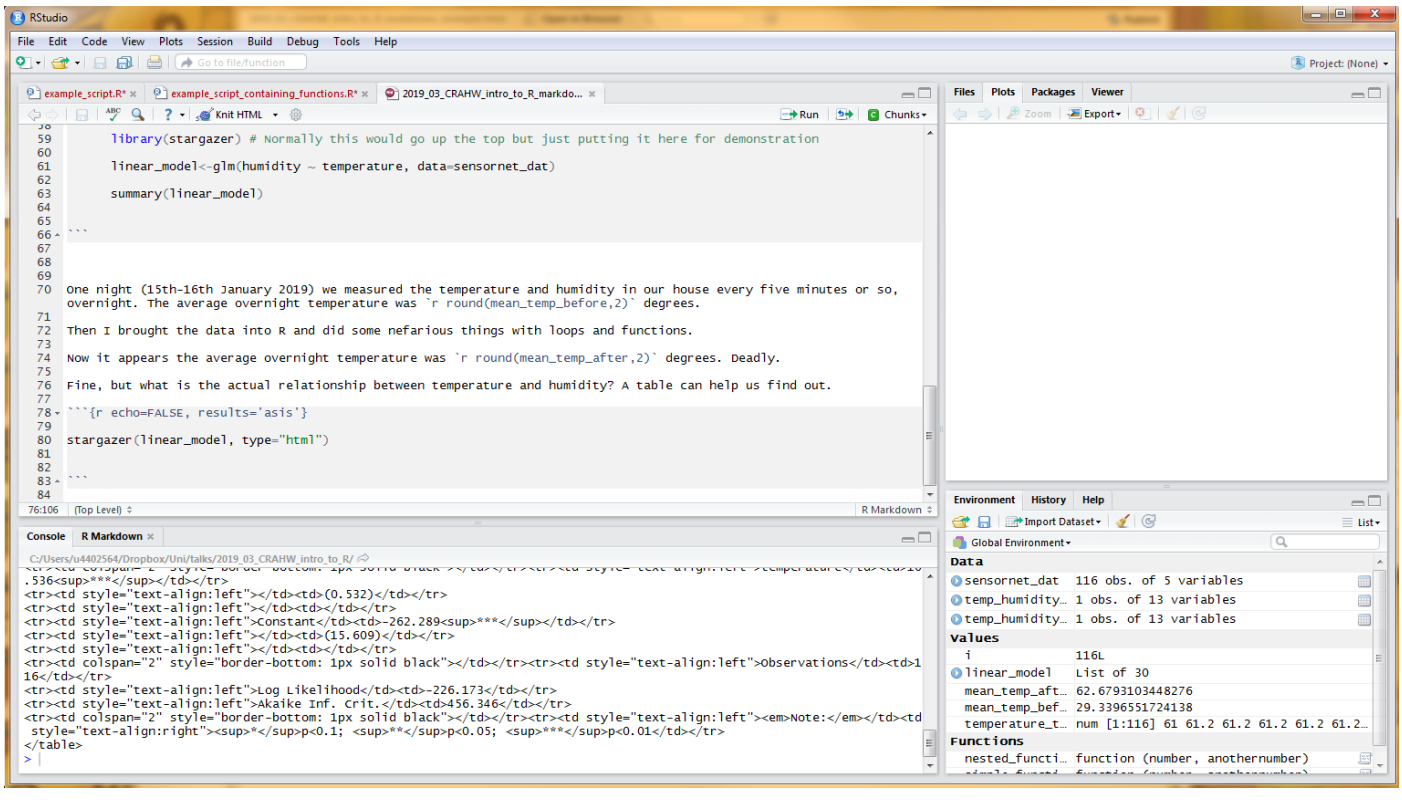


You can add as many code chunks with as many different behaviours and things in them as you would like. Just remember that R runs them from top to bottom, one after another, just like it is a script.

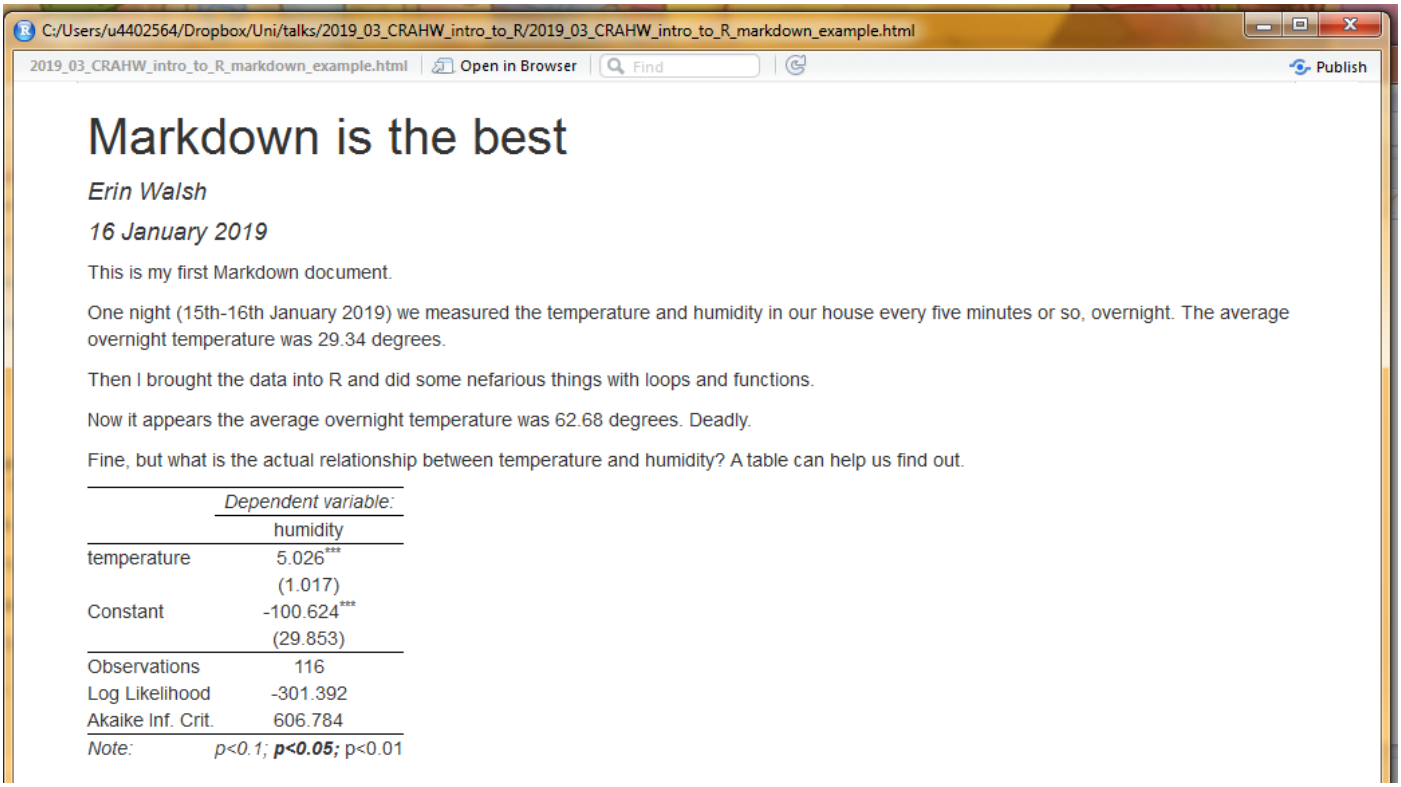
Let's finish the exercise by importing a couple more handy packages, and running a linear regression to describe the association between temperature and humidity. Stargazer is a great package for neatly summarising most linear models in a table.



We've made a new chunk below, and told it to render `'asis'` (as well as not showing constituent code). This is because Stargazer produces really lovely html tables; you can see we have asked stargazer to report our linear model with html output. It looks unreadable in the console window, but when we knit…



We get a beautiful document.



The formatting here can be customised substantially (lots of good resources online; this is a great start: https://www.markdowntutorial.com/).