Using R

David John Baker

12/23/2018

Lesson Goals

- Run Basic Commands in R
- Understand Basic Data Structures
- Run commands over vectors
- Index data frames
- Learn basic data structures
- Understand base R vs Tidyverse
- Import and export data to/from R

RStudio Shortcuts and Markdown

In this session I will be using a lot of Keyboard Shortcuts when typing myself. In the past, people have always asked about these, so I'm anticipating that question with a link here to that page.

R as Calculator

The Console of R is where all the action happens. You can use it just like you would use a calculator. Try to do some basic math operations in it.

```
2 + 2
```

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

```
5 - 2
```

```
## [1] 3
```

```
10 / 4
```

```
## [1] 2.5
```

```
9 * 200
```

```
## [1] 1800
```

```
sqrt(81)
```

```
## [1] 9
```

```
10 > 4
```

```
## [1] TRUE
```

```
2 < -1
```

```
## [1] FALSE
```

Now from the output above, you'll notice that there are a few different types of responses that R will give. For the math responses, we get numbers, but we can also get TRUE and FALSE statements.

When working with data, we need to be aware not only of what the data represents, but what R thinks it represents. We won't go over the differences between things like ordinal, ratio, and categorical data, I'll assume you have a basic understanding of this. What we will focus on is the different data types that R thinks in.

For now, we are going to talk about R's basic data structures.

- Logical
- Integer
- Double (numeric)
- Character
- Factor

The first is logical. Logical is basically just TRUE or FALSE. We can try a few different expressions that show how this works.

```
2 > 4
```

```
## [1] FALSE
```

```
1 > 0
```

```
## [1] TRUE
```

```
4 >= 7
```

```
## [1] FALSE
```

```
5 != 5
```

```
## [1] FALSE
```

Eventually you will learn to take advantage of the complexities of this when we get to subsetting and combining them with other logical operators like `&`(and) and `|` (OR).

Next we have integers and double. Both integers and double are R's numeric forms of data. The `is.numeric()` command checks for if data is number-y.

```r
is.integer(7L)
```

```
## [1] TRUE
```

```r
is.double(7)
```

```
## [1] TRUE
```

```r
is.numeric(7)
```

```
## [1] TRUE
```

Next we have characters. Characters are not *just* letters, but rather data that is text. Character data is always wrapped in quotes `" "`

```r
is.character("hello, world!")
```

```
## [1] TRUE
```

```r
is.character("7")
```

```
## [1] TRUE
```

```r
is.character("I will drink 7 coffees by the end of today!")
```

```
## [1] TRUE
```

```r
is.character("NA")
```

```
## [1] TRUE
```

Note that if a special character like `NA` is in quotes, R will still think it is a character. To change this, we need to coerce our data into a different type. We will cross that bridge later. For now, you just need to be aware of the different character types.

Lastly, there are factors which sometimes LOOK like characters, but are R's way of thinking about categorical data. We need to assign this to R. When you first import in data into R, it will sometimes guess it as being a factor which is very annoying! If R is being slow, or not responding to something you want it to do, a common rookie mistake is to have your data accidentally be a factor.

```r
is.factor("doggo")
```

```
## [1] FALSE
```

```r
doggo <- as.factor("doggo")

is.factor(doggo)
```

```
## [1] TRUE
```

```
is.character(doggo)
```

```
## [1] FALSE
```

```
is.numeric(doggo)
```

```
## [1] FALSE
```

Now that we're at least aware of the different types of data in R, we can move on to building up an intuitive understanding of how R thinks about data under the hood.

Being Lazy

You don't always want to print your output and retype it in. The idea of being a good programmer is to be very lazy (efficient).

One of the best ways to be efficient when programming is to save variables to objects. Below is some example code that uses the `<-` operator to assign some math to an object. After you assign it to an object, you can then manipulate it like you would any other number. Yes, you can use `=` as an assignment operator (for all you Pythonistas), but in R this is considered bad practice as R is primarily a statistical programming language and the `=` sign means something very different in a math context.

```
foo <- 2 * 3
foo * 6
```

```
## [1] 36
```

After running these two lines of code, notice what has popped up in your environment in RStudio! You should see that you now have any object in the Environment called `foo`.

In addition to saving single values to objects, you can also store a collection of values. Below we use an example that might have a bit more meaning, the below stores what could be some data into an object that represents what it might be.

```
yearsSellingWidgets <- c(2,1,4,5,6,7,3,2,4,5,3)
```

The way that the line above works is that we use the `c()` function (c for combine) to group together a bunch of the same type of data (numbers) into a vector. Once we have everything combined and stored into an object, we can then manipulate all the numbers in the object just like we did above with a single number. A single dimensional object is called a **vector**. For example, we could multiply all the numbers by three.

```
yearsSellingWidgets * 3
```

```
##  [1]  6  3 12 15 18 21  9  6 12 15  9
```

Or maybe we realized that our inputs were wrong and we need to shave off two years off of each of the entries.

```
yearsSellingWidgets - 2
```

```
##  [1]  0 -1  2  3  4  5  1  0  2  3  1
```

Or perhaps we want to find out which of our pieces of data (and other data associated with that observation) are less than 2.

```
yearsSellingWidgets < 2
```

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

Any sort of mathematical operation can be performed on a vector! In addition to treating it like a mathematical operation, we can also run functions on objects. By looking at the name of each function and it's output, take a guess at what each of the below functions does.

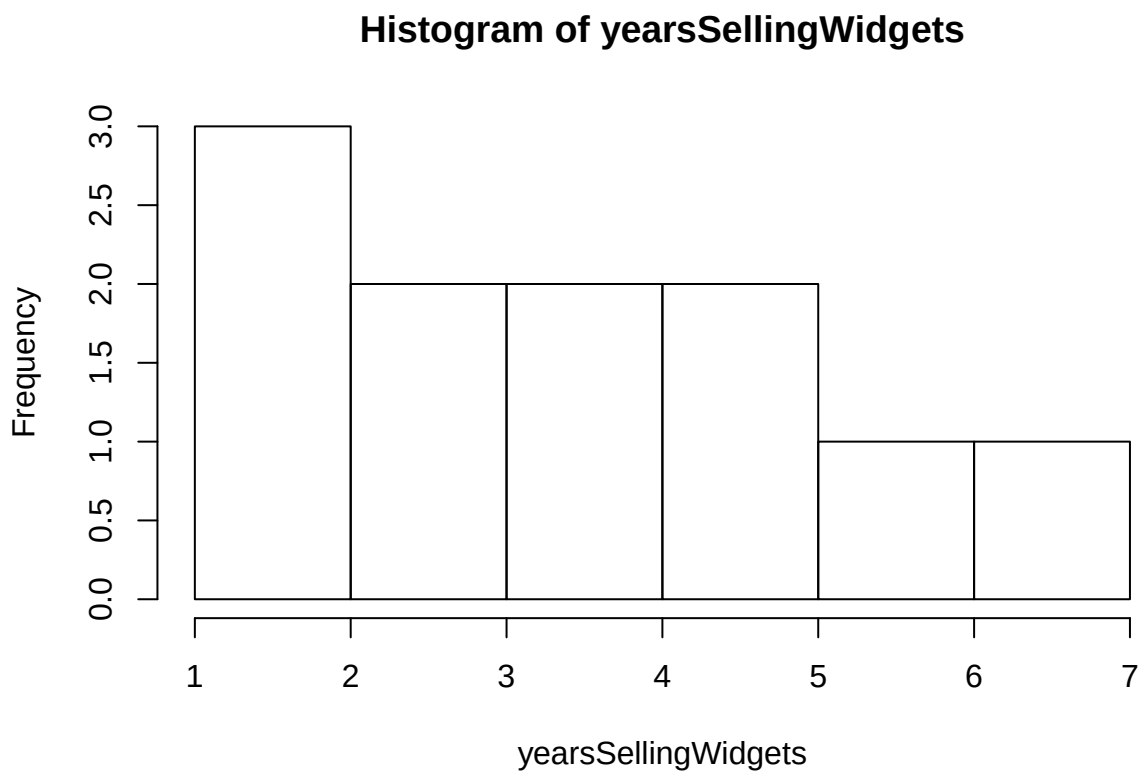
```
mean(yearsSellingWidgets)
```

```
## [1] 3.818182
```

```
sd(yearsSellingWidgets)
```

```
## [1] 1.834022
```

```
hist(yearsSellingWidgets)
```



```
scale(yearsSellingWidgets)
```

```
##              [,1]
##  [1,] -0.99136319
##  [2,] -1.53661295
##  [3,]  0.09913632
##  [4,]  0.64438608
##  [5,]  1.18963583
##  [6,]  1.73488559
##  [7,] -0.44611344
##  [8,] -0.99136319
##  [9,]  0.09913632
## [10,]  0.64438608
## [11,] -0.44611344
## attr(,"scaled:center")
## [1] 3.818182
## attr(,"scaled:scale")
## [1] 1.834022
```

```
range(yearsSellingWidgets)
```

```
## [1] 1 7
```

```
min(yearsSellingWidgets)
```

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

```
class(yearsSellingWidgets)
```

```
## [1] "numeric"
```

```
str(yearsSellingWidgets)
```

```
##  num [1:11] 2 1 4 5 6 7 3 2 4 5 ...
```

```
summary(yearsSellingWidgets)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   2.500   4.000   3.818   5.000   7.000
```

Often working with data, we don't want to just play with one group of numbers. Most of the time we are trying to compare different observations in data science. If we then create two vectors (one of which we have already made!) and then combine them together into a data frame, we have something sort of looking like a spreadsheet. A two-dimensional object is called a **data frame**.

```
yearsSellingWidgets <- c(2,1,4,5,6,7,3,2,4,5,3)
numberOfSales <- c(5,2,5,7,9,9,2,8,4,7,2)
salesData <- data.frame(yearsSellingWidgets,numberOfSales)
salesData
```

```
##    yearsSellingWidgets numberOfSales
## 1                    2             5
## 2                    1             2
## 3                    4             5
## 4                    5             7
## 5                    6             9
## 6                    7             9
## 7                    3             2
## 8                    2             8
## 9                    4             4
## 10                   5             7
## 11                   3             2
```

Now if we wanted to use something like R's correlation function we could just pass in the two objects that we have like this and get a correlation value.

```
cor(yearsSellingWidgets,numberOfSales)
```

```
## [1] 0.6763509
```

But often our data will be saved in data frames and we need to be able to access one of our vectors inside our data frame. To access a piece of information in a data frame we use the `$` operator.

```
salesData$yearsSellingWidgets
```

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

Running the above code will print out the vector called `yearsSellingWidgets` from the data frame `salesData`. Using this form, we can then use this with the correlation function.

```
cor(salesData$yearsSellingWidgets,salesData$numberOfSales)
```

```
## [1] 0.6763509
```

In addition to just getting numeric output, we also want to be able to look at our data. Take a look at the code below and try to figure out what the function call is, as well as what each argument (or thing you pass to a function) does.

```
plot(yearsSellingWidgets,numberOfSales,
     data = salesData,
     main = "My Plot",
     xlab = "Years at Company",
     ylab = "Number of Sales")
```

```
## Warning in plot.window(...): "data" is not a graphical parameter

## Warning in plot.xy(xy, type, ...): "data" is not a graphical parameter

## Warning in axis(side = side, at = at, labels = labels, ...): "data" is not
## a graphical parameter

## Warning in axis(side = side, at = at, labels = labels, ...): "data" is not
## a graphical parameter
```

```
## Warning in box(...): "data" is not a graphical parameter

## Warning in title(...): "data" is not a graphical parameter
```

If you are having a hard time understanding arguments, one thing that might help to think about is that each argument is like a click in a software program like SPSS or Excel. Imagine you want to make the same plot with this data in SSPSS, what would you do? The first thing you would do is to go to the top of the bar and find the `Plot` function and click it. This is the same as typing out `plot()` in R. Then you would have to tell that new pop up screen what two variables you want to plot and click on the related variables. Dragging and dropping those variables into your plot builder is the same as just typing out the variables you want. Lastly you want to put names on your axes and a title on your plot. The same logic would follow. We'll explore these ideas a bit more in the next section

Packages and Help

One of the beautiful aspects of programming in R is that there is wealth of other software that other people have created and shared for free that you can use. In order to use this other software beyond Base R, you need to install packages then call them using the `library()` function.

