Debugging with RStudio

Leo Sun

December 3, 2017

Introduction: Debugging

If you ever programmed before, you probably had a situation where some line(s) were causing errors in your program. Sometimes you can spot the bug easily, but other times you have no idea what is happening. This is where debugging becomes handy.

All programming languages have some method for debugging code. In Python, you can debug with a library called pdb (python debugger). In R, you can use the built-in function `debug()`.

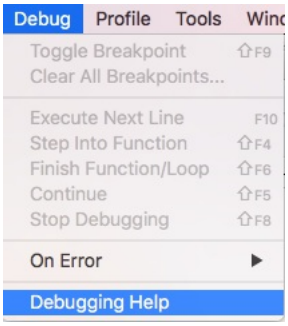
Debugging functions can be used in the console, but it is very hard to visualize what's happening. You also need to memorize the debugging functions and shortcuts. Luckily, we have an IDE (integrated development environment) to help us debug code. IDEs are extremely helpful in debugging because they provide visual cues that help users debug their program.

As a side note, you can only debug scripts (.R) and not .Rmd or .md files.

Getting Started

This is a brief overview of each option in the debug menu. I will explain them in depth later on.

If you click on the debug menu, you will see a list of options, but what do they mean?



1. A breakpoint is a pause that you can place in your code that will cause the debugger to stop on that breakpoint. You can place a breakpoint by going to the line that you want to debug and pressing Toggle Breakpoint. You can also place a breakpoint by left clicking the blank space left of the line number. It will appear as a red dot.

```
1  add = function(x, y) {
2    sum = x + y
3    return(sum)
4  }
```

2. Clear all breakpoints will clear all of the breakpoints in your script.
3. Execute next line will run the line that the debugger is stopped on.
4. Step into function allows you to go "inside" of a function.
5. Continue will execute the rest of the program, and can stop on a future breakpoint if one exists.
6. Stop debugging will close the debugger.
7. On error allows you to select what happens when the debugger executes and receives and error.
8. Debugging help will open a link in your browser that brings you RStudio's debugging website.

Simple Example

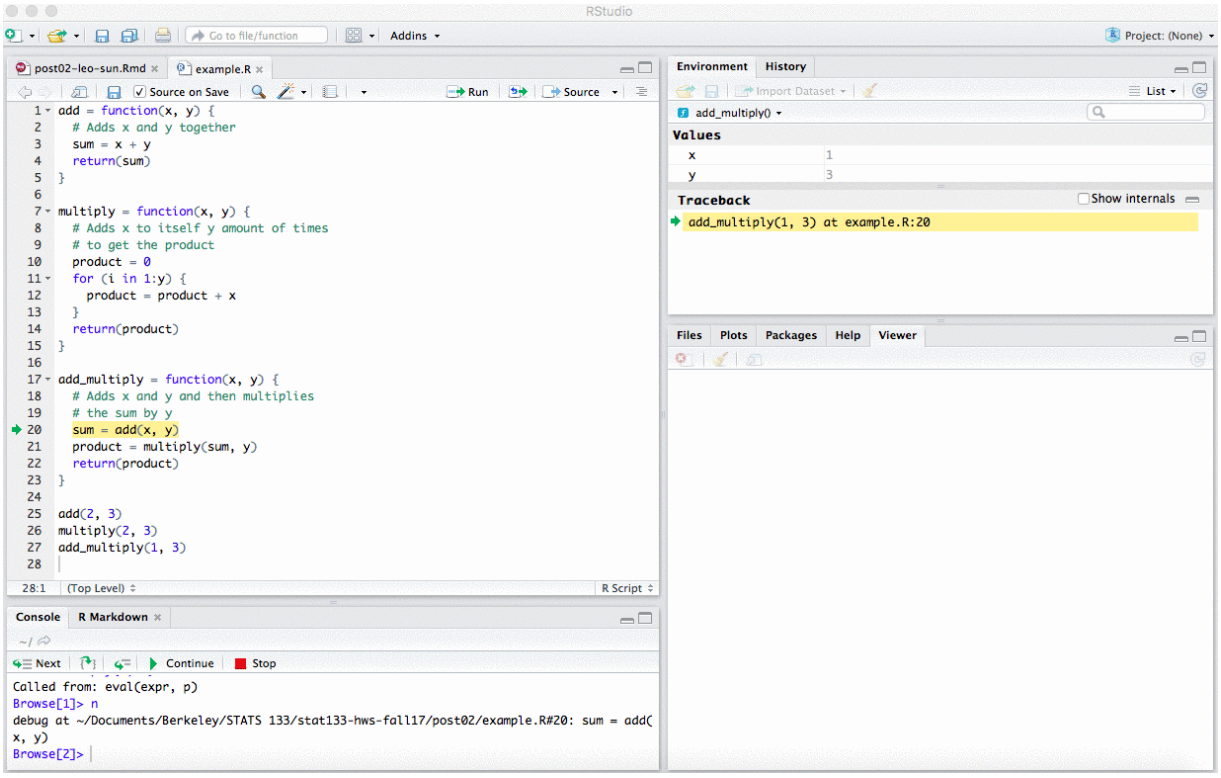
Lets try to debug this function.

```r
add = function(x, y) {
  # Adds x and y together
  sum = x + y
  return(sum)
}

multiply = function(x, y) {
  # Adds x to itself y amount of times
  # to get the product
  product = 0
  for (i in 1:y) {
    product = product + x
  }
  return(product)
}

add_multiply = function(x, y) {
  # Adds x and y and then multiplies
  # the sum by y
  sum = add(x, y)
  product = multiply(sum, y)
  return(product)
}

add(2, 3)
multiply(2, 3)
add_multiply(1, 3)
```

