Module 2

Day 3

Recap last week

- “Joined” location and property data using the url column
- Introduce `str_replace`, `str_split_fixed`, `str_trim`, and `str_to_upper` to clean variables in our data

Goals for Today

R:

- Finish cleaning current data
- Motivate the use of user-created functions and "function safety"
- Learn how to create functions
- Create our own function to find the distance from each house to the nearest house

Economics:

- Take
- Begin exploring the relationshiop between distance from a metro stop and home prices

Cleaning zip-codes

- Let's investigate our zip-code variable

```r
dat_path <- ## Your file path here
joined_data <- read_csv(paste0(dat_path,
                               "joined_data.csv"))
head(joined_data$ZIP_CODE)
```

```
## # A tibble: 6 x 28
##   SALE.TYPE          SOLD.DATE PROPERTY.TYPE  PRICE  BEDS BATHS SQUARE.FEET
##       <chr>              <chr>         <chr>  <dbl> <int> <dbl>       <int>
## 1 PAST SALE September-29-2017   Condo/Co-op 280000     2   1.0        1055
## 2 PAST SALE  September-6-2017   Condo/Co-op 405000     2   2.0        1030
## 3 PAST SALE September-19-2017   Condo/Co-op 510000     2   2.0        1209
## 4 PAST SALE September-29-2017   Condo/Co-op 395000     2   2.0        1135
## 5 PAST SALE September-25-2017   Condo/Co-op 339900     2   1.0         930
## 6 PAST SALE September-12-2017   Condo/Co-op 415000     2   2.5        1606
## # ... with 21 more variables: LOT.SIZE <int>, YEAR.BUILT <int>,
## #   HOA.MONTH <int>, STATUS <chr>, NEXT.OPEN.HOUSE.START.TIME <chr>,
## #   NEXT.OPEN.HOUSE.END.TIME <chr>, URL <chr>, SOURCE <chr>, MLS. <chr>,
## #   FAVORITE <chr>, INTERESTED <chr>, LIST.DATE <chr>, ADDRESS <chr>,
## #   ZIP_CODE <chr>, LAT_LON <chr>, CITY_STATE <chr>, LOCATION <chr>,
## #   CITY <chr>, STATE <chr>, LATITUDE <dbl>, LONGITUDE <dbl>
```

- We can see two big problems with our current data:

str_sub and str_length

- `str_length()` tells us the number of characters in a string
- `str_sub()` returns part of a text string between the start and end position provided

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

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

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

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

str_sub and str_length

- `str_length()` is primarily used with other functions
- We can combine both functions in order to extract the actual zip-code from our variable

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

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

- **Use dplyr and these new stringr functions to fix the rest of our zip-code data**

Reviewing the Data

- We'll start by looking at the head of the data

```
## # A tibble: 6 x 3
##    PRICE SQUARE.FEET LOT.SIZE
##    <dbl>       <int>    <int>
## 1 280000        1055     1055
## 2 405000        1030     1030
## 3 510000        1209     1209
## 4 395000        1135     1135
## 5 339900         930      930
## 6 415000        1606     1606
```

- We have a lot of NA values. Why is that?

Cleaning The Data

- Let's get rid of the `NA` values by replacing them with square footage
- We'll create a new column called Property Size that equals either lot size or square feet
  - Use an `ifelse()` statement in our mutate, where we test whether lot size is `NA`
  - If lot size is `NA`, use square feet. Otherwise, use lot size
- Next we will summarize our data by price per foot of property size.
  - We want to group by property type and state!