Probably the most useful package in R is the `tidyverse` which is actually a suite of packages all built around the same philosophy. We will talk more about that philosophy later, but for now, let's install the tidyverse.

To do this, uncomment (delete the hash) of the first line of R chunk below. Run that line. You can also do this by just typing that line into the R console. When you do this, R will connect to the internet, download the necessary software and add it to your library. Now even though the package is installed, it is not ready to use just yet. In order to do this, you need to use the `library()` function and pass it the name of the

library you want to use. Every time you run an R script or have R running, you need to tell R that you want this suite of tools. Best practice is putting all of the R scripts you need at the top of your R file so they are there when you need them. **You do NOT need to install the package every time you open or run R**.

```
#install.packages("tidyverse")
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------------------- tidyverse 1.2.1 --

## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ---------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

With the tidyverse loaded, you will also get a series of datasets that are used to help with the examples. Run the following commands below to see what datasets are currently available to you.

```
data()
```

We will be using the `txhousing` dataset for the rest of these examples.

Getting Help in R

You should know that programming is basically getting very good at Googling your problems. Do yourself a favor and disavow yourself of the notion that you need to "know" all the functions in order to be a good programmer. Everyone who learns R has gone though a similar process of getting stuck on stupid problems. Not related to the technical aspects of R, but there are no dumb questions when it comes to learning. Anyone that scoffs at a problem for being too simple or tells you something like "You should have just read the documentation" is just gate-keeping and you shouldn't ask them for help.

That said, there are a lot of great ways to get help when you run into problems with R.

1. Your peers
2. The internet /Stack Overflow
3. R's built in help functions

There is a bit of basic etiquette when asking for programming help. The first is that you provide enough detail to reproduce your problem and errors. The second is that you do not ask questions in a way that seems like you are just getting other people to do your work for you. Check out this post for more information on asking questions online.

One of best things to do is just open an R help page and play around with things (and break things) until you "get" how it works.

To access R's in built help function you can easier use the Help viewer in R studio or type in a question mark before the command in the console. Using two `??` will search more generally

```
?scale()
??scale()
```

Data Exploration

```
str(txhousing)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    8602 obs. of  9 variables:
##  $ city     : chr  "Abilene" "Abilene" "Abilene" "Abilene" ...
##  $ year     : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ month    : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ sales    : num  72 98 130 98 141 156 152 131 104 101 ...
##  $ volume   : num  5380000 6505000 9285000 9730000 10590000 ...
##  $ median   : num  71400 58700 58100 68600 67300 66900 73500 75000 64500 59300 ...
##  $ listings : num  701 746 784 785 794 780 742 765 771 764 ...
##  $ inventory: num  6.3 6.6 6.8 6.9 6.8 6.6 6.2 6.4 6.5 6.6 ...
##  $ date     : num  2000 2000 2000 2000 2000 ...
```

```
class(txhousing)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
summary(txhousing)
```