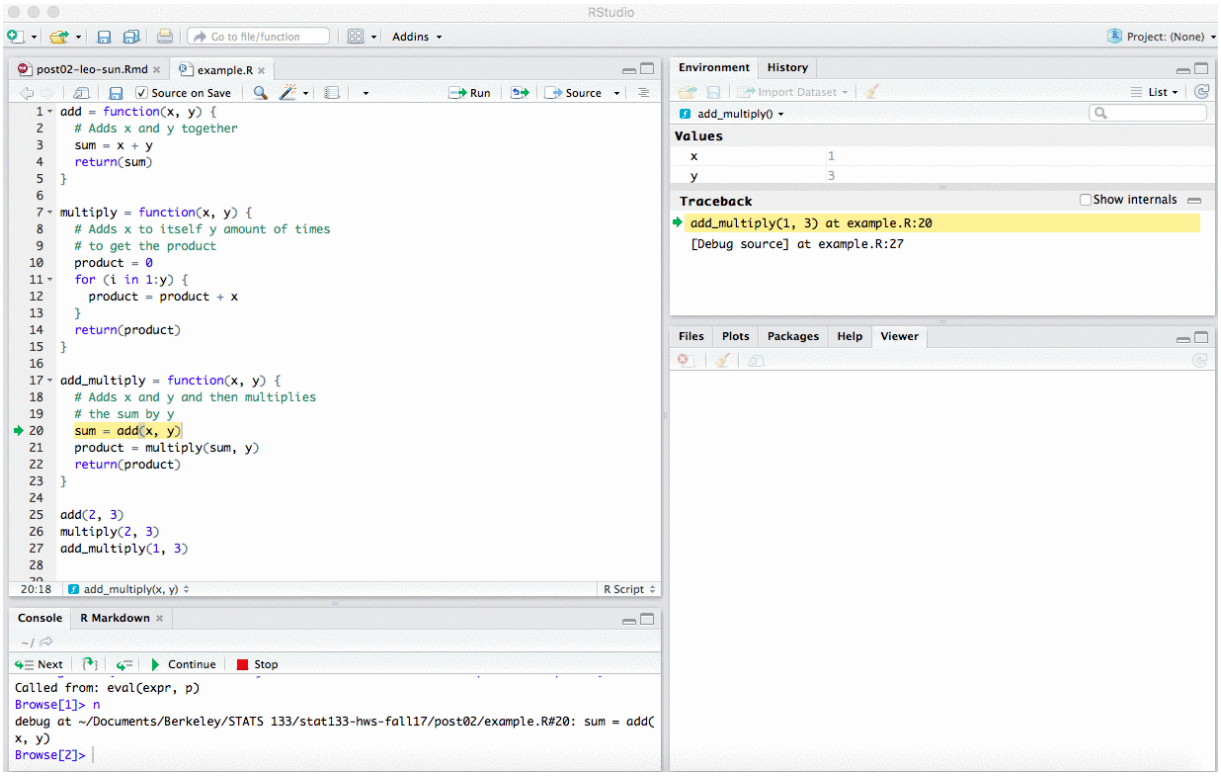
Add a breakpoint on line 80 and then run the script. The debugger should stop on line 20. The green arrow is the current line the debugger is on.

On the console, you can see that the options in the debug menu are now shown.

Clicking next (Execute Next Line in debug menu) will execute the current line of code. Clicking `Next` once will execute line 20. Clicking it again executes line 21.

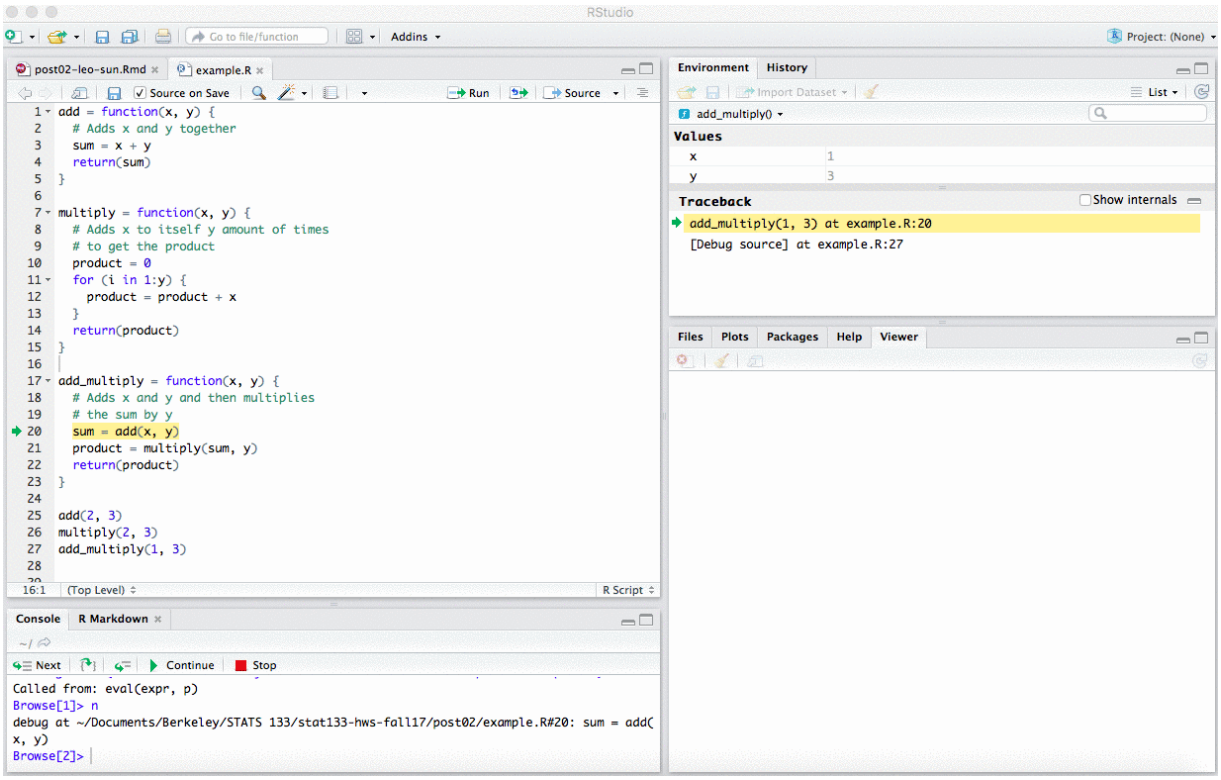


But what if you want to see what is happening in the add and multiply function? The 2nd button the console (Step Into Function in debug menu) allows us to view the internals of add and multiply.

