Module 2

Goals of module 2:
* Can have this be a 4 day module

- Learn for loops
  - Reading in all of the Redfin data
    - If statements as well
- Joining data
  - bind_rows
    - concatenating the property files
    - concatenating the location files
  - Joining
    - joining the location and property data.frames
- String manipulation
  - Cleaning some of the column names
    - splitting (separate the city from the state)
    - Cleaning (match any misspellings and reassign)
  - %in%
- Date manipulation
  - Converting dates correctly - formats
  - Doing date math (finding time on market)
    - days on market
    - could find weeks on market as well (student exercise)
- Custom functions
  - Rewriting the `sum()` function
  - Another practice instance of rewriting a function
  - Writing a custom function (to find distance from metro station to house)
- More plotting
  - Themes
  - Different scales
    - size and shape?
    - date scales
  - How to analyze what is a "good" plot
- Regression
  - Variable transformation
    - Log of price
    - price per square foot (?)
    - Interaction effects (?) - may not find any significant ones

Day One

For the past month we've looked at income data using the American Community Survey to see how incomes in California differ for different subsets of the population. This module will take a look at a different type of economic question, home prices. We will examine how proximity to a metro station impacts the sale price of homes using data collected from Redfin, a Seattle real-estate company.

Once you have downloaded all data for this module, let's set our working directory and take a look at the files that we have available.

```
invisible(library(plyr))
invisible(library(tidyverse))
```

```r
invisible(library(dplyr))

# Set your working directory to be the mod2 folder
# setwd()
list.files("Data/")
```

```
##  [1] "location_redfin_01.csv" "location_redfin_02.csv"
##  [3] "location_redfin_03.csv" "location_redfin_04.csv"
##  [5] "location_redfin_05.csv" "location_redfin_06.csv"
##  [7] "location_redfin_07.csv" "location_redfin_08.csv"
##  [9] "Metro_lat_lon.xlsx"     "property_redfin_01.csv"
## [11] "property_redfin_02.csv" "property_redfin_03.csv"
## [13] "property_redfin_04.csv" "property_redfin_05.csv"
## [15] "property_redfin_06.csv" "property_redfin_07.csv"
## [17] "property_redfin_08.csv"
```

So, in looking at our files names, we have a set of files that deal with "property," a set of files that deal with "location" data, and an excel file that doesn't match either pattern, "Metro_lat_lon.xlsx".

Let's read in one of our property files.

```r
property_data <- read.csv("Data/property_redfin_01.csv")
names(property_data)
```

```
##  [1] "SALE.TYPE"                  "SOLD.DATE"
##  [3] "PROPERTY.TYPE"              "PRICE"
##  [5] "BEDS"                       "BATHS"
##  [7] "SQUARE.FEET"                "LOT.SIZE"
##  [9] "YEAR.BUILT"                 "HOA.MONTH"
## [11] "STATUS"                     "NEXT.OPEN.HOUSE.START.TIME"
## [13] "NEXT.OPEN.HOUSE.END.TIME"   "URL"
## [15] "SOURCE"                     "MLS."
## [17] "FAVORITE"                   "INTERESTED"
## [19] "LIST.DATE"
```

This file seems to be about home sales and includes variables about how large the house is, when it was sold, when it was put on the market, etc...

We can see the data using the `View()` function.

```r
#View(property_data)
```

Read in one of our location files, what are the names of the columns in the file? What does this data look like? Are there any columns that exist in both our property and location data?

```r
# location_data <- read.csv()
```

Maybe these files belong somehow together. To check this we should read in another one of our property files, does it have the same columns as our other property data file?

```r
property_data2 <- read.csv("Data/property_redfin_02.csv")
names(property_data)
```

```
##  [1] "SALE.TYPE"                  "SOLD.DATE"
##  [3] "PROPERTY.TYPE"              "PRICE"
##  [5] "BEDS"                       "BATHS"
##  [7] "SQUARE.FEET"                "LOT.SIZE"
##  [9] "YEAR.BUILT"                 "HOA.MONTH"
## [11] "STATUS"                     "NEXT.OPEN.HOUSE.START.TIME"
```

```
## [13] "NEXT.OPEN.HOUSE.END.TIME"   "URL"
## [15] "SOURCE"                     "MLS."
## [17] "FAVORITE"                   "INTERESTED"
## [19] "LIST.DATE"
```

```
identical(names(property_data2), names(property_data))
```

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

This is promising, our files have the same columns, this means we can stick one of top of the other.

```
nrow(property_data)
```

```
## [1] 149
```

```
nrow(property_data2)
```

```
## [1] 100
```

```
property_data_bound <- bind_rows(property_data, property_data2)
nrow(property_data_bound) # Sum of the rows of the two tables
```

```
## [1] 249
```

Are our location files also the same? Read in a second location file and test if it has the same columns as the first one. If it does, create a new table, location_data_bound which has the data for both tables in it.

```
#location_data2 <- read.csv()

#location_data_bound <-
```

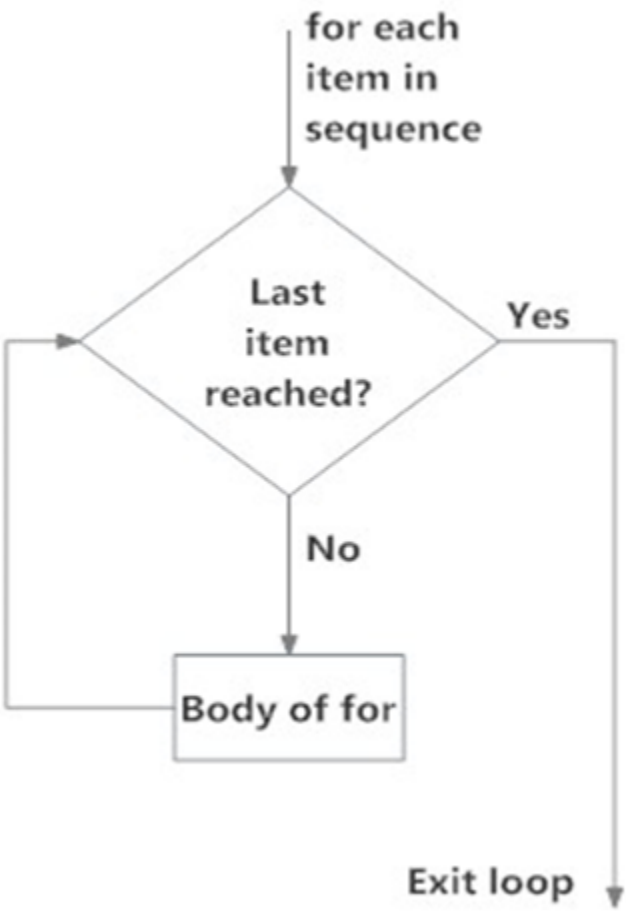
Now we need to figure out a way to bind our files together so that we can go from 8 property and 8 location data frames into a single property data frame and single location data frame.

This is the perfect opportunity to use a for loop!

for loops

Oftentimes when we are preparing data for analysis, we will have to repeat a task or calculation multiple times. If we only have to do this four or five times copy and pasting or retyping our code may not be too inconvenient, but what if we had to run a similar chunk of code 20 or even 1,000 times? Loops are constructs that allow us to automate these kinds of repetitive tasks, saving us time and lines of code.

If we know how many times we would like to run a particular chunk of code or we have a list we would like to iterate over we may utilize a for loop.



For loops have three key components:
* a variable that is used within the code of the loop * a list of numbers or elements that the loop iterates over * the code that is to be executed in the loop

In R, for loops take the following form:

for(*variable* in *index*) { *code to be executed* }

Let's create a simple loop that squares every number from 1 to 10 and printing the results to the console.

```
for(i in 1:10){
    print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Pretty simple! For loops can iterate over many different types of lists. For example, let's try printing the

names of all of the months in the year.

```
months <- c("January","February", "March", "April", "May", "June", "July", "August", "September", "Octo

for(mon in months){
  print(mon)
}
```

```
## [1] "January"
## [1] "February"
## [1] "March"
## [1] "April"
## [1] "May"
## [1] "June"
## [1] "July"
## [1] "August"
## [1] "September"
## [1] "October"
## [1] "November"
## [1] "December"
```

Not too bad! Ideally, we'll be using these loops to do more than just print out results. We can create loops that save and update objects without overwriting loops if we are careful. The Fibonacci sequence is a series of numbers where any number can be found by adding the previous two numbers together. We can see the Fibonacci sequence (or more specifically, the Fibonacci spiral) in hurricanes, galaxies, and even Rennaisance art! Below we have a function that calculates 15 numbers in the Fibonacci sequence.

```
fibSeq <- c(0, 1)
print(fibSeq)
```

```
## [1] 0 1
```

```
for(i in 3:15) {
 fibNext <- fibSeq[i - 1] + fibSeq[i - 2]
 fibSeq <- c(fibSeq, fibNext)
 print(fibSeq)
}
```

```
## [1] 0 1 1
## [1] 0 1 1 2
## [1] 0 1 1 2 3
## [1] 0 1 1 2 3 5
## [1] 0 1 1 2 3 5 8
## [1]  0  1  1  2  3  5  8 13
## [1]  0  1  1  2  3  5  8 13 21
##  [1]  0  1  1  2  3  5  8 13 21 34
##  [1]  0  1  1  2  3  5  8 13 21 34 55
##  [1]  0  1  1  2  3  5  8 13 21 34 55 89
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233
##  [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

Notice how we initialize the vector first and then once we are inside the loop, we calculate the next number then add it to the vector. Our helper variable, `fibNext`, allows us to calculate the next number in the series without overwriting the vector.