```
##      city                year          month            sales       
##  Length:8602        Min.   :2000   Min.   : 1.000   Min.   :   6.0  
##  Class :character   1st Qu.:2003   1st Qu.: 3.000   1st Qu.:  86.0  
##  Mode  :character   Median :2007   Median : 6.000   Median : 169.0  
##                     Mean   :2007   Mean   : 6.406   Mean   : 549.6  
##                     3rd Qu.:2011   3rd Qu.: 9.000   3rd Qu.: 467.0  
##                     Max.   :2015   Max.   :12.000   Max.   :8945.0  
##                                                     NA's   :568     
##      volume             median          listings       inventory     
##  Min.   :8.350e+05   Min.   : 50000   Min.   :    0   Min.   : 0.000  
##  1st Qu.:1.084e+07   1st Qu.:100000   1st Qu.:  682   1st Qu.: 4.900  
##  Median :2.299e+07   Median :123800   Median : 1283   Median : 6.200  
##  Mean   :1.069e+08   Mean   :128131   Mean   : 3217   Mean   : 7.175  
##  3rd Qu.:7.512e+07   3rd Qu.:150000   3rd Qu.: 2954   3rd Qu.: 8.150  
##  Max.   :2.568e+09   Max.   :304200   Max.   :43107   Max.   :55.900  
##  NA's   :568         NA's   :616      NA's   :1424    NA's   :1467    
##       date     
##  Min.   :2000  
##  1st Qu.:2004  
##  Median :2008  
##  Mean   :2008  
##  3rd Qu.:2012  
##  Max.   :2016  
##
```

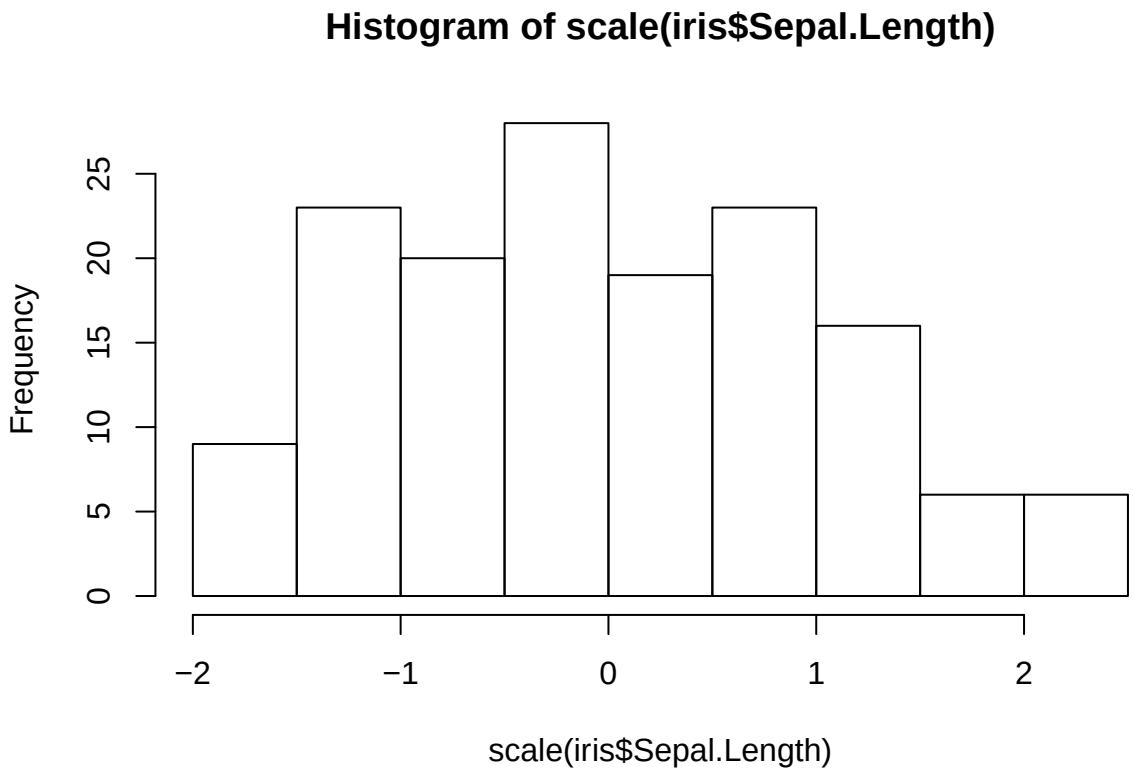
Accessing individual 'columns' is done with the $ operator

```
txhousing$sales
```

Can you use this to plot the different numeric values against each other?

What would the follow commands do?

```
hist(scale(iris$Sepal.Length))
```




```
iris$Sepal.Length.scale <- scale(iris$Sepal.Length)
```

Indexing

Let's combine logical indexing with creating new objects.

What do the follow commands do? Why?