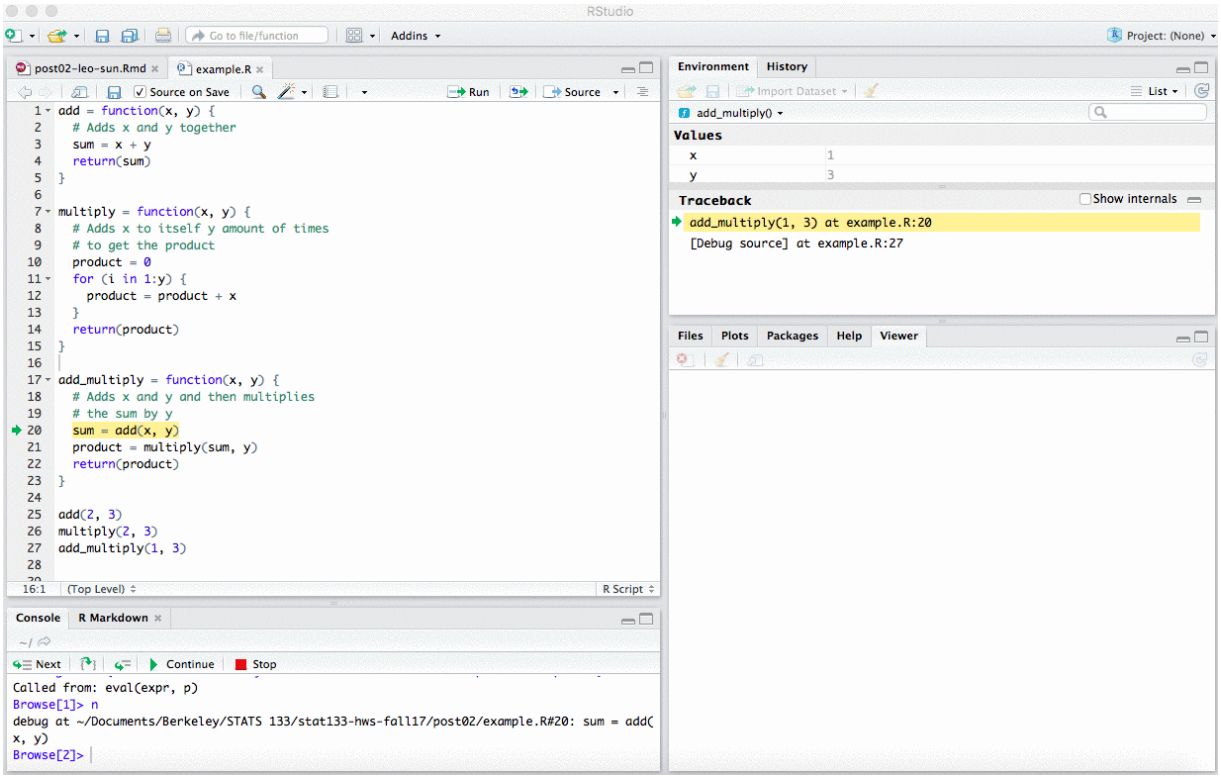


As you can see, stepping into add brought the debugger to the add function. We can now debug the add function to see what it does.

In multiply, there is a loop, so you will have to press next a lot to execute the rest of the function. The 3rd button (Finish Function/Loop in debug menu) allows to us to execute the whole loop.