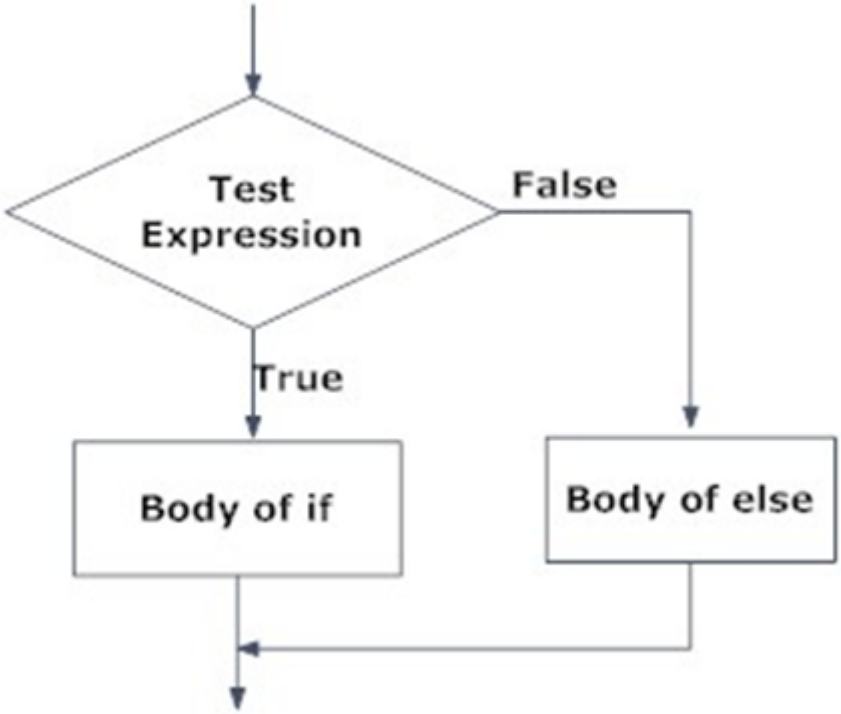
Let's try converting a bunch of substrings in a vector into a single string using a for loop.

```
string_vec <- c("Never","Gonna", "Give", "You", "Up", "Never", "Gonna", "Let", "You", "Down")

## initialize the variable
rr = NULL

for(sv in string_vec){
  rr = paste(rr,sv)
}

print(rr)
```

```
## [1] " Never Gonna Give You Up Never Gonna Let You Down"
```

However, our current loops are less than ideal. Sometimes we would like to perform one action on certain elements of our list and a separate action on the other elements. To save time and lines of code, we want our code to make these kinds simple decisions without our direction. Enter if/else statements.

If Else Statements



Previously we have used `ifelse()` to help make simple decisions when recoding our variables with `mutate`. If/else statements are constructs that function in a similar manner to `ifelse()` and are applied to segments of code *outside* of functions. In R, an if/else statement takes the following form:

if(*logical argument*){ *code to be executed* } else{ *code to be executed* }

Instead of printing out the square of all of the numbers from 1 to 10, why don't we try printing out whether each of these numbers are even or odd.

```
for(i in 1:10){
  if(i %% 2 == 0){ # %% is the modulous operator --- we are finding the remainder!
    print(paste(i, "is EVEN"))
```

```
  } else{
    print(paste(i, "is ODD"))
  }
}
```

```
## [1] "1 is ODD"
## [1] "2 is EVEN"
## [1] "3 is ODD"
## [1] "4 is EVEN"
## [1] "5 is ODD"
## [1] "6 is EVEN"
## [1] "7 is ODD"
## [1] "8 is EVEN"
## [1] "9 is ODD"
## [1] "10 is EVEN"
```

Now our code can be much more flexible! We can program our code to have even greater flexibility by chaining together if/else statements to test multiple conditions. Suppose we have a list of test scores that we want to assign letter grades.

```
test_scores <- c(85,55,67,73,92,94,99,100,87)

for(grade in test_scores){
  if(grade >= 90){
    print("A")
  } else if(grade >= 80) {
    print("B")
  } else if (grade >= 70) {
    print("C")
  } else if (grade >= 60) {
    print("D")
  } else {
    print("F")
  }
}
```

```
## [1] "B"
## [1] "F"
## [1] "D"
## [1] "C"
## [1] "A"
## [1] "A"
## [1] "A"
## [1] "A"
## [1] "B"
```

Let's try creating a loop that will take the square root of a positive number, but print a message if there is a negative message

```
num_list <- c(4,169,-9,9,-144,0)

#Answer here
```

Our loop checks each condition one by one, and if that condition is met it moves on to the next one until we reach the final condition. For example, 85 is not greater than 90, so then our program checks if 85 is greater than 80. It turns out that 85 *is* greater than 80, therefore our program assigns this student a B. If we were

working within `mutate()`, `casewhen()` would be the ideal choice for handling mutliple decisions.

Reading in the Redfin Data

Now that we understand what a for loop is and how to use if/else statements to enable our programs to make decisions for us, let's think about how we can use it for our analysis. First we need the names of all of the property and location files. We can use a nice R trick to create those names.

```
 paste("test_0", c(1:8), sep = "")
```

```
## [1] "test_01" "test_02" "test_03" "test_04" "test_05" "test_06" "test_07"
## [8] "test_08"
```

```
property_files <- paste("Data/property_redfin_0", c(1:8), ".csv", sep = "")
location_files <- paste("Data/location_redfin_0", c(1:8), ".csv", sep = "")

files <- c(property_files, location_files)
files
```

```
##  [1] "Data/property_redfin_01.csv" "Data/property_redfin_02.csv"
##  [3] "Data/property_redfin_03.csv" "Data/property_redfin_04.csv"
##  [5] "Data/property_redfin_05.csv" "Data/property_redfin_06.csv"
##  [7] "Data/property_redfin_07.csv" "Data/property_redfin_08.csv"
##  [9] "Data/location_redfin_01.csv" "Data/location_redfin_02.csv"
## [11] "Data/location_redfin_03.csv" "Data/location_redfin_04.csv"
## [13] "Data/location_redfin_05.csv" "Data/location_redfin_06.csv"
## [15] "Data/location_redfin_07.csv" "Data/location_redfin_08.csv"
```

Now that we have our file names, our goal is to use a for loop to go through each file, decide if it is a property or location file and then read in the data and bind it with the previous data frame.

To do that we need to know if our file is a property file or a location file. Luckily for us, the tidyverse has functions that can look at character strings and tell us if the string contains a certain word.

str_detect is a simple but powerful function. It takes a string (or a vector of strings), and a pattern and tells you TRUE if your string contains the pattern and FALSE if your string does not contain the pattern.

```
str_detect("property_redfin_01.csv", "property") # TRUE
```

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

```
str_detect("location_redfin_01.csv", "property") # FALSE
```

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

```
str_detect(c("property_redfin_01.csv", "location_redfin_01.csv"),
           "location") # c(FALSE, TRUE)
```

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

Your turn, using our vector of files from above, along with str_detect, create a vector that only has our location files

Now that we've learned how we can use str_detect to read strings, we are ready to put it all together and read in all of our data.

Let's create a loop that reads in all of the property and location data into two seperate data sets.

```
property_data <- data.frame() # initialize empty data.frame
location_data <- data.frame() # initialize empty data.frame

# put your loop here
```

```
## [1] 1190    6
```

```
## [1] 1190   19
```

Day Two

Last week we discussed for loops and conditional execution (if statements), as a means to reading in and combining our property and location data using bind_rows(). Today we will look at how we can combine our property and location data and then we will look at our data to make sure that it is free of errors and issues before we use it for our regressions.

Our overarching question is how proximity to a metro station impacts the sale price of a house.

What variables would we need to have in order to answer this question?

Take a look at the variables we have in our property and location data, we already have much of this data, or the ability to calculate these variables.

```
names(property_data)
```

```
##  [1] "SALE.TYPE"                  "SOLD.DATE"
##  [3] "PROPERTY.TYPE"              "PRICE"
##  [5] "BEDS"                       "BATHS"
##  [7] "SQUARE.FEET"                "LOT.SIZE"
##  [9] "YEAR.BUILT"                 "HOA.MONTH"
## [11] "STATUS"                     "NEXT.OPEN.HOUSE.START.TIME"
## [13] "NEXT.OPEN.HOUSE.END.TIME"   "URL"
## [15] "SOURCE"                     "MLS."
## [17] "FAVORITE"                   "INTERESTED"
## [19] "LIST.DATE"
```

```
names(location_data)
```

```
## [1] "URL"        "ADDRESS"    "ZIP_CODE"   "LAT_LON"    "CITY_STATE"
## [6] "LOCATION"
```

The good news is that we have much of the data we need, the bad news is that we have one file with location data (how close the house is to a metro station), while in the other file we have the sale price of the house. We need to combine these two data.frames.

"Joining" the data

Combining data frames is called "joining." There are multiple types of joins as you can see in the dplyr cheat sheet:

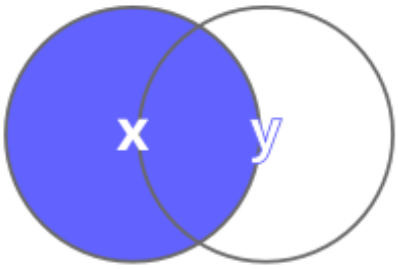
Help -> cheat sheets -> Data transformation with dplyr.

Additionally, the below graphic shows all the different join types.

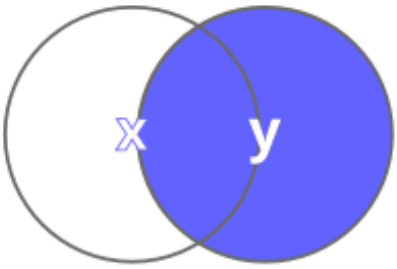
```
knitr::include_graphics("~/howard_class/mod2/joins.png")
```

dplyr *joins*

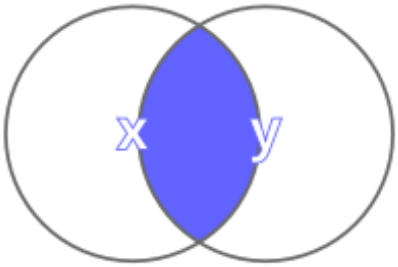
left_join(x, y)



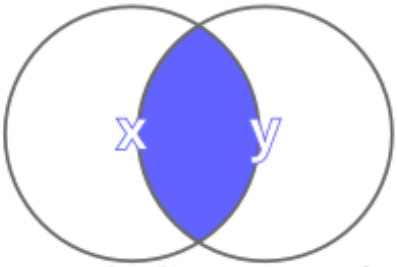
right_join(x, y)



inner_join(x, y)

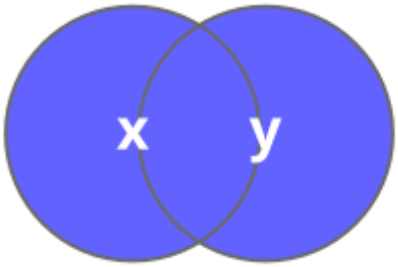


semi_join(x, y)

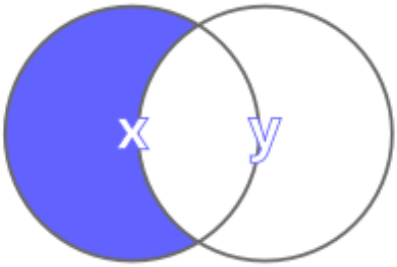


(never duplicate rows of x)

full_join(x, y)



anti_join(x, y)



In order to join to data frames, they must have at least one column of the same variable. For us we want to figure out what that column name is, then we can tell our join function what column to join our data on.

We can do this with a handy new operator, %in%, which tells us what elements of our object on the left are contained in the elements of our object on the right.

```
3 %in% c(1, 2, 3, 4)
```

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

```
3 %in% c(1, 2, 4)
```

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

```
c(1, 2, 3, 4) %in% c(1, 7, 8, 9)
```

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

```
vec1 <- c(1, 5, 8, 10)
vec2 <- c(4, 5, 19, 10)
vec1 %in% vec2
```

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

```
which(vec1 %in% vec2) # position in vec1 with overlapping elements
```

```
## [1] 2 4
```

```
vec1[vec1 %in% vec2] # we can even get back those elements that overlap
```

```
## [1]  5 10
```

```
property_names <- names(property_data)
location_names <- names(location_data)

property_names %in% location_names
```

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

```
overlap <- property_names[property_names %in% location_names]
overlap
```

```
## [1] "URL"
```

Ok, we got it, our URL column exists in both of the tables! Since we need every single property to have location data for our analysis, we are only interested in properties where the join is successful, so we want a full join.

```
joined_data <- full_join(property_data,
                         location_data,
                         by = overlap)

names(joined_data)
```

```
##  [1] "SALE.TYPE"                 "SOLD.DATE"
##  [3] "PROPERTY.TYPE"             "PRICE"
##  [5] "BEDS"                      "BATHS"
##  [7] "SQUARE.FEET"               "LOT.SIZE"
##  [9] "YEAR.BUILT"                "HOA.MONTH"
## [11] "STATUS"                    "NEXT.OPEN.HOUSE.START.TIME"
## [13] "NEXT.OPEN.HOUSE.END.TIME"  "URL"
```

```
## [15] "SOURCE"                    "MLS."
## [17] "FAVORITE"                  "INTERESTED"
## [19] "LIST.DATE"                 "ADDRESS"
## [21] "ZIP_CODE"                  "LAT_LON"
## [23] "CITY_STATE"                "LOCATION"
```

Success!

Note: It is also possible to perform a join where the columns we are matching in each table do not have the same name using the by argument: by = c("name1" = "name2").

Similarly, it is also possible to join on multiple columns: by = c(column1, column2, etc...)

We did it, we got our housing data all put together, phew!

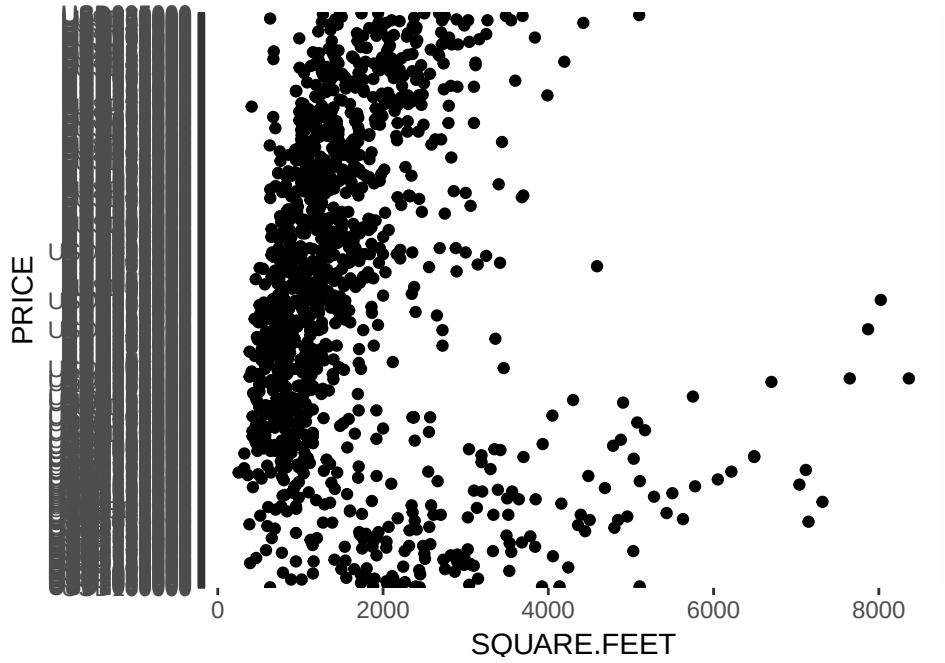
Now that it is all together, it is time to take a look at it and see if we can spot any problems.

Take a look at the data with View(). What are some problems you see in terms of usability for this data?

Clearly we have our work cut out for us!

Let's first make a scatter-plot looking at our price vs square feet variables.

```
joined_data %>%
    ggplot(aes(x = SQUARE.FEET, y = PRICE)) +
    geom_point()
```



Hmm, something seems off with our PRICE variable, let's take a look at it.

```
head(joined_data$PRICE)
```

```
## [1] "USD280000" "USD405000" "USD510000" "USD395000" "USD339900" "USD415000"
```

Oh no, we have a USD in front of each price number, this means that instead of numeric data our prices are character data. We need a way to remove the USD.

One way we could do this is by replacing the USD with nothing, "". This would have the same effect as removing it.

```
str_replace("USD40000", "USD", "")
```

```
## [1] "40000"
```

Luckily we can now use mutate to fix our whole column.

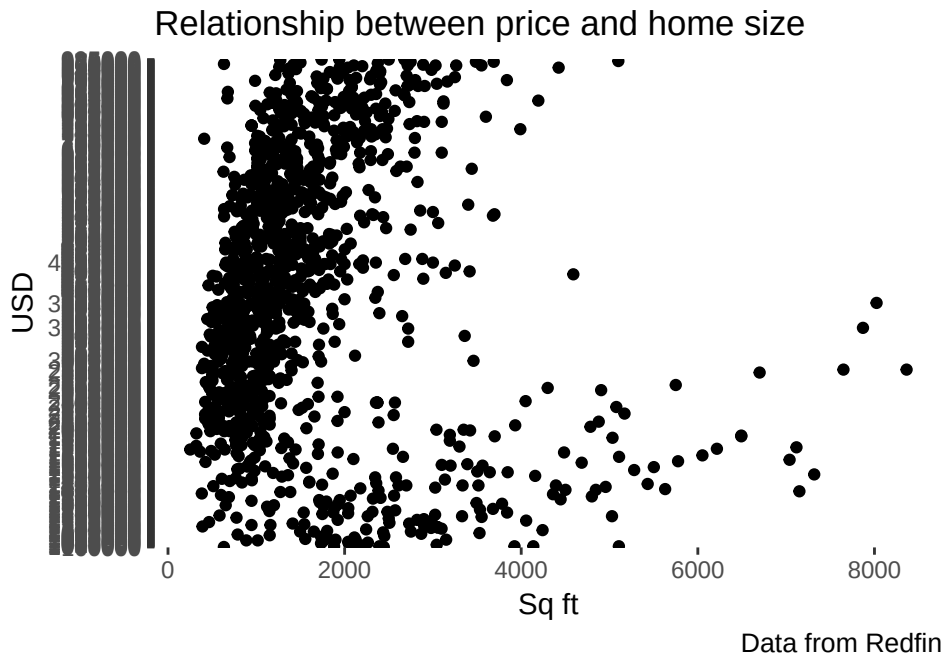
```
joined_data <- joined_data %>% mutate(PRICE = str_replace(PRICE, "USD", ""))
```

We just used str_replace to fix our PRICE column. Take a look at our PROPERTY.TYPE column. Using str_replace, fix this column as well.

```
# Answer Here
```

Now that our price column is fixed we can make our plot again.

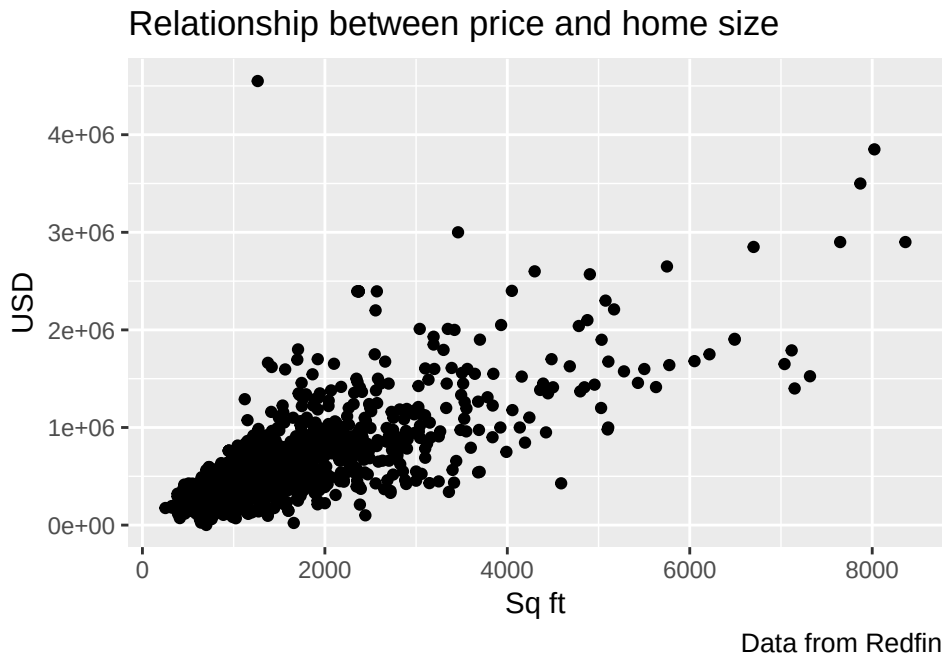
```
joined_data %>%
    ggplot(aes(x = SQUARE.FEET, y = PRICE)) +
    geom_point() +
    labs(title = "Relationship between price and home size",
         y = "USD",
         x = "Sq ft",
         caption = "Data from Redfin")
```



We still have the same problem, we forgot to convert our PRICE column to be numeric!

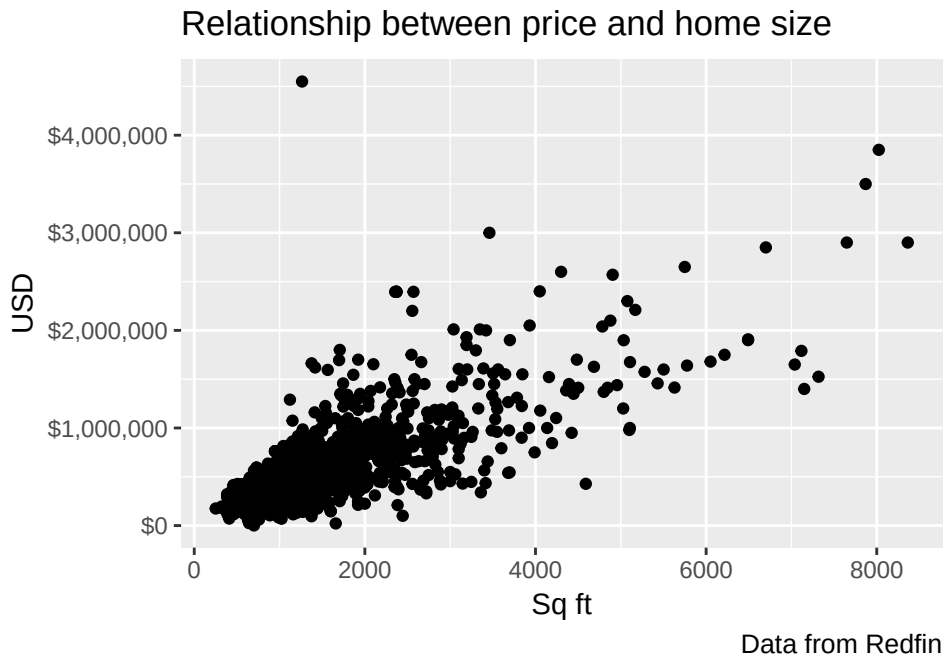
```
joined_data <- joined_data %>%
    filter(!is.na(PRICE)) %>%
    mutate(PRICE = as.numeric(PRICE))

# Now make the plot
joined_data %>%
    ggplot(aes(x = SQUARE.FEET, y = PRICE)) +
    geom_point() +
    labs(title = "Relationship between price and home size",
         y = "USD",
         x = "Sq ft",
```

```
        caption = "Data from Redfin")
```



Hmm, I don't like this y-axis, I want the labels to be formatted with dollar signs so that it is clear what is going on here. Luckily we can do this easily with the scales package.

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

joined_data %>%
    ggplot(aes(x = SQUARE.FEET, y = PRICE)) +
    geom_point() +
    scale_y_continuous("USD", labels = dollar) +
    labs(title = "Relationship between price and home size",
         x = "Sq ft",
         caption = "Data from Redfin")
```



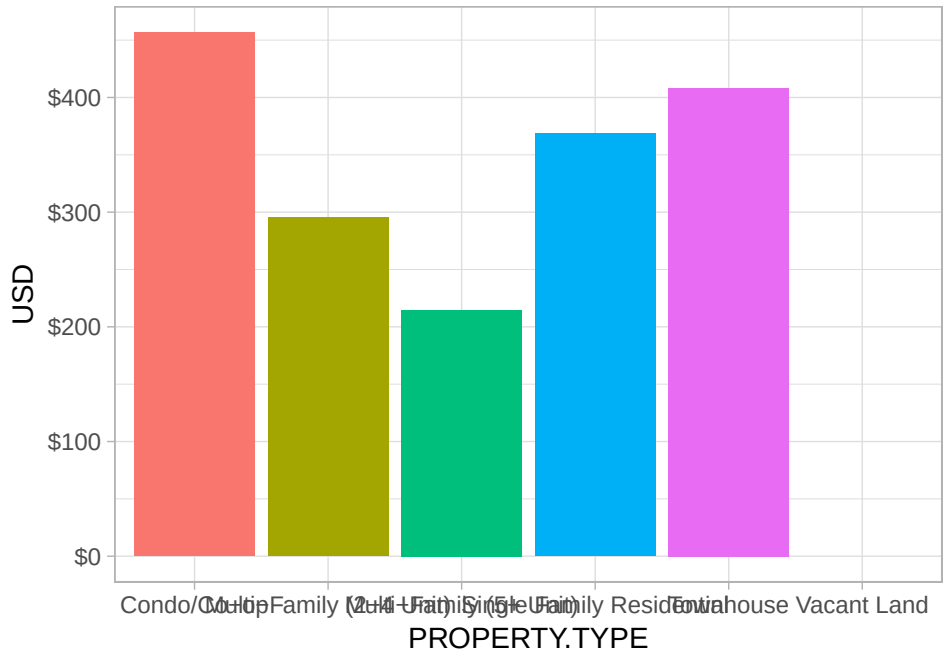
How would you describe the relationship between price and size of a home?

So, we see that there is definitely a relationship between size of a home and price of a home. We would probably also expect to see a relationship between the property type and its price.

```
type_price <- joined_data %>%
    group_by(PROPERTY.TYPE) %>%
    summarize(price = mean(PRICE/SQUARE.FEET, na.rm = T))

type_price
```

```
## # A tibble: 6 x 2
##               PROPERTY.TYPE    price
##                       <chr>    <dbl>
## 1               Condo/Co-op 456.6841
## 2   Multi-Family (2-4 Unit) 295.3326
## 3    Multi-Family (5+ Unit) 214.7378
## 4 Single Family Residential 368.6735
## 5                 Townhouse 408.2464
## 6               Vacant Land      NaN
```

```
type_price %>%
            ggplot(aes(x = PROPERTY.TYPE,
                       y = price,
                       fill = PROPERTY.TYPE)) +
            geom_bar(stat = "identity") +
            scale_y_continuous("USD", labels = dollar) +
            scale_fill_discrete(guide = F) + # get rid of the fill legend
            theme_light() # Change the overall aesthetic
```



Our plot here clearly needs some work. Luckily we aren't planning on showing this to anyone, just looking at it for ourselves to see if we can come to some quick conclusions.

What changes did adding theme_light() make to the plot?

We've now seen that prices seem to show a relationship to square footage and that there is some variance is price per square foot by property type.

Another variable we should think about is the city and state of the house's location. Taxes in DC, Maryland, and Virginia are different, and these may have an impact on a home's sale price.

Let's take a look at all of our different possibilities for the CITY_STATE variable.

```
unique(joined_data$CITY_STATE)
```

```
##  [1] "Arlington, va"         "Alexandria, va"
##  [3] "Arlington, VA"         "Falls Church, va"
##  [5] "Arlington, Virginia"   "Mclean, Virginia"
##  [7] "Chevy Chase, MD"       "Bethesda, MD"
##  [9] "Glen Echo, MD"         "Kensington, MD"
## [11] "Washington, DC"        "Washington, Michigan"
## [13] "Oxon Hill, MD"         "Silver Spring, MD"
## [15] "Washington, COLORADO"  "Capitol Heights, MD"
## [17] "Takoma Park, MD"       "Alexandria, VA"
## [19] "Fairmount Heights, MD" "Fulton, MD"
```

It seems that our formatting is consistent in one regard, we have "city, state" but it is not consistent in how our states are given. We have va as well as VA, and somehow we got data for Michigan and Colorado in there!

First we will need to break up this column into a city column and a state column, then we can look at states individually. Luckily we have a function which can help us, str_split_fixed.

str_split_fixed takes a string (or vector of strings), and a pattern. The functions returns the pieces of the string after splitting on your pattern. Let me show you.

```
strings <- str_split_fixed("United States of America", "of", n = 2)
strings
```

```
##      [,1]             [,2]
## [1,] "United States " " America"
```

We told R to split the string into 2 pieces (n = 2), so we got a matrix back with two columns, one for each of our new strings.

We can subset this to get back our elements.

```
strings[, 1] # first row, first column
```

```
## [1] "United States "
```

```
strings[, 2] # first row, second column
```

```
## [1] " America"
```

Let's test it out with something closer to our actual use case.

```
str_split_fixed("Arlington, Virginia", ",", n = 2)
```

```
##      [,1]        [,2]
## [1,] "Arlington" " Virginia"
```

Great, let's use this with mutate now and create a city and a state column.

```
joined_data <- joined_data %>%
    mutate(CITY = str_split_fixed(CITY_STATE, ",", n = 2)[,1],
           STATE = str_split_fixed(CITY_STATE, ",", n = 2)[, 2])

joined_data %>% select(CITY_STATE, CITY, STATE) %>% head
```

```
##      CITY_STATE      CITY STATE
## 1 Arlington, va Arlington    va
## 2 Arlington, va Arlington    va
## 3 Arlington, va Arlington    va
## 4 Arlington, va Arlington    va
## 5 Arlington, va Arlington    va
## 6 Arlington, va Arlington    va
```

```
head(joined_data$STATE)
```

```
## [1] " va" " va" " va" " va" " va" " va"
```

So we have successfully created a state column and a city column. We have one problem left, we now have a space in front of our state name. str_trim can be used to fix this problem.

```
str_trim(" word ") # It removes the blank space!
```

```
## [1] "word"
```

```
joined_data <- joined_data %>%
    mutate(STATE = str_trim(STATE))

head(joined_data$STATE)
```

```
## [1] "va" "va" "va" "va" "va" "va"
```

Just as our CITY_STATE column had to be split up, so too does our LAT_LON column which needs to be split into a LATITUDE and a LONGITUDE column. Right now our data is separated by a & character. Using the same method as above, split our LAT_LON column.

Now that we've created our STATE column we still have some issues, we have "va" as well as "VA" and "Virginia."

```
unique(joined_data$STATE)
```

```
## [1] "va"       "VA"       "Virginia" "MD"       "DC"       "Michigan"
## [7] "COLORADO"
```

The first thing we want to do is turn our lower case "va" to upper case VA. We can do that with the appropriately named str_to_upper function.

```
str_to_upper("va")
```

```
## [1] "VA"
```

```
joined_data <- joined_data %>%
    mutate(STATE = str_to_upper(STATE))

unique(joined_data$STATE)
```

```
## [1] "VA"       "VIRGINIA" "MD"       "DC"       "MICHIGAN" "COLORADO"
```

Now we need to turn VIRGINIA into VA, using a function we've already seen today, make that change.

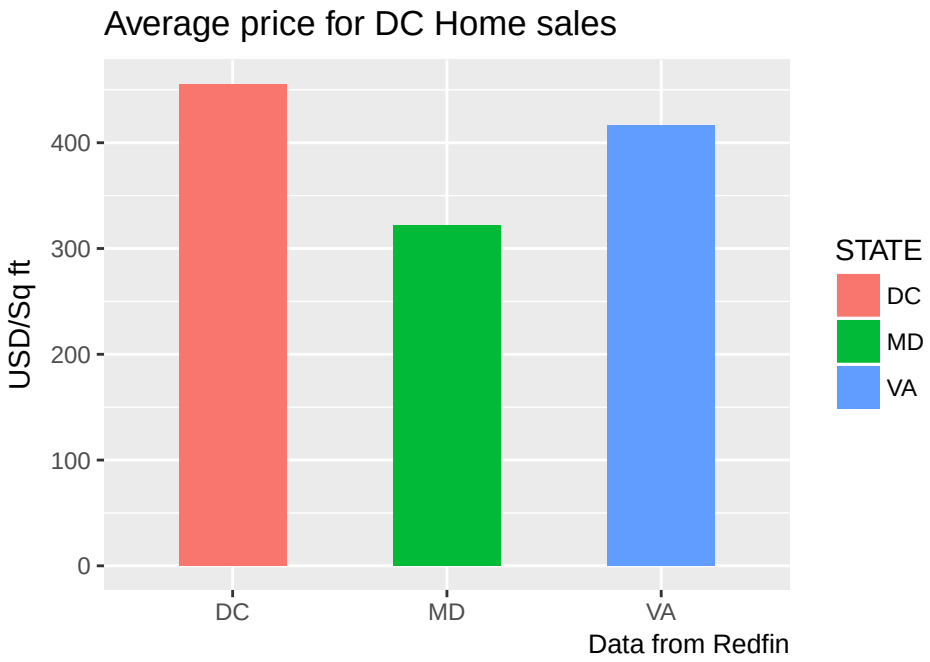
We're almost done cleaning up our STATE data column. We just have to throw out rows from MICHIGAN and COLORADO since we are only interested in home sales that will fall along the DC metro. We can just use filter to keep the rows we want.

```
joined_data <- joined_data %>%
    filter(STATE %in% c("VA", "DC", "MD"))

unique(joined_data$STATE)
```

```
## [1] "VA" "MD" "DC"
```

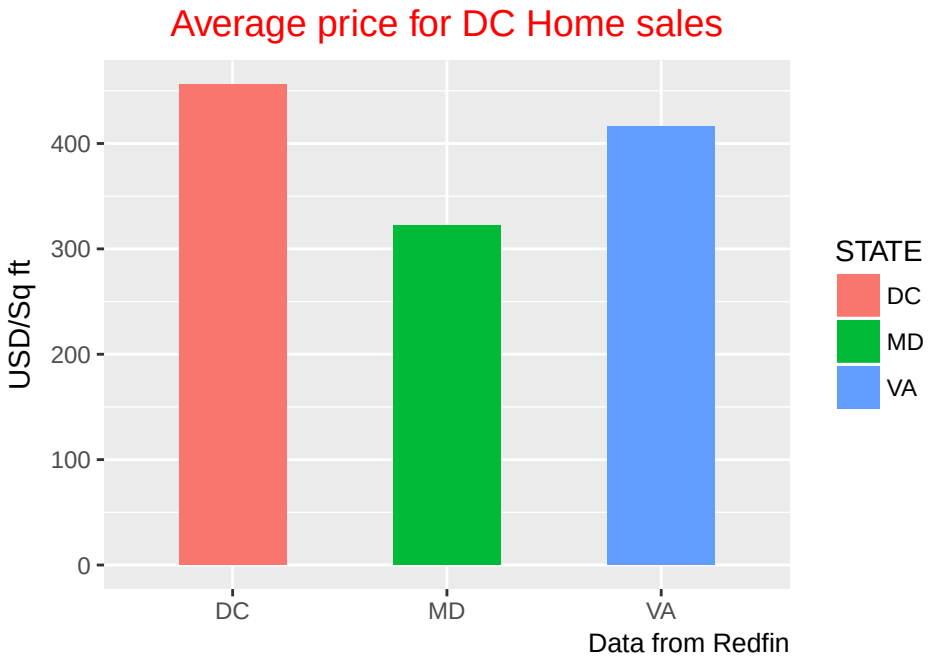
Now that our data is clean we can find out state average prices.

```
state_average <- joined_data %>% group_by(STATE) %>%
    summarize(price = mean(PRICE/SQUARE.FEET, na.rm = T))

state_average_plot <- state_average %>%
    ggplot(aes(x = STATE, y = price, fill = STATE)) +
    geom_bar(stat = "identity", width = 0.5) +
    labs(x = NULL, # What does this do?
         y = "USD/Sq ft",
         title = "Average price for DC Home sales",
         caption = "Data from Redfin")

state_average_plot
```



I think that this would look better if my plot title were larger sized font and centered. Remember previously how we discussed plot elements as being divided into data related dimensions (controlled by the scale functions) and purely aesthetic options? We can change our aesthetic options using the theme() function.

```
state_average_plot +
    theme(plot.title = element_text(size = 14, hjust = 0.5, color = "Red"))
```

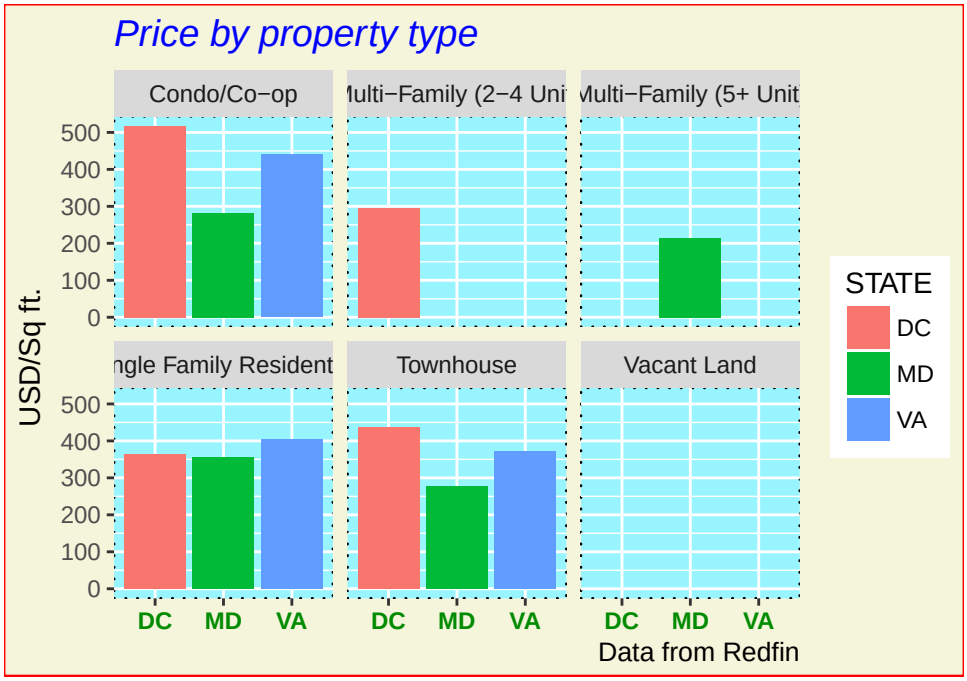


Since our title is a character string, or text, we use element_text to control it. element_text allows us to set:
* font family (Times New Roman, Arial, etc...) * font size (10, 12, 14, etc...)
* font face (bold, italic) * color * hjust (horizontal adjustment) * vertical adjustment * angle
* other aspects that impact the display of text.

What are some other text elements that you see on our plot?

Previously we saw that price per square foot differs by the property type, let's see if that holds true across states as well.

```
state_proptype <- joined_data %>%
    group_by(PROPERTY.TYPE, STATE) %>%
    summarise(price = mean(PRICE/SQUARE.FEET, na.rm = T)) # empty lots

state_proptype %>%
    ggplot(aes(x = STATE, y = price, fill = STATE)) +
    geom_bar(stat = "identity") +
    facet_wrap("PROPERTY.TYPE") +
    labs(title = "Price by property type",
         x = NULL,
         y = "USD/Sq ft.",
         caption = "Data from Redfin") +
    theme(plot.title = element_text(face = "italic", size = 14, color = "Blue"),
          axis.text.x = element_text(face = "bold", color = "green4"),
          plot.background = element_rect(fill = "beige", color = "red"),
          panel.background = element_rect(fill = "cadetblue1",
                                          color = "black", linetype = 3))
```



Just as we could use element_text to control the text elements of our plot, so too can we use element_rect to control the rectangular elements of our plot. The main rectangle of our plot is the background which we changed to "beige" with a "red" border. Additionally, we can change the background color for the area where the data is displayed by use of panel.background.

What did the linetype argument in panel.background do? What other rectangles do you see that we could adjust?

Perhaps looking at the price of the home per square foot of the home is not the best way to do it. Many of the sales include additional land on which the house is situated.

Let's take a look at making a plot of price/lot size

```
joined_data %>% select(PRICE, SQUARE.FEET, LOT.SIZE) %>% head()
```

```
##    PRICE SQUARE.FEET LOT.SIZE
## 1 280000        1055       NA
## 2 405000        1030       NA
## 3 510000        1209       NA
## 4 395000        1135       NA
## 5 339900         930       NA
## 6 415000        1606       NA
```

So we seem to have a lot of NA values, these are probably condo units that are in a building so not on a lot. Let's update our data so that if the lot size is NA, we get the square feet of the unit itself.

```
joined_data <- joined_data %>%
    mutate(LOT.SIZE = ifelse(is.na(LOT.SIZE), SQUARE.FEET, LOT.SIZE))

# Now we can find our average
lot_state_data <- joined_data %>%
    group_by(PROPERTY.TYPE, STATE) %>%
    summarize(price = mean(PRICE/LOT.SIZE, na.rm = T))

head(lot_state_data)
```

```
## # A tibble: 6 x 3
## # Groups:   PROPERTY.TYPE [4]
##                PROPERTY.TYPE STATE    price
##                        <chr> <chr>    <dbl>
## 1               Condo/Co-op    DC 517.2998
## 2               Condo/Co-op    MD 281.4152
## 3               Condo/Co-op    VA 439.9952
## 4   Multi-Family (2-4 Unit)    DC 338.9963
## 5    Multi-Family (5+ Unit)    MD 179.8902
## 6 Single Family Residential    DC 166.2042
```

Now that our data is prepared make a plot like the one above looking at the average price per square foot by each state and property type.

- **Make your title centered and Green**
- **Make your panel background White with a red border**

Day Three

Last week we successfully joined all of our data into a single data frame and cleaned up many of our columns using string cleaning functions like str_split_fixed and str_replace. We have one last column to clean before our data will be ready for our next step.

Remember, our goal is to find the impact of the distance of a house from a metro station on the price of the house.

Cleaning zip code data

```
head(joined_data$ZIP_CODE)
```

```
## [1] "'222040000" "'222040000" "'222020000" "'222040000" "'222060000"
## [6] "'222020000"
```

Someone added a bunch of zeroes to our zip code data and put a quote mark in the beginning. We need to pull out the valid zip codes from these numbers. We can do this by combining two new str functions, str_sub and str_length.

str_sub takes a text string, a start position and an end position and returns the part of the text string between the start and end.

```
str_sub("Hello", start = 2, end = 4)
```

```
## [1] "ell"
```

It works on vector as well like all of our other string functions.

The other function we need, str_length, tells you how many characters there are in a string.

```
str_length("Hello")
```

```
## [1] 5
```

```
str_length("'222040000")
```

```
## [1] 10
```

```
str_sub("'222040000", 2, str_length("'222040000") - 4)
```

```
## [1] "22204"
```

Now we can use mutate to fix our zip code data.

```
joined_data <- joined_data %>%
    mutate(ZIP_CODE = str_sub(ZIP_CODE, 2,
                                          str_length(ZIP_CODE) - 4))

head(joined_data$ZIP_CODE)
```

```
## [1] "22204" "22204" "22202" "22204" "22206" "22202"
```

Let's see how housing prices vary by zip code, this will give us a clue about what effects the neighborhood in which the house is located are.

```
zip_data <- joined_data %>%
    group_by(ZIP_CODE) %>%
    summarize(price = mean(PRICE/SQUARE.FEET, na.rm = T))

range(zip_data$price, na.rm = T)
```

```
## [1]  90.54545 682.77778
```

So we see pretty big differences in price per square foot between zip codes, exactly as we would expect. We will have to take this into account in our later regressions.

Now that we have successfully cleaned up our housing data, it is time to take a look at our metro station data which we will use to calculate the distance of each property to the nearest metro station. Our metro station location data is stored in an xlsx file, so we will not be able to use read_csv to get the data.

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

metro_data <- read.xlsx("Data/Metro_lat_lon.xlsx", sheetIndex = 1)
```

```
names(metro_data)
```

```
## [1] "station"   "address1"  "address2"  "Latitude"  "Longitude"
```

Now we have a latitude and longitude for each metro station and a latitude and longitude for each property. We can use them to calculate the distance. Unfortunately this is actually a pretty difficult calculation and we are going to have to write our own function to solve this problem.

Before we can do that though we will have to discuss how to do this.

Review: What is a function

For us, a function is a piece of code that takes one or more inputs and returns one or more outputs. For example, the min() function takes a vector of numbers and returns the number with the lowest value.

Let's talk about how to write one.

In R a function is defined as follows:

function_name <- function(*arguments*) {
Code that does your task
}

Here is a very simple function for adding two numbers together.

```
add <- function(number1, number2){
    number1 + number2
}

add(2, 5)
```

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

Write a new function, "subtract" which takes two numbers as arguments and returns the first number minus the second number.

We can do something similar with exponents.

```
exponent <- function(base, power){
    base**power
}

exponent(base = 2, power = 8)
```

```
## [1] 256
```

We are able to use any code that we want inside a function. For example, we can use for loops. Let's write a function that takes a vector of numbers and returns the sum of that vector.

```
vector_sum <- function(numeric_vector){

    sum <- 0
    for(number in numeric_vector){
        # Use for loop to add up every number in the vector
        sum <- sum + number
    }
    return(sum)
}
```

```
vector_sum(1:5)
```

```
## [1] 15
```

Be careful not to use names of functions that have already been defined elsewhere in R. For example, `sum()` is already a function in base R. If we were to name our function sum as well, our new function would replace the old sum functino.

Notice that the last line of my function uses the `return()`. This function is special, it says: "I want the output of my function to be this value." If we don't use `return()`, by default our output will be the last object created within the function code. If we don't specify the output of our function explicitly, it may make it more difficult for another person to read our code. More importantly, we may forget what we were outputing and could return the wrong value or object! It is good practice to be explicit whenever possible to avoid confusion and errors, especially since RStudio will not be able to help you find errors where our functions output the wrong things.

Just like we can use for loops, we can also use if/else statements. Let's look at a way to write a function that finds the minimum value in a vector.

```
vector_min <- function(numeric_vector){

    m <- numeric_vector[1] # initialize our minimum to the first element

    for(number in 2:length(numeric_vector)){
        # Now go through the rest of the vector
        if(numeric_vector[number] < m){
            # If our vector elements is less than min, update our minimum value
            m <- numeric_vector[number]
        }
    }
    # Once we finish going through our vector, output the minimum
    return(m)
}


test_vector <- c(1, 4, 6, -8, 0, 11)

vector_min(test_vector)
```

```
## [1] -8
```

Write a function called vector_max which takes a numeric vector and returns the maximum element in it. Test out your function on our test_vector.

As you've seen, functions are very useful when you have code you want to use to do something many times. Instead of having to write out our for loop to find the max of a set of numbers each time, we can simply use our function.

Let's try this out with a more fun example, rock paper scissors.

First we want to write out our code without putting it into a function. Once we are sure that it works, we can write it as a function.

```
player1 <- "rock"
player2 <- "paper"

if((player1 == "rock" & player2 == "paper") |
   (player1 == "paper" & player2 == "rock") |
```

```
   (player1 == "scissors" & player2 == "paper")){
        print("Player 1 Wins!")
} else if (player1 == player2){
    print("Tie game!")
} else {
    print("Player 2 Wins!")
}
```

```
## [1] "Player 1 Wins!"
```

Great, so we now have the "body" of our function.

Based on the above code, create a function: RPS, which takes two arguments - player1 and player2 - and returns who wins the game of rock paper scissors.

Going back to our housing and metro data. Our goal is to calculate how far each property is from the nearest metro station using latitude and longitude data.

Before we try and tackle this problem for all properties, let's first try and find the distance from one house to one metro station.

```
test_property <- joined_data[1, ]
test_metro <- metro_data[1, ]
```

Now we have one house and one metro station with a latitude and longitude for the house and a latitude and longitude value for the metro station. We can try using some trigonometry to find the distance. The distance between two points is the hypotenuse given by a triangle made from the change in the x direction and a change in the y direction. What this means is that if we find the difference in longitude values and the difference in latitude values, we can use them to find how far apart metro station and our property are.

```
delta_x <- test_property$LONGITUDE - test_metro$Longitude
```

```
## Error in test_property$LONGITUDE - test_metro$Longitude: non-numeric argument to binary operator
```

```
delta_y <- test_property$LATITUDE - test_metro$Latitude
```

```
## Error in test_property$LATITUDE - test_metro$Latitude: non-numeric argument to binary operator
```

```
delta_x
```

```
## Error in eval(expr, envir, enclos): object 'delta_x' not found
```

```
delta_y
```

```
## Error in eval(expr, envir, enclos): object 'delta_y' not found
```

Darn, we hit an error. Let's check the class of our values.

```
class(test_property$LONGITUDE)
```

```
## [1] "character"
```

```
# As we fear, character but we need numeric
test_property <- test_property %>%
    mutate(LONGITUDE = as.numeric(LONGITUDE),
           LATITUDE = as.numeric(LATITUDE))

# Now let's try again
delta_x <- test_property$LONGITUDE - test_metro$Longitude
delta_y <- test_property$LATITUDE - test_metro$Latitude
```

```
delta_x
```

```
## [1] -0.1797564
```

```
delta_y
```

```
## [1] -0.0209796
```

Now that we have our distances we can use the Pythagorean formula to calculate the distance.

```
distance <- sqrt(delta_x**2 + delta_y**2)
distance
```

```
## [1] 0.1809765
```

```
# One degree is equal to about 69 miles
distance/69
```

```
## [1] 0.002622848
```

Write a function that takes 4 arguments: property long and lat values, and metro long and lat values. The function should then calculate the pythagorean distance between the property and the metro.

Hmm, something is wrong. Our test property is in Arlington and our test metro is in Maryland east of DC, a distance of about ten miles. What happened?

The earth is not flat! We can't use a simple Pythagorean formula to calculate the distance between our two points. Luckily someone has solved this issue for us in the geosphere library.

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

```
## Loading required package: sp
```

```
prop_long_lat <- c(test_property$LONGITUDE, test_property$LATITUDE)
metro_long_lat <- c(test_metro$Longitude, test_metro$Latitude)

distance <- distHaversine(prop_long_lat, metro_long_lat)
# By default the output distance is in meters
# There are 1609.344 meters in a mile
distance/1609.344
```

```
## [1] 9.788104
```

This is much more realistic!

Now that we calculated our distance from one property to one metro, let's find a way to calculate the distance from one property to every metro using for loops.

```
output_distances <- c()

for(metro in 1:nrow(metro_data)){
    # Pull out our longitude and latitude values for the metro station
    metro_long_lat <- c(metro_data$Longitude[metro],
                        metro_data$Latitude[metro])
    # Calculate the distance to our property
    distance <- distHaversine(p1 = prop_long_lat, # from above
                              p2 = metro_long_lat)
    # append that distance to our distances vector
    output_distances <- c(output_distances, distance)
```

```
}
output_distances
```

```
##  [1] 15752.427  6808.755  5686.320  2294.317  3610.982 12334.899 13411.056
##  [8]  5991.290 14872.383 10426.885 16955.960  6462.454 15100.376  2979.417
## [15]  8054.654 17915.274  8013.070  7942.591  3061.625  2247.640 13101.042
## [22] 13471.525  5800.173  7326.065  7278.210  7117.959  5300.422  5028.079
## [29]  5674.650  5252.833  4601.759 16973.911 11795.640 13587.682 10712.317
## [36]  6166.356  9207.172 22112.838 21788.387 18702.894 18099.527  7659.582
## [43]  6193.604  6562.596 17668.589 20627.055  5381.064 13389.573  5528.208
## [50] 15109.480  5477.876 11505.533 18503.333  6520.878  6447.943 10483.111
## [57] 19864.409  7874.672  1867.334  1395.632  8053.466 15300.988  9593.329
## [64] 26900.125  3197.683  3573.158 29470.347  7058.915 14815.427  4812.142
## [71]  8711.598 16017.337  9114.143 12799.316 13296.907  9276.221 22377.886
## [78] 14160.494  7087.601  7046.896  8706.385  8906.887 17147.858  3105.545
## [85]  5343.297 10400.395 13650.843 19416.373 20580.771 24645.052  7072.820
```

Turn the above code into a function called prop_metros_dist that does the following:

- **Finds the distance from a property to every metro station**
- **Converts the distance from meters to miles**
- **Returns the distance (in miles) of the closest metro station**

```
## [1] 0.8672053
```

We are so close, now we just need to figure out how to apply this function for every property in our data-set. We have one main problem, our distance function expects us to give longitude and latitude as a two number vector, but we have a LATITUDE column and a LONGITUDE column in our joined_data. What we need is a "wrapper function."

A wrapper function is really just a function that calls another function. For example, the function that we just wrote is a "wrapper function" around distHaversine. What we need to do is write a function that can take our LATITUDE and LONGITUDE columns, turn them into a vector of two numbers, and then pass them along to our prop_metros_dist function which can give us the distance of the closest metro to that property.

Function Saftey

While functions are another convinient way to help us improve the readibility and replicibility of our code, user errors can throw a wrench in our program. Look at the above wrapper function. We need to have the geosphere package installed in order to utilize `prop_metros_dist()`. Unless you read the code `prop_metros_dist()` and *also* recognize that `distHaversine()` is a function from geosphere, then a user may not install and load the package. It is our job as programmers to ensure that our functions can address these concerns.

require() and warning()

We will utilized two functions that can assist us in function saftey: `require()` and `warning()`. The first function works similarly to `library()` and attempts to load and attach a package, however `require()` was designed specifically to be used *inside* other functions. When a package fails to load, `require()` returns `FALSE` and gives us a warning message rather than an error. This allows us to use `require()` with if/else statments to load packages and control whether the rest of the function is run or not. When we combine this with `warning()`, a funtion that prints warning messages, we are able to more accurately describe why our functions are not working so users can make the appropratie changes.

```
distance_function <- function(LONG, LAT, metro_df){
    if(!require(geosphere)){ # what does require do?
      warning("geosphere package is not installed")
    } else{
      # We need to turn our LONG and LAT columns into a vector
      prop_long_lat <- c(LONG, LAT)
      # Now call our prop_metros_dist function from above
      out <- prop_metros_dist(prop_long_lat,
                              metro_df)
      return(out)
    }
}
```

```
#test that it does not give us an error
distance_function(joined_data$LONGITUDE[3],
                  joined_data$LATITUDE[3],
                  metro_data)
```

Improving our function

You may have noticed that we have still recieved an error. This one talks about a inputing a non-numeric argument. Let's make sure that our input data were the correct classes.

```
class(joined_data$LONGITUDE[3])
```

```
## [1] "character"
```

```
class(joined_data$LATITUDE[3])
```

```
## [1] "character"
```

Ah, there is the problem! We need to update our data.

```
joined_data <- joined_data %>%
    mutate(LATITUDE = as.numeric(LATITUDE),
           LONGITUDE = as.numeric(LONGITUDE))

#test that it does not give us an error
distance_function(joined_data$LONGITUDE[3],
                  joined_data$LATITUDE[3],
                  metro_data)
```

```
## [1] 0.7918705
```

It works! Now we can apply it over our entire data frame.

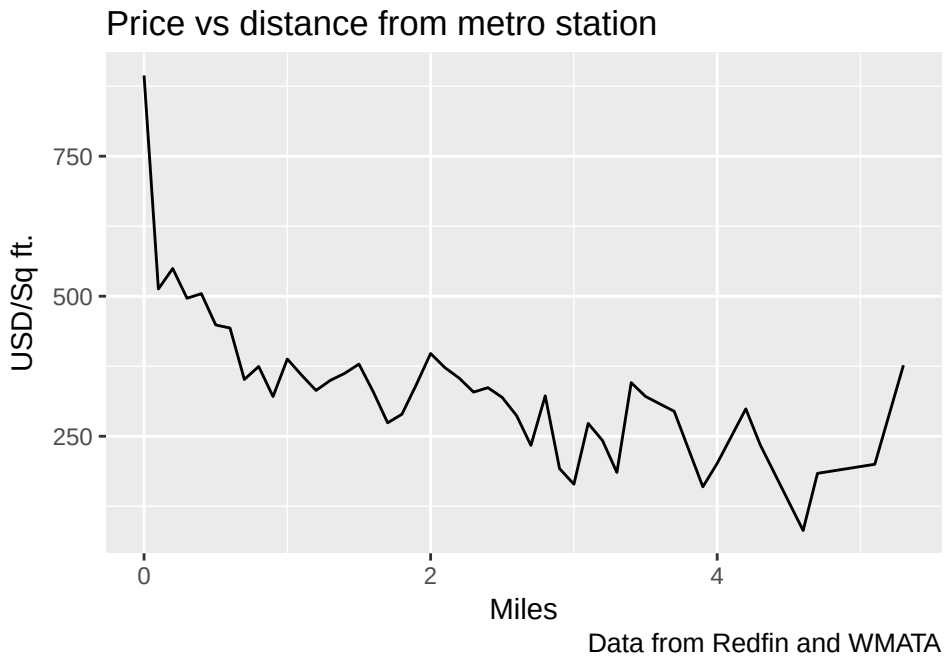
How could we improve this function in order to ensure function saftey?

```
joined_data$metro_distance <- NA #initialize the column

for(i in 1:nrow(joined_data)){
    joined_data$metro_distance[i] <- distance_function(joined_data$LONGITUDE[i],
                                                       joined_data$LATITUDE[i],
                                                       metro_data)
}
head(joined_data$metro_distance)
```

```
## [1] 0.8672053 2.1850417 0.7918705 2.2188679 2.4463603 0.3075996
```

Now that we have a column showing how far each property is from the metro, we can make a plot showing price as it varies for metro distance. Note that we will want to round our distance value so that we can get more properties in each distance bucket.

```
joined_data <- joined_data %>%
    mutate(distance_tenth = round_any(metro_distance, 0.1))

joined_data %>% group_by(distance_tenth) %>%
    summarize(price = mean(PRICE/SQUARE.FEET, na.rm = T)) %>%
    ggplot(aes(x = distance_tenth, y = price)) +
    geom_line() +
    labs(title = "Price vs distance from metro station",
         y = "USD/Sq ft.",
         x = "Miles",
         caption = "Data from Redfin and WMATA")
```



What if any relationship does this seem to show between the distance of a property to a metro station and the price per square foot of the property?