```
txhousing[1,1]
```

```
## # A tibble: 1 x 1
##   city
##   <chr>
## 1 Abilene
```

```
txhousing[2,]
```

```
## # A tibble: 1 x 9
##   city     year month sales  volume median listings inventory  date
##   <chr>   <int> <int> <dbl>   <dbl>  <dbl>    <dbl>     <dbl> <dbl>
## 1 Abilene  2000     2    98 6505000  58700      746       6.6 2000.
```

```
txhousing[,5]
```

```
## # A tibble: 8,602 x 1
##      volume
##       <dbl>
##  1  5380000
##  2  6505000
##  3  9285000
##  4  9730000
##  5 10590000
##  6 13910000
##  7 12635000
##  8 10710000
##  9  7615000
## 10  7040000
## # ... with 8,592 more rows
```

```
txhousing[txhousing$year < 2003,]
```

```
## # A tibble: 1,656 x 9
##    city     year month sales   volume median listings inventory  date
##    <chr>   <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
##  1 Abilene  2000     1    72  5380000  71400      701       6.3 2000
##  2 Abilene  2000     2    98  6505000  58700      746       6.6 2000.
##  3 Abilene  2000     3   130  9285000  58100      784       6.8 2000.
##  4 Abilene  2000     4    98  9730000  68600      785       6.9 2000.
##  5 Abilene  2000     5   141 10590000  67300      794       6.8 2000.
##  6 Abilene  2000     6   156 13910000  66900      780       6.6 2000.
##  7 Abilene  2000     7   152 12635000  73500      742       6.2 2000.
##  8 Abilene  2000     8   131 10710000  75000      765       6.4 2001.
##  9 Abilene  2000     9   104  7615000  64500      771       6.5 2001.
## 10 Abilene  2000    10   101  7040000  59300      764       6.6 2001.
## # ... with 1,646 more rows
```

```
txhousing[,c(1:4)]
```

```
## # A tibble: 8,602 x 4
##    city     year month sales
##    <chr>   <int> <int> <dbl>
##  1 Abilene  2000     1    72
##  2 Abilene  2000     2    98
##  3 Abilene  2000     3   130
##  4 Abilene  2000     4    98
##  5 Abilene  2000     5   141
```

```
##  6 Abilene  2000     6   156
##  7 Abilene  2000     7   152
##  8 Abilene  2000     8   131
##  9 Abilene  2000     9   104
## 10 Abilene  2000    10   101
## # ... with 8,592 more rows
```

```
txhousing[txhousing$city=="San Antonio",c(1:6,8)]
```

```
## # A tibble: 187 x 7
##    city         year month sales    volume median inventory
##    <chr>       <int> <int> <dbl>     <dbl>  <dbl>     <dbl>
##  1 San Antonio  2000     1   820  98974924  90900       4.7
##  2 San Antonio  2000     2  1075 120851076  86000       4.7
##  3 San Antonio  2000     3  1433 167748201  87000       4.9
##  4 San Antonio  2000     4  1263 145280248  90200       5
##  5 San Antonio  2000     5  1574 183281564  91200       5
##  6 San Antonio  2000     6  1666 210779154 100100       5
##  7 San Antonio  2000     7  1508 185816640 100500       4.9
##  8 San Antonio  2000     8  1626 195515195  93400       5.2
##  9 San Antonio  2000     9  1300 156643797  94800       5.2
## 10 San Antonio  2000    10  1192 141630200  93500       5.2
## # ... with 177 more rows
```

```
AbilineData <- txhousing[txhousing$city == "Abilene",]
```

This could be an entire lecture by itself!!! It is important to know how R's indexing works, but in the year 2019 there is no need to be using base R command to index. We will talk more about the tidyverse tomorrow, but the following code does the exact same indexing as the base R code above, but is much more human readable.

Tidyverse

```
txhousing %>%
  select(year)
```

```
## # A tibble: 8,602 x 1
##     year
##    <int>
##  1  2000
##  2  2000
##  3  2000
##  4  2000
##  5  2000
##  6  2000
##  7  2000
##  8  2000
##  9  2000
## 10  2000
## # ... with 8,592 more rows
```

```
txhousing %>%
  filter(year < 2003)
```

```
## # A tibble: 1,656 x 9
##    city     year month sales   volume median listings inventory  date
##    <chr>   <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
##  1 Abilene  2000     1    72  5380000  71400      701       6.3 2000
##  2 Abilene  2000     2    98  6505000  58700      746       6.6 2000.
##  3 Abilene  2000     3   130  9285000  58100      784       6.8 2000.
##  4 Abilene  2000     4    98  9730000  68600      785       6.9 2000.
##  5 Abilene  2000     5   141 10590000  67300      794       6.8 2000.
##  6 Abilene  2000     6   156 13910000  66900      780       6.6 2000.
##  7 Abilene  2000     7   152 12635000  73500      742       6.2 2000.
##  8 Abilene  2000     8   131 10710000  75000      765       6.4 2001.
##  9 Abilene  2000     9   104  7615000  64500      771       6.5 2001.
## 10 Abilene  2000    10   101  7040000  59300      764       6.6 2001.
## # ... with 1,646 more rows
```

```
txhousing %>%
  select(city:volume)
```

```
## # A tibble: 8,602 x 5
##    city     year month sales   volume
##    <chr>   <int> <int> <dbl>    <dbl>
##  1 Abilene  2000     1    72  5380000
##  2 Abilene  2000     2    98  6505000
##  3 Abilene  2000     3   130  9285000
##  4 Abilene  2000     4    98  9730000
##  5 Abilene  2000     5   141 10590000
##  6 Abilene  2000     6   156 13910000
##  7 Abilene  2000     7   152 12635000
##  8 Abilene  2000     8   131 10710000
##  9 Abilene  2000     9   104  7615000
## 10 Abilene  2000    10   101  7040000
## # ... with 8,592 more rows
```

```
txhousing %>%
  select(1:6, inventory) %>%
  filter(city == "San Antonio")
```

```
## # A tibble: 187 x 7
##    city         year month sales    volume median inventory
##    <chr>       <int> <int> <dbl>     <dbl>  <dbl>     <dbl>
##  1 San Antonio  2000     1   820  98974924  90900       4.7
##  2 San Antonio  2000     2  1075 120851076  86000       4.7
##  3 San Antonio  2000     3  1433 167748201  87000       4.9
##  4 San Antonio  2000     4  1263 145280248  90200       5
##  5 San Antonio  2000     5  1574 183281564  91200       5
##  6 San Antonio  2000     6  1666 210779154 100100       5
##  7 San Antonio  2000     7  1508 185816640 100500       4.9
##  8 San Antonio  2000     8  1626 195515195  93400       5.2
##  9 San Antonio  2000     9  1300 156643797  94800       5.2
```

```
## 10 San Antonio  2000    10  1192 141630200  93500       5.2
## # ... with 177 more rows
```

```
AbilineData <- txhousing %>%
  filter(city == "Abiline")
```

As your code gets longer, the tidyverse becomes more readable. It is also more helpful for exploring data sets. More to come on this!

Saving and Importing

Finally, if we want to Import or Save other data, we can do that via the Console.

The Working Directory

Most of the work we have done this far is data that we do not want to save. Most of the work you will do after this workshop, you will want to save.

R works by pointing at a folder or directory on your computer. To see where R is pointing now, run the following code

```
getwd()
```

```
## [1] "/Users/dbaker/flatiron_curriula/r_for_python_crash/guides"
```

Whatever you **do** in your R session will happen here unless you tell it to otherwise. If you do not want R pointing in this location in your computer, you need to set your working directory elsewhere. To do this, use the `setwd()` command. This is also a good chance to use RStudio's auto complete feature.

```
setwd()
```

Open a double quotation in the function then press TAB. This will allow you to navigate your computer. Going deeper into your directory structure can be done by just following the auto complete. Going higher in the directory requires you to type `../` which will allow you to look up a level. Set your working directory to the `output` directory.

The console should now read that it is pointed to the output directory.

You can write a dataset to your working directory with the `write_csv()` command.

```
write.csv(x = AbilineData, file = "MyData.csv")
```

Importing Data

Data is imported using the same logic. You can use the `read.csv()` function to read in a csv file. At first, it might be easier to use the Import Dataset function in RStudio (Top right pane).