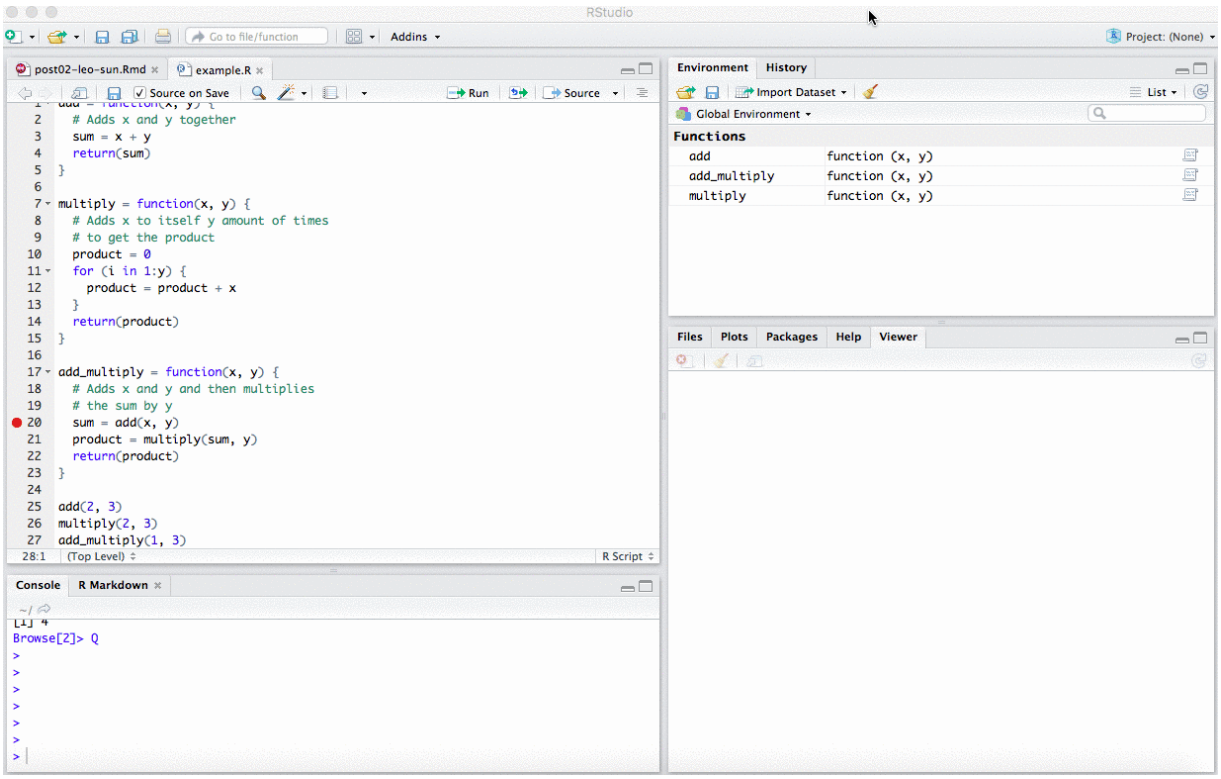


The 4th button (Continue in debug menu) will execute the rest of your script or stop at the next breakpoint.



Pressing continue executes the rest of the script. After adding another breakpoint on line 22, pressing continue executed the script until it hit the other breakpoint.

Another useful tool for debugging is the console itself. You can call functions and variables on the console when you are in the debugger.



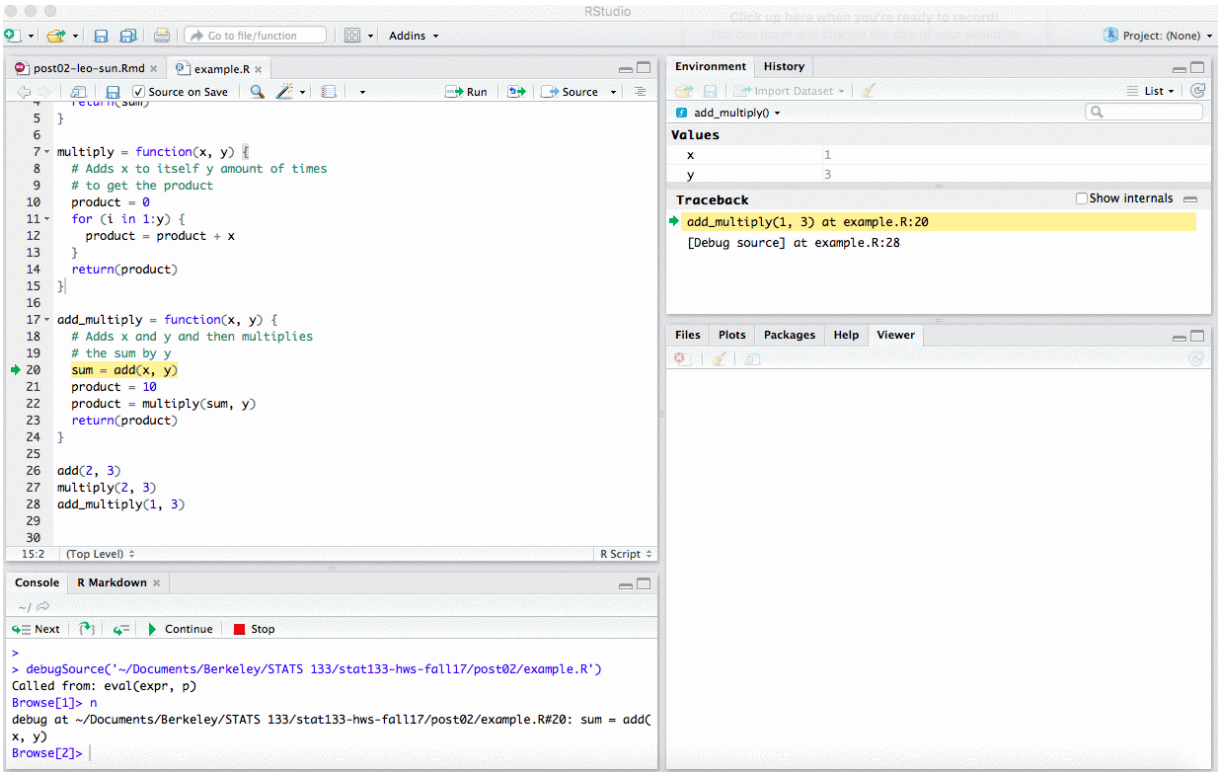
You can see that calling the variables will print out the variable and using them with functions will produce an output.

When calling variables, keep in mind the concept of scope. Wikipedia states that:

> In computer programming, the scope of a name binding – an association of a name to an entity, such as a variable – is the region of a computer program where the binding is valid: where the name can be used to refer to the entity. Such a region is referred to as a scope block. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).

Wikipedia's definition maybe sound confusing, here is a more concrete example. Modify the script so that `add_multiply` looks like this:

```r
add_multiply = function(x, y) {
  # Adds x and y and then multiplies
  # the sum by y
  sum = add(x, y)
  product = 10
  product = multiply(sum, y)
  return(product)
}
```



When I first called product, our current scope is in the `add_multiply` function. So, it prints out 10. After I stepped into the `multiply` function, the scope is now within the `multiply` function, so product is now 0. Once I finish executing the function, the scope is now back to `add_multiply`. Product now outputs 12.

Simple Example with Bug

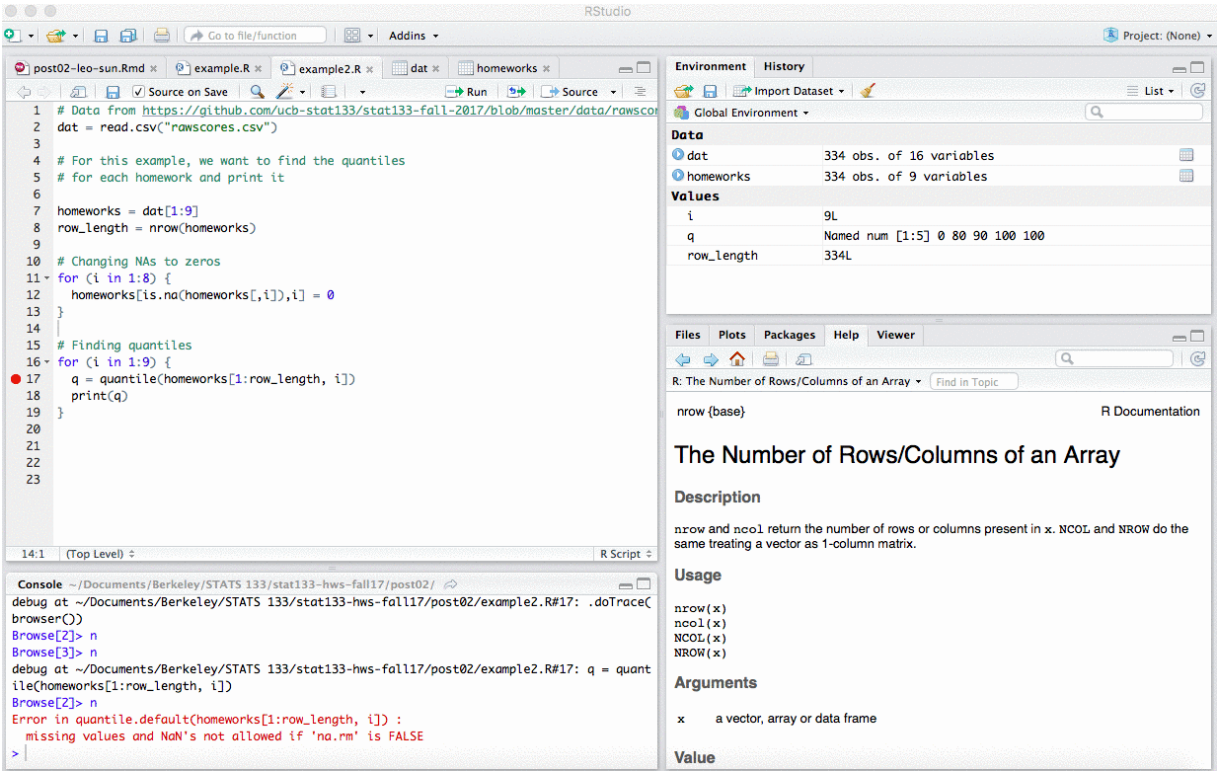
```r
# Data from https://github.com/ucb-stat133/stat133-fall-2017/blob/master/data/rawscores.csv
dat = read.csv("rawscores.csv")

# For this example, we want to find the quantiles
# for each homework and print it

homeworks = dat[1:9]
row_length = nrow(homeworks)

# Changing NAs to zeros
for (i in 1:8) {
  homeworks[is.na(homeworks[,i]),i] = 0
}

# Finding quantiles
for (i in 1:9) {
  q = quantile(homeworks[1:row_length, i])
  print(q)
}
```

If we run this script, it seems to work until we get this error:

```
Error in quantile.default(homeworks[1:row_length, i]) :
  missing values and NaN's not allowed if 'na.rm' is FALSE
```

Let's debug this!

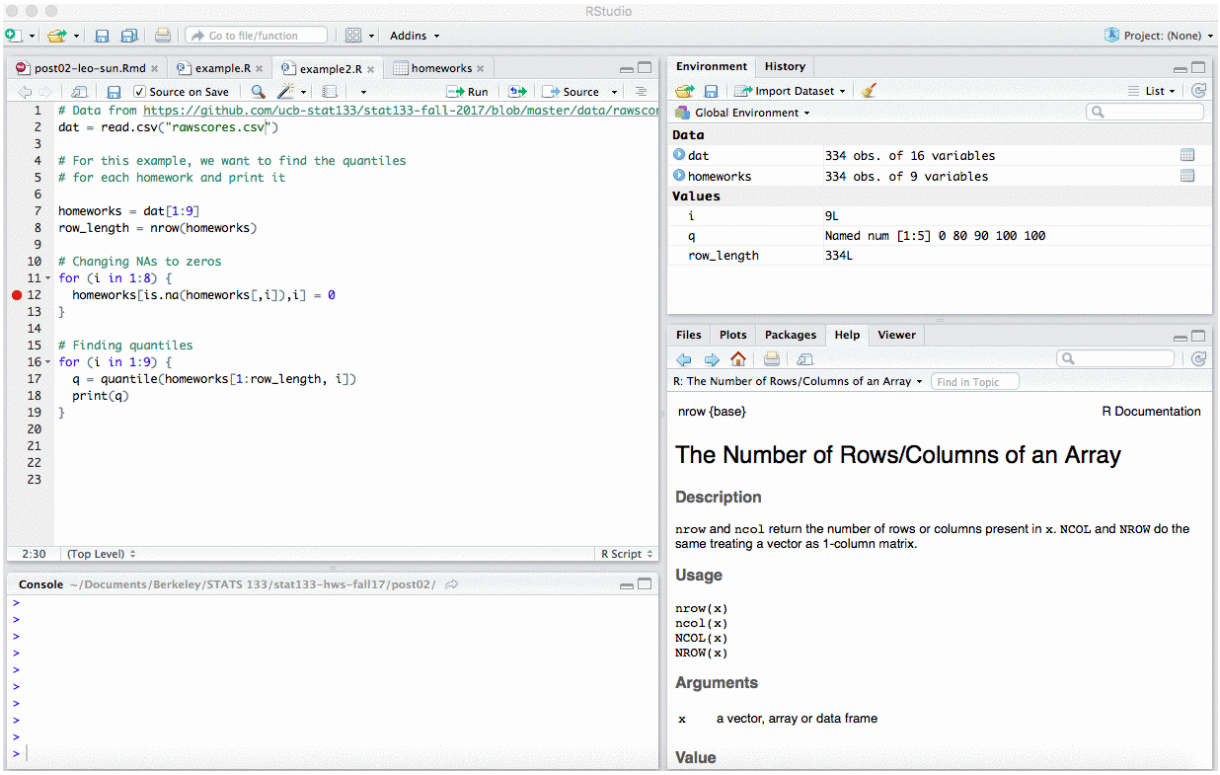
Set a breakpoint on line 17. Keep executing the next line until we get the error.



After printing the value of i, we can infer that the error occurred on the last quantile call.

The error states that missing values and NaN's are not allowed, which means that there are NA values in our data set.

Let's set another breakpoint on line 12.



After executing through the for loop and checking our homework data frame, we can see that the code is working. The problem is that the last column's values are not being changed.

If we look at the for loop condition, we can see that the loop only has 8 iterations, 1 less than the amount of columns in our data. Setting it to 9 fixes our problem and the script runs.

```
# Changing NAs to zeros
for (i in 1:9) { # Change 8 to 9
  homeworks[is.na(homeworks[,i]),i] = 0
}
```

Closing Thoughts

Debugging is a powerful tool. It allows us to examine the fine details of our scripts and functions to see what is causing the error. Debugging is helpful when we don't know what exactly is causing the issue. The 2nd example has an obvious error, but not all code will look as simple as that. It is critical to use common sense before debugging, in order to catch simple errors. For more advance debugging and error catching tricks, I recommend reading Hadley Wickham's documentation on debugging.

References

https://en.wikipedia.org/wiki/Debugging

https://en.wikipedia.org/wiki/Scope_(computer_science)

https://www.rstudio.com/products/rstudio/release-notes/debugging-with-rstudio/

https://support.rstudio.com/hc/en-us/articles/200713843?version=1.0.153&mode=desktop

https://github.com/ucb-stat133/stat133-fall-2017/blob/master/data/rawscores.csv

http://seananderson.ca/2013/08/23/debugging-r.html

http://adv-r.had.co.nz/Exceptions-Debugging.html