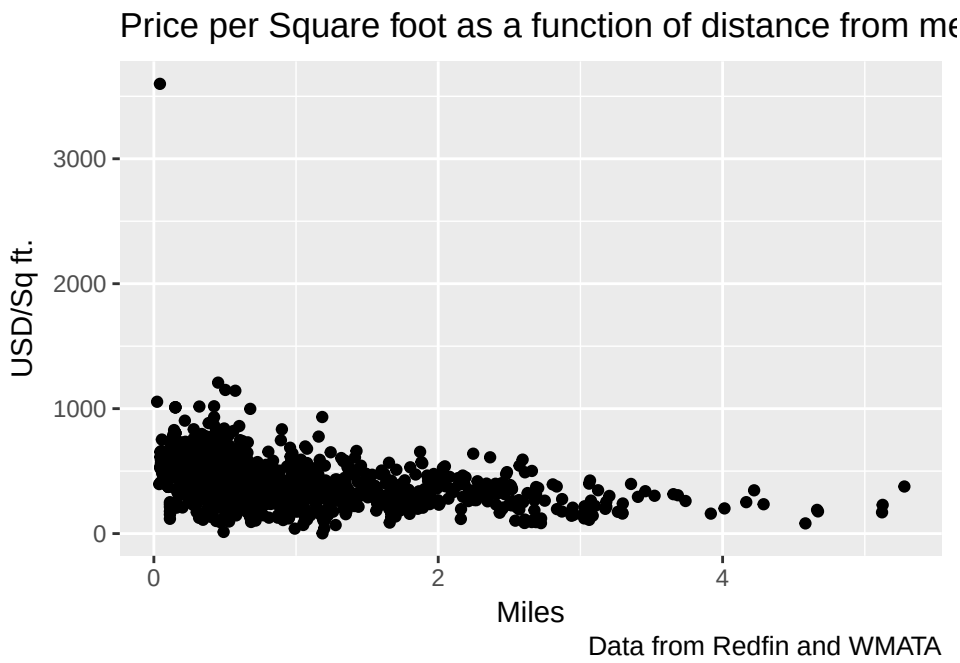
Day Four

Effect of metro distance on home prices

Previously in this module we read in our raw property data from Redfin, cleaned it, and then used our metro station location data to understand how close the nearest metro station is to each of our properties. Now we can try running some regressions to look at the impact that metro distance might have on our prices.

```
joined_data %>%
    filter(!is.na(SQUARE.FEET)) %>%
    ggplot(aes(x = metro_distance,
               y = PRICE/SQUARE.FEET)) +
```

```
    geom_point() +
    labs(title = "Price per Square foot as a function of distance from metro",
         x = "Miles",
         y = "USD/Sq ft.",
         caption = "Data from Redfin and WMATA")
```



Wow, we really don't see all that much of a relationship here. Let's try using a regression model.

```
price_distance <- lm(PRICE/SQUARE.FEET ~ metro_distance, data = joined_data)
```

And now we can use stargazer to look at the impact of metro distance on price.

```
library(stargazer)
```

```
##
## Please cite as:
##  Hlavac, Marek (2015). stargazer: Well-Formatted Regression and Summary Statistics Tables.
##  R package version 5.2. http://CRAN.R-project.org/package=stargazer
```

```
stargazer(price_distance, header = F,
          title = "Impact of Distance From Metro on Price",
          type = "text")
```

```
##
## Impact of Distance From Metro on Price
## ================================================
##                         Dependent variable:
##                     ---------------------------
##                           PRICE/SQUARE.FEET
## ------------------------------------------------
## metro_distance               -93.621***
##                               (6.467)
##
## Constant                     503.684***
```

```
##                                (8.202)
##
## ------------------------------------------------
## Observations                    1,179
## R2                              0.151
## Adjusted R2                     0.150
## Residual Std. Error     184.570 (df = 1177)
## F Statistic          209.565*** (df = 1; 1177)
## ================================================
## Note:                *p<0.1; **p<0.05; ***p<0.01
```

So it seems that metro distance does have a negative effect on the sales price of the house. Perhaps there are other unobservable factors that may affect home price. Let's add in another variable we looked at earlier in our analysis, state.

Can you think of any STATE SPECIFIC factors that may affect home price?

```
state_dist <- lm(PRICE/SQUARE.FEET ~ metro_distance + STATE,
                 data = joined_data)

stargazer(price_distance, state_dist, type = "text",
          header = F)
```

```
##
## ===============================================================================
##                                      Dependent variable:
##                      ---------------------------------------------------------
##                                      PRICE/SQUARE.FEET
##                                (1)                          (2)
## -------------------------------------------------------------------------------
## metro_distance             -93.621***                   -81.569***
##                              (6.467)                      (7.352)
##
## STATEMD                                                 -54.973***
##                                                          (14.816)
##
## STATEVA                                                   0.995
##                                                          (13.996)
##
## Constant                    503.684***                   505.487***
##                              (8.202)                      (8.592)
##
## -------------------------------------------------------------------------------
## Observations                    1,179                        1,179
## R2                              0.151                        0.163
## Adjusted R2                     0.150                        0.161
## Residual Std. Error     184.570 (df = 1177)      183.453 (df = 1175)
## F Statistic          209.565*** (df = 1; 1177) 76.168*** (df = 3; 1175)
## ===============================================================================
## Note:                                          *p<0.1; **p<0.05; ***p<0.01
```

What STATE is missing from the results, how should we interpret these coefficients? What happened to the coefficient for metro distance when we added our STATE variable? What about the explanatory power of the model, did that improve?

However, home prices are affected by more than just the location. The amount of time a home spends on the market is another componenet that could affect home prices. Ideally, a homeowner would sell there house

quickly at the price they desire but sometimes the prices are too high or there is little demand to move in the area. As the homeowner keeps their house on the market longer and longer, they may be willing to lower the price in order to sell the home and purchase a new one.

We are not provided with this data, but luckily have the date the house was listed and the date it finally sold! In order to calculate the amount of days a home has spent on the market, we first need to work with dates.

```
joined_data %>% select(LIST.DATE, SOLD.DATE) %>% head()
```

```
##                 LIST.DATE         SOLD.DATE
## 1   Tuesday- Sep 26, 2017 September-29-2017
## 2    Friday- Aug 11, 2017  September-6-2017
## 3 Wednesday- Sep 06, 2017 September-19-2017
## 4   Tuesday- Sep 26, 2017 September-29-2017
## 5    Monday- Sep 18, 2017 September-25-2017
## 6 Wednesday- Aug 23, 2017 September-12-2017
```

```
date1 <- as.Date("July-15-2017", format = "%B-%d-%Y")

date2 <- as.Date("Tuesday- Sep 26, 2017", format = "%A- %b %d, %Y")
```

We now have our dates not as character strings but as date objects in R. Dates in R are just numbers which are displayed in a special format. Because they are numbers we can do mathematical operations with them.

```
date2-date1
```

```
## Time difference of 73 days
```

Additionally, we can generate date sequences.

```
seq(date1, date2, by = "day")
```

```
##  [1] "2017-07-15" "2017-07-16" "2017-07-17" "2017-07-18" "2017-07-19"
##  [6] "2017-07-20" "2017-07-21" "2017-07-22" "2017-07-23" "2017-07-24"
## [11] "2017-07-25" "2017-07-26" "2017-07-27" "2017-07-28" "2017-07-29"
## [16] "2017-07-30" "2017-07-31" "2017-08-01" "2017-08-02" "2017-08-03"
## [21] "2017-08-04" "2017-08-05" "2017-08-06" "2017-08-07" "2017-08-08"
## [26] "2017-08-09" "2017-08-10" "2017-08-11" "2017-08-12" "2017-08-13"
## [31] "2017-08-14" "2017-08-15" "2017-08-16" "2017-08-17" "2017-08-18"
## [36] "2017-08-19" "2017-08-20" "2017-08-21" "2017-08-22" "2017-08-23"
## [41] "2017-08-24" "2017-08-25" "2017-08-26" "2017-08-27" "2017-08-28"
## [46] "2017-08-29" "2017-08-30" "2017-08-31" "2017-09-01" "2017-09-02"
## [51] "2017-09-03" "2017-09-04" "2017-09-05" "2017-09-06" "2017-09-07"
## [56] "2017-09-08" "2017-09-09" "2017-09-10" "2017-09-11" "2017-09-12"
## [61] "2017-09-13" "2017-09-14" "2017-09-15" "2017-09-16" "2017-09-17"
## [66] "2017-09-18" "2017-09-19" "2017-09-20" "2017-09-21" "2017-09-22"
## [71] "2017-09-23" "2017-09-24" "2017-09-25" "2017-09-26"
```

We can increment our sequence by different time units:

```
seq(date1, date2, by = "week")
```

```
##  [1] "2017-07-15" "2017-07-22" "2017-07-29" "2017-08-05" "2017-08-12"
##  [6] "2017-08-19" "2017-08-26" "2017-09-02" "2017-09-09" "2017-09-16"
## [11] "2017-09-23"
```

```
seq(date1, date2, by = "2 weeks")
```

```
## [1] "2017-07-15" "2017-07-29" "2017-08-12" "2017-08-26" "2017-09-09"
## [6] "2017-09-23"
```

```
seq(date1, date2, by = "month")
```

```
## [1] "2017-07-15" "2017-08-15" "2017-09-15"
```

Knowing that we can use dates for math, how would you generate a sequence of dates showing the first day of the month for the year 2017? How about the last day of each month for 2017?

For our analysis we want to see how the length of time on the market impacts the final sales price of the home. To do this we will need to convert our current character date columns in R date values before manipulating those values.

```
joined_data <- joined_data %>%
    mutate(SOLD.DATE = as.Date(SOLD.DATE, format = "%B-%d-%Y"),
           LIST.DATE = as.Date(LIST.DATE, format = "%A- %b %d, %Y"),
           days_on_market = as.numeric(SOLD.DATE - LIST.DATE))
```

Using our new days_on_market value, create a new variable, "weeks_on_market" which should be exclusively integer values (3 days would be 0 weeks and 4 days would be 1 week).

Now we can add our time on the market to our regressions.

```
days_regression <- lm(PRICE/SQUARE.FEET ~ metro_distance + days_on_market,
                      data = joined_data)

stargazer(days_regression, type = "text")
```

```
##
## ================================================
##                         Dependent variable:
##                     ---------------------------
##                           PRICE/SQUARE.FEET
## ------------------------------------------------
## metro_distance               -104.118***
##                               (7.459)
##
## days_on_market                -0.037
##                               (0.757)
##
## Constant                     513.691***
##                               (14.171)
##
## ------------------------------------------------
## Observations                    970
## R2                             0.168
## Adjusted R2                    0.166
## Residual Std. Error      186.268 (df = 967)
## F Statistic           97.456*** (df = 2; 967)
## ================================================
## Note:               *p<0.1; **p<0.05; ***p<0.01
```

Although our adjusted R^2 has improved, the number of days on the market appears to have no affect on the price of a home. Bummer.