An Introduction to R

Part 2

Joey Stanley
Doctoral Candidate in Linguistics, University of Georgia
joeystanley.com
orcid.org/0000-0002-9185-0048

Presented at the UGA Willson Center DigiLab
Friday, January 26, 2018

This is the second installment of the R workshop series, and the second workshop that introduces R. This document will cover some of the basics of R including the following topics: (1) R Basics, such as using R as a calculator, variables, and basic functions; (2) Getting data into R from a .csv, .txt, or other data types with a tangent into R packages; (3) working with, displaying, extraction portions of, and filtering your data, with another tangent on logical operators; (6) where to go for help, both in R and on the internet.

Download this PDF from my website at
joeystanley.com/r2018

(Updated January 25, 2018)

1 Introduction

If you're reading this handout, I'm assuming you meet the following basic prerequisites:

1. You know what R is.
2. You know what RStudio is.
3. You have both R and RStudio installed an running on your computer.

That's it. Today's workshop is meant to be a really basic introduction to the R language.

2 R Basics

2.1 R as a calculator

Obligatorily, any tutorial on R has to show you that R can be a calculator. In your R script, type the following:

```
2+2
```

With your cursor still on this line, hit command+enter (or for Windows, ctrl+enter), which is the keyboard shortcut for "execute this line". In the bottom left quadrant of RStudio, you'll now see a new line, in blue `>2+2` and then, in black, the output `[1] 4`.

```
## [1] 4
```

You've just executed your first R command! You can think of the bottom left quadrant, the *console*, as the actual R portion of RStudio. The script above it is just a placeholder for the various commands. When you want to run a command, RStudio will send it down to the console and execute it. Once it does that, the next line in the console will be the output, which in this case is `4`. The `[1]` before it just tells you that the output is actually a list that is one unit long. No need to worry about that right now.

You can do other arithmetic in R as well:

```
10*(5-2)/3+1

## [1] 11
```

Not too shabby. But this is boring. Let's move on to bigger and better things.

2.2 Variables

You can create *variables* to store data. Variables have arbitrary names, so you can call them whatever you want. To create a variable, provide it's name (in this case `six`), then the assignment operator `<-`, and some value. (Mac users: a keyboard shortcut for the assignment operator is option+dash.) Let's create a variable called `six` and give it the value of `6`.

```
six <- 6
```

Okay great. Now what we can do is use this `six` variable as if it were any other number.

```
six + 3

## [1] 9

six * 2

## [1] 12

1 / six

## [1] 0.1666667
```

Just to show the names really are arbitrary, you can use whatever names you want and it'll still work.

```
blue <- 4
six + blue

## [1] 10

seven <- 8
dog <- -5
blue * seven + dog

## [1] 27
```

It's probably not a good idea to use these kinds of variable names—and in general it's good practice to give brief but descriptive variable names—but it goes to show you can use whatever names you want. Think of variables as storage containers: what you write on the outside should be brief but informative.

You can store multiple numbers in a single variable using the `c()` command, which stands for "combine". Let's create a new variable called `fibs` and have it contain the first several numbers of the Fibonacci sequence.

```
fibs <- c(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144)
```

Technically this list of numbers is called a *vector* in R, specially, a *double vector*. This doesn't mean that there's anything repeated or doubled: "double" is a term used in computer science to essentially mean number. What we have is a vector of numbers. What's cool about these is that you can treat it like a single number, and it'll run the command on each element of the vector.

```
fibs / 2

##  [1]  0.0  0.5  0.5  1.0  1.5  2.5  4.0  6.5 10.5 17.0 27.5 44.5 72.0

six * fibs

##  [1]   0   6   6  12  18  30  48  78 126 204 330 534 864
```

You can display the entire contents of the vector by simply typing the name of the variable.

```
fibs

##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144
```

But if you want to access a single item in that list, type the variable name and in square brackets immediately after the variable name, type what the which element you'd like to access. For example, here's how you would access the first, fifth, and tenth elements of the list

```
fibs[1]

## [1] 0

fibs[5]

## [1] 3

fibs[10]

## [1] 34
```

You can even pass in a list of numbers! If you want to see all three of these elements of the list at once, just wrap them up in the `c` function and put that in place of the number.

```
fibs[c(1,5,10)]

## [1]  0  3 34
```

These lists are really important, because when it comes time to read in your own spreadsheets into R, each column will be treated as a list just like this one. We'll get to that in a sec.

2.3 Functions

This is great and all. We can do more though. Here, let's look at what are called *functions*. These are chunks of code all wrapped up into one "word." These take one or more *arguments*. The function takes these arguments, works its magic under the hood, and returns some value. For example, the `sqrt` function takes the square root of some number.

```
sqrt(16)
```

```
## [1] 4
```

Note that the functions are case sensitive, and the arguments go in parentheses. If there are multiple arguments, they are separated by commas. Some other functions include `sum`, `mean`, and `range` (`range` shows the highest and lowest values). You can pass variables as arguments, and you can even nest functions.

```
sum(six, 3, dog)

## [1] 4

mean(fibs)

## [1] 28.92308

range(fibs, -4, sqrt(16), six)

## [1]  -4 144
```

You should get very comfortable running functions and getting all the syntax right like keeping track of your parentheses, commas, and spelling. The idea that you run functions, and on functions within functions, is super common in R. You don't have to memorize all the functions that R has, and in fact, you'll probably never use them. You're probably wondering what all the functions in R even are and how you're supposed to know about them. Google. There is tons of documentation for R online, and you'll hopefully be able to find your answer very quickly.

3 Getting data into R

At this point, a lot of R tutorials start you off with working with generating data within R itself. While this is an important skill to learn eventually, what's most relevant to you right now is getting your own data into R.

I'm going to assume you have your own spreadsheet somewhere saved on your computer. Ideally, each row represents one *observation* and each column is a variable of that observation. For example, if you had a spreadsheet that had the area, population, and capital of each of the 50 states, you'll have 50 rows and 4 columns (one for each of the variables and the fourth for the state name). Presumably, your data is clean and tidy, meaning that dates and numbers are formatted the same, capitalization is standardized, and everything is consistent. I could go on for a long time about the importance of making your data clean, but suffice it to say that it's paramount for proper analysis in R.

The functions for reading in data in R depend on what kind of file you have. The most common options are `.csv`, `.txt`, and an Excel file. Let's look at each one of those.

3.1 .CSV FILES

I've got a file on my computer called `menu.csv` that contains all the menu items at McDonald's with complete nutrition information and is available for free at *Kaggle.com*. The file is a *comma-separated values* format (it ends with .csv), which is probably the easiest way to go in R. This means that it's essentially a table, with each row on its own line, and each column separated by commas.

You can use `read.csv`, and as the only argument, put the *path* to the file you want to read in, in quotation marks. Ideally, the file should be close to, if not in the same folder as, this script. That way you only need to type a relative path.

```
read.csv("menu.csv")
```

If it's somewhere else on your computer, you can type a full path:

```
read.csv("/Users/joeystanley/Desktop/menu.txt") # on a Mac
read.csv("C:\\Users\\joeystanley\\Desktop\\menu.txt") # on a Windows
```

Note that Mac users should use single forward slashes while Windows users have to use double back slashes.

Alternatively, I have the file on my website, and you can actually read it in straight from there.

```
read.csv("http://joeystanley.com/downloads/menu.csv")
```

When you do this, you'll start to see the contents of your file displayed in your R console (the bottom left portion of the RStudio screen). Hooray! You just read your data into R!

Unfortunately, all it did was read it in and forget it. Computers do *exactly* as they're told. What you didn't tell R was to *remember* the contents of the file. So, let's create a new variable called `menu` and save the contents of the `menu.csv` file into that variable.

```
menu <- read.csv("menu.csv") # for Macs
```

Okay, so now we have a new `menu` object that has the full contents of that file. Before we move on to working with that file, let's see how to read in files that are in other formats.

3.2 .TXT FILES

If your file is a *tab-delimited* file (it ends with .txt), then you can use `read.table` instead. As an additional argument (which is separated from the path name with a comma), you may need to specify that the cells of your table are separated by tabs. To do this, add the `sep="\t"` argument (`\t` is "computer-talk" for a tab). Also, your file may have a header, meaning the first row of your file might contain the names of the columns. By default, `read.csv` assumes this, but for `read.table` you'll need to make that explicit so R knows what to do with them. You can add this using the `header=TRUE` argument. So the final command might look like this.

```r
menu <- read.table("/Users/joeystanley/Desktop/menu.csv", sep="\t", header=TR
UE) # for Macs
menu <- read.table("C:\\Users\\joeystanley\\Desktop\\menu.txt", sep="\t", hea
der=TRUE) # for Windows
```

3.3 Excel files

A lot of my data is stored in an Excel file (.xlsx), but as far as I know, there is no "base R" function that lets you read in Excel files into R. What do we do?

3.3.1 Tangent: Packages

One of the best things that makes R great is that it's open source and so people have developed their own *packages* that do that. R packages are bundles of code that other people have written and made available for others to download. These usually contain several additional functions that plain ol' R can't handle very easily. The best way I know of to read in Excel files is to use the `read_excel` function that's in the `readxl` library.

By the way, `readxl` is one of several packages that together comprise what's called the `tidyverse` and is written by Hadley Wickham. Later this semester, I'll be doing two workshops specifically on `tidyverse`.

To download `readxl`, we first have to install it. We can do this with `install.packages()`.

```r
install.packages("readxl")
```

That'll run some code in your R Console and will take a few seconds to install. You'll need the internet for this too since we're downloading stuff. Once it's done, the code is installed to your computer.

However, R doesn't make downloaded packages available to you by default. One main reason is because it would slow down R and your computer if it had to keep track of all of them at once. Also, two people might create two different functions with the exact same name, which would create a clash. By only loading the packages you need in any particular R session, this avoids clashes and makes your computer run faster.

In order to make the functions in that package available for use in this session, you have to use the `library` function (with the package name *not* in quotes).

```r
library(stringr)
```

Now, we have at our disposal a whole bunch of new functions specifically geared towards working with Excel.

Note that you only need to install a package once ever. So run the code, and then erase it from your script because once it's on your computer it's on and you don't need to keep installing it every time you run R. However, you *do* need to load the package every time you open R. So keep the `library()` function in your script.

3.3.2 Back to working with Excel

Now that we've got `readxl` loaded, we can read data into R from an Excel file.

```
menu <- read_excel("http://joeystanley.com/downloads/menu.xlsx", sheet = 2)
```

You can optionally specify which sheet of the file to read in. Otherwise, it'll assume the first sheet.

Also nota that because this package is part of the Tidyverse, it'll formats your data a little differently. This will make sense when we get to those workshops.

3.4 Don't like typing path names?

There is one other way to read in data that doesn't involve typing those long path names. You can use the `file.choose` command instead. When you do this, a window will open up and you'll be able to find your file and click on it just like you were opening any other file.

```
menu <- file.choose() # for macs
menu <- choose,file() # for Windows
```

I don't like this option mostly because it just takes too many clicks. Every time I want to run this line of code, it takes 5 or so clicks to get the file I want. This gets really tedious. It's worth the time to just type (or copy and paste!) the path name to the file one time, and with a single keystroke you can load it in nearly instantaneously the exact same way every time.

4 Working with your data

Okay, so your data is in R. Now we need to be able to view it to make sure it's all there, and also be able to extract portions of it.

4.1 Displaying your data

The easiest way to display your data is to simply type the name of the variable itself. However, depending on the size of your data frame, this could get huge. I've truncated mine, particularly the description so it would fit on this page, but yours might be different.

```
menu

##     Category                     Item  Oz Calories Fat Sugars
## 1  Breakfast               Egg McMuffin 4.8      300  13      3
## 2  Breakfast          Egg White Delight 4.8      250   8      3
## 3  Breakfast           Sausage McMuffin 3.9      370  23      2
## 4  Breakfast Sausage McMuffin with Eg 5.7      450  28      2
## 5  Breakfast Sausage McMuffin with Eg 5.7      400  23      2
## 6  Breakfast      Steak & Egg McMuffin 6.5      430  23      3
## 7  Breakfast Bacon, Egg & Cheese Bisc 5.3      460  26       3
## 8  Breakfast Bacon, Egg & Cheese Bisc 5.8      520  30       4
## 9  Breakfast Bacon, Egg & Cheese Bisc 5.4      410  20       3
## 10 Breakfast Bacon, Egg & Cheese Bisc 5.9      470  25       4
## 11 Breakfast Sausage Biscuit (Regular 4.1      430  27       2
## 12 Breakfast Sausage Biscuit (Large B 4.6      480  31       3
## 13 Breakfast Sausage Biscuit with Egg 5.7      510  33       2
## 14 Breakfast Sausage Biscuit with Egg 6.2      570  37       3
## 15 Breakfast Sausage Biscuit with Egg 5.9      460  27       3
## 16 Breakfast Sausage Biscuit with Egg 6.4      520  32       3
## 17 Breakfast Southern Style Chicken B 5.0      410  20       3
## 18 Breakfast Southern Style Chicken B 5.5      470  24       4
## 19 Breakfast Steak & Egg Biscuit (Reg 7.1      540  32       3
## 20 Breakfast Bacon, Egg & Cheese McGr 6.1      460  21      15
```

This may spill over onto multiple lines. That's only because my screen isn't wide enough to display the full row. An alternative that is easier for scrolling, is to use the `View` function (yes, that's a capital *V*). This opens up a new tab in RStudio, and you'll be able to view your data like you would a normal spreadsheet.

```
View(menu) # Not run here because we're not in RStudio.
```

You can also just display portions of your data using `head` and `tail`, which, respectively, show the first and last couple of rows.

```
head(menu)
```

```
##    Category                     Item  Oz Calories Fat Sugars
## 1 Breakfast             Egg McMuffin 4.8      300  13      3
## 2 Breakfast        Egg White Delight 4.8      250   8      3
## 3 Breakfast         Sausage McMuffin 3.9      370  23      2
## 4 Breakfast Sausage McMuffin with Eg 5.7      450  28      2
## 5 Breakfast Sausage McMuffin with Eg 5.7      400  23      2
## 6 Breakfast     Steak & Egg McMuffin 6.5      430  23      3
```

```
tail(menu)
```

```
##                Category                       Item   Oz  Calories Fat Sugars
## 255 Smoothies & Shakes McFlurry with M&M\x89▯s Can  7.3      430  15      5
9
## 256 Smoothies & Shakes   McFlurry with Oreo Cooki 10.1      510  17     64
## 257 Smoothies & Shakes   McFlurry with Oreo Cooki 13.4      690  23     85
## 258 Smoothies & Shakes   McFlurry with Oreo Cooki  6.7      340  11     43
## 259 Smoothies & Shakes   McFlurry with Reese's Pe 14.2      810  32    103
## 260 Smoothies & Shakes   McFlurry with Reese's Pe  7.1      410  16     51
```

4.2 Extracting portions of your data

Before, when we had several numbers saved into a single variable, we called it a *vector*. Now, we have an entire spreadsheet saved into the `menu` variable. Each column in your data frame is treated as a vector, and each row of that column is an element in that list. We call this type of variable a *dataframe*.

You can think of a vector as a one-dimensional variable and a data frame as a two-dimensional variable. To access an element of a vector, you type one number in the square brackets (`fibs[1]`). Because a data frame is two-dimensional, you'll have to send two numbers, separated by a comma: the first for the row number and the second for the column number.

```
head(menu)

##    Category                     Item  Oz Calories Fat Sugars
## 1 Breakfast              Egg McMuffin 4.8      300  13      3
## 2 Breakfast         Egg White Delight 4.8      250   8      3
## 3 Breakfast          Sausage McMuffin 3.9      370  23      2
## 4 Breakfast Sausage McMuffin with Eg 5.7      450  28      2
## 5 Breakfast Sausage McMuffin with Eg 5.7      400  23      2
## 6 Breakfast     Steak & Egg McMuffin 6.5      430  23      3

menu[2,5]

## [1] 8
```

The `8` is content of the element you just extracted. It's the fifth column of the second row, which is how many grams of fat are in a "Egg White Delight". You can extract an entire row by leaving off the column number (but still keeping the comma).

```
menu[2,]

##    Category              Item  Oz Calories Fat Sugars
## 2 Breakfast Egg White Delight 4.8      250   8      3
```

Notice how the output shows the column names along the top as well as the row number on the left side. You can extract an entire column by leaving off the row number, again only if you keep the comma. (I've truncated my list for display purposes: yours will be a lot longer.)

```
menu[,5]

##  [1] 13  8 23 28 23 23 26 30 20 25
```

As an alternative to extracting an entire column, it's easier to refer to the column by its name (`Fat`) rather than its number (`5`). This is especially true if you have many columns in your spreadsheet and you don't feel like counting all of them. You can refer tho the column using the dollar sign `$`, which is placed between the variable name and the column name.

```
menu$Item

##  [1] "Egg McMuffin"             "Egg White Delight"
##  [3] "Sausage McMuffin"         "Sausage McMuffin with Eg"
##  [5] "Sausage McMuffin with Eg" "Steak & Egg McMuffin"
##  [7] "Bacon, Egg & Cheese Bisc" "Bacon, Egg & Cheese Bisc"
##  [9] "Bacon, Egg & Cheese Bisc" "Bacon, Egg & Cheese Bisc"
```

4.3 Filtering your data

We can also perform queries on your data so that we can filter your data in whatever way we want. We can view entire portions of the data frame just as easily in Excel as we can in R, so why bother with R in the first place?

In Excel it's certainly possible to filter your data. But it gets a little cumbersome if you have multiple filters on at once. Or if you have to switch back and forth between two or more different filters, you have to do a lot of clicking. With R, it's a little bit easier.

4.3.1 *Tangent: Logical Operators*

To use filters, I'll have to introduce a couple ways to *evaluate* data. What a filter does is it looks at your data, and decides whether certain rows meet certain conditions. If we start simple, and take our `fibs` vector, we can see how many of those numbers are greater than 10.

```
fibs

##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144

fibs > 10

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE
```

Here, the first 7 items in our list are `FALSE`, meaning they are not greater than 10. The last 6 are `TRUE`, so they are greater than 10. Similar to greater than, which uses the rightward-pointing angled bracket, we can use the following in similar ways:

- `>` means "greater than"
- `<` means "less than"
- `>=' means "greater than or equal to"
- `<=` means "less than or equal to"
- `==` means "equal to". Note here that you need *two* equals signs, and not just one.
- `!=` means "not equal to"

Here are some examples.

```
fibs > 10

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE

fibs < 20
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [12] FALSE FALSE

fibs >= 55

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [12]  TRUE  TRUE

fibs <= 13

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [12] FALSE FALSE

fibs == 1

##  [1] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE

fibs != 34

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
## [12]  TRUE  TRUE
```

4.3.2 Back to the data

We can use the same kind of thing to subset your data. For example, if we want to find all menu items that have 0 fat, we can do so like this.

```
menu[menu$Fat == 0,]

##          Category                     Item   Oz Calories Fat Sugars
## 101 Snacks & Sides               Side Salad  3.1       20   0      2
## 102 Snacks & Sides             Apple Slices  1.2       15   0      3
## 111      Beverages Coca-Cola Classic (Small 16.0      140   0     39
## 112      Beverages Coca-Cola Classic (Mediu 21.0      200   0     55
## 113      Beverages Coca-Cola Classic (Large 30.0      280   0     76
## 114      Beverages Coca-Cola Classic (Child 12.0      100   0     28
## 115      Beverages        Diet Coke (Small) 16.0        0   0      0
```

Let's break this down. First, we have the `menu[]` template like we've been doing before to display portions of the `menu` variable. Next, if you look carefully, we have some stuff, followed by a comma, and then nothing: `menu[...,]`. This is exactly like what we did before where we extracted an entire row and displayed all columns. Only this time, instead of a row number, we're giving a true/false statement. We're referring to just the `Fat` column in the data frame, like we did before. Since each column in a data frame is a vector, we can treat it like we did with `fibs` above and find which of the elements meet the qualification. So, we're finding all rows such that the `Fat` column of that row is less than or equal to `0`.

There are 49 fat free menu items, but some of them, like the sodas, have a lot of sugar. We can apply another filter to the data and remove anything where the `Category` is `"Beverages"`. To do that, all we need to do is put an ampersand `&` after the filter but before the comma, and just type another filter.

```
menu[menu$Calories <= 150 & menu$Category != "Beverages",]
```

```
##           Category                      Item   Oz Calories Fat Sugars
## 39       Breakfast                Hash Brown  2.0      150 9.0      0
## 85          Salads Premium Bacon Ranch Sala  7.9      140 7.0      4
## 88          Salads Premium Southwest Salad   8.1      140 4.5      6
## 100 Snacks & Sides         Kids French Fries  1.3      110 5.0      0
## 101 Snacks & Sides                Side Salad  3.1       20 0.0      2
## 102 Snacks & Sides              Apple Slices  1.2       15 0.0      3
## 103 Snacks & Sides  Fruit 'n Yogurt Parfait  5.2      150 2.0     23
## 106       Desserts    Oatmeal Raisin Cookie  1.0      150 6.0     13
```

Looks like that got rid of the sodas and juices, but we probably want to get rid of the coffee and tea as well if we really just want to display foods. We can add a third filter just as we added the second filter.

```
menu[menu$Calories <= 150 &
         menu$Category != "Beverages" &
         menu$Category != "Coffee & Tea",]
```

```
##           Category                      Item  Oz Calories Fat Sugars
## 39       Breakfast                Hash Brown 2.0      150 9.0      0
## 85          Salads Premium Bacon Ranch Sala 7.9      140 7.0      4
## 88          Salads Premium Southwest Salad  8.1      140 4.5      6
## 100 Snacks & Sides         Kids French Fries 1.3      110 5.0      0
## 101 Snacks & Sides                Side Salad 3.1       20 0.0      2
## 102 Snacks & Sides              Apple Slices 1.2       15 0.0      3
## 103 Snacks & Sides  Fruit 'n Yogurt Parfait 5.2      150 2.0     23
## 106       Desserts    Oatmeal Raisin Cookie 1.0      150 6.0     13
## 107       Desserts      Kids Ice Cream Cone 1.0       45 1.5      6
```

Note that I spread the command onto multiple lines. To me, it's easier to read. R is not the most English-like language, so the more you can make it easier to read, the better you'll be when you come back to this code tomorrow. In fact, you can start to add comments to your code, using the `#` symbol:

```
# Filter out all the unhealthy stuff.
menu[menu$Calories <= 150 &  # Anything with 150 calories or less
         menu$Category != "Beverages" &  # not including soda and juice
         menu$Category != "Coffee & Tea",] # not including coffee and tea
```

For fun, we can see what the really fatty or sugary food are.

```
menu[menu$Calories > 1000 | menu$Sugars > 100,]
```

```
##               Category                      Item   Oz Calories Fat Sugars
## 32           Breakfast   Big Breakfast with Hotca 14.8     1090  56     17
## 33           Breakfast   Big Breakfast with Hotca 15.3     1150  60     17
## 35           Breakfast   Big Breakfast with Hotca 15.4     1050  50      18
## 83      Chicken & Fish   Chicken McNuggets (40 pi 22.8     1880 118       1
## 244 Smoothies & Shakes      Vanilla Shake (Large) 22.0      820  23     101
## 247 Smoothies & Shakes   Strawberry Shake (Large) 22.0      850  24     123
## 250 Smoothies & Shakes    Chocolate Shake (Large) 22.0      850  23     120
## 252 Smoothies & Shakes     Shamrock Shake (Large) 22.0      820  23     115
## 254 Smoothies & Shakes McFlurry with M&M\x89□s Can 16.2      930  33      12
8
## 259 Smoothies & Shakes   McFlurry with Reese's Pe 14.2      810  32     103
```

So that's how you might extract portions of your dataset.

5 Where to go for help

5.1 In-R Help

R offers some good resources to help you learn about its functions. For example, if you put a question mark `?` before a function name, on the lower right quadrant of RStudio, some help will pop up.

```
?sqrt
```

These give documentation about how to use the function, what arguments it takes, and some example code. Usually, I scroll down to the example code and can find what I need there. It takes some experience to understand what the help pages do to be honest, but they eventually get to be very useful.

You can also precede a function name with two question marks `??` to search all of your installed packages for it.

```
??read_excel
```

This is useful if you know there's some sort of R function but you don't remember what package it's in. So if you get some code with the function `read_excel` in it but you forgot to load the `readxl` package, you can search for it using this double question mark and it'll tell you which ones to load.

5.2 Books and websites

Because R is so widely used, there are tons of resources out there. Here, I list just a few sites and books that I have found useful on a variety of topics.

- *An Introduction to R* by Venables, Smith, and the R Core Team (2017). This is a 99-page PDF that introduces R and some basic skills on how to use it. This version is only a few months old and it a thorough resource for beginners.

- This *R Cookbook* site is great and has helped me a lot.
- The *tidyverse* website is the launchpad for learning to use the `tidyverse` package.
- The publishing company Springer has a series called *Use R!* that has over 50 volumes in it. Many of them cover general skills like `ggplot2` or Shiny, but others are geared towards specific field like business, ecology, biostatistics, and general statistical procedures. The ones that might be most useful for beginners include *A Beginner's Guide to R* and *R Through Excel*. These books are all available as free PDFs to download through UGA's library
- *Lynda.com*, which is free for UGA students, has some great help for learning R.

Your first stop really should just be google though, which will often take you to StackOverflow, YouTube, or other websites.

6 Conclusions

The goal for this workshop was to expose you to the kinds of things that are possible with R, to give you some exposure to a few R concepts, and to point you in the right direction to learn more. As a review, we covered the following ideas:

- What R is and how it compares to some alternatives
- The difference between R and RStudio.
- Basic R skill: Installing R and RStudio, variables, functions, getting data into R with a tangent into packages
- Displaying your data, extracting portions of it, and filtering, with a tangent into logical operators
- Where to find help.

This workshop is not enough to be a game-changer for you: you'll have to take the initiative to learn more on your own. But hopefully it has got you curious enough to make you want to learn more about R.