Cleaning Code

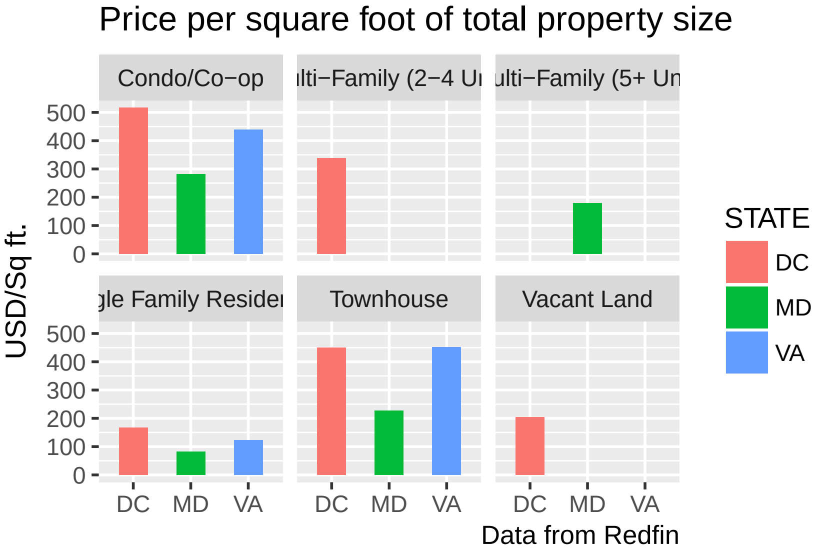
```r
joined_data <- joined_data %>%
    mutate(PROPERTY.SIZE = ifelse(is.na(LOT.SIZE), SQUARE.FEET, LOT.SIZE))

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

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
```

Visualizing The Relationship

- **Make a plot showing the average price per lot size for each property type in each state**

Finished Plot



Different Forms of Data

- There are two types of data that we work with:
  1. Long
  2. Wide
- Up until now we have been looking at only one format of data — long type
  - Much easier to manipulate and graph
- Wide-form data is easier to see visually
  - Generally used to display tables

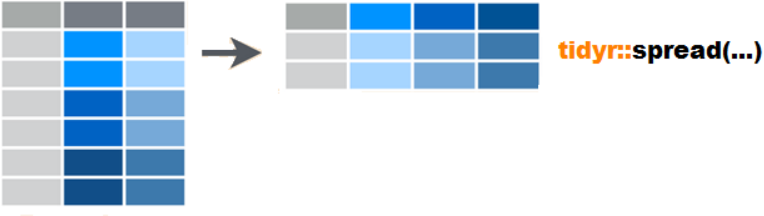
A Real Estate Example

- Let's look at a toy example with data from Redfin

```
wide_data <- read_csv(paste0(dat_path,
                             "wide_sample.csv"))
```

- Each observation is a property at every point in time for our data
  - How is this different from an observation in our original data?
  - If this data were in long form, how many columns would there be?

Converting Our Data



gather()

- `gather(data, key, value, ...)`
  - Converts multiple columns from wide-format to long-format
  - `key` - name of the variable you are trying to create
  - `value` - variable that holds the values in the key variables
  - Must specify which columns to use for the key

```
long_data <- wide_data %>%
  gather(key = "year", value = "price", 5:11)
```

spread()

- `spread(data, key, value, ...)`
  - Converts two columns from long-format to wide-format
  - `key` - variable you wish to spread into multiple columns
  - `value` - variable that holds the values that will be used by your new variables

```
wide_data <- long_data %>%
  spread(key = "year", value = "price")
```

Other Uses

- We can use our reshaping functions in conjunction with stargazer to produce good looking tables

```
lot_state_data %>%
  spread(key = STATE, value = price) %>%
  stargazer(summary = FALSE,
            header = FALSE,
            type = "Latex")
```

Reading In The Metro Station Data

- Use this data in conjunction with the housing data to find the distance from a metro
  - Includes latitude and longitude of each metro station
- Unfortunately it is in an xlsx file so we can't use read_csv

```
library(xlsx)
metro_data <- read.xlsx(paste0(dat_path,
                               "Metro_lat_lon.xlsx"),
                               sheetIndex = 1)
```

- There is one big problem with finding the distance — the calculation is fairly difficult
  - We need to create a function!

What Is A Function

- Piece of code that takes one or more inputs and returns one or more outputs
- For example:
  - `min()` takes a vector of numbers and returns the number with the lowest value
  - `read_csv()` takes a string that contains a file path and returns a data.frame object
- Some of inputs may have a default set value for certain arguments
  - `min(..., na.rm = FALSE)`

Writing A Function

- Functions are extremely useful when you have to repeat a complex chunk of code
  - Can be used in loops, if else statements, and even other functions
- A user-defined function takes the following form:

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

Simple Examples

- Here is a funciton for adding two numbers together:

```
add <- function(num1, num2){
    num1 + num2
}
add(num1 = 2,num2 = 5)
```

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

- **Create your own function that raises a number provided by the user to a power provided by the user.**

Defaults

- Use these when one option is used much more than any of the other options
- For example `read_csv(..., col_names = T)`
  - read in the first row of the data as column names
  - may also use `FALSE` or a vector of column names we would like to assign
- Merely set one of our arguments equal to the default value

```
root <- function(base, denom = 2){
  base**(1/denom)
}
```

return()

- Let's take a look the functions we created so far
  - What do we input and what do we output?
  - Is this completely clear from our code?
- By default, a function returns the last object that was created
  - Can be confusing when reading someone elses code
- `return()` says I want the output of my function to be this value
  - Only useable inside of a user-generated function

A More Complex Example

- We can use any of the coding structures we've already learned inside of our functions
- Let's create a function that sums together the numbers in a vector

```
vector_sum <- function(numeric_vector){
    sum <- 0
    for(number in numeric_vector){
        sum <- sum + number
    }
    return(sum)
}
vector_sum(c(1,1,5,12))
```

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

Another Complex Example

- Let's find the minimum number in a vector

```
vector_min <- function(numeric_vector){
    m <- numeric_vector[1]
    for(number in 2:length(numeric_vector)){
        if(numeric_vector[number] < m){
            m <- numeric_vector[number]
        }
    }
    return(m)
}

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

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

Another Complex Example

Now it's your turn. Write a function called vector_max which takes a numeric vector and returns the maximum element in it. Make sure to test out your function.

Rock, Paper, Scissors

- Let's try using a more fun example now with a simple logic game

```
if(( & ) | ( & ) | ( & )){
        print("Player 1 Wins!")
} else if (){
    print("Tie game!")
} else {
    print("Player 2 Wins!")
}
```

Fill in the logical arguments above. Create a function called RPS that uses the code as the body of the function and returns the results of the match.

Returning to Metro Data

- Remember, our goal is to calculate the nearest metro station
  - Use our longitude and latitude data combined with a little trigonometry to find the distance
  - Remember: $a^2 + b^2 = c^2$ or $\Delta longitude^2 + \Delta latitude^2 = distance^2$

Calculating Distance

- Now that our data is numeric we can calculate the distance

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

delta_x <- test_property$LONGITUDE - test_metro$Longitude
delta_y <- test_property$LATITUDE - test_metro$Latitude

distance <- sqrt(delta_x**2 + delta_y**2)
# One degree is equal to about 69 miles
distance/69
```

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

Basic Distance Function

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 in miles.

```
metro_dist <- function( , , , ){



}
```

Double checking the distance

- Our test property is in Arlington and our test metro is in Maryland east of D.C.
  - That's about 10 miles
  - Why might our distance be incorrect?

geosphere

- We can't use the simple Pythagorean formula
- `geosphere` is a package that allows us to easily work with geographic coordinates
- `distHaversine()` calculates the shortest distance between two points
  - Points must contain longitude AND latitude
  - "As the crow flys"
  - Default output is in meters

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

Calculating distance

```r
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)
# There are 1609.344 meters in a mile
distance/1609.344
```

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

- This looks much more realistic!

Every Metro Stop

```r
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
```

Better Distance Function

Turn the code from the previous slide 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**

Function safety

- Functions help use improve readability and replicability of our code, but are not fool proof
- People often input the wrong arguments or use functions in ways that weren't intended
- Let's look at `prop_metros_dist()`:
  - If a user doesn't recoginze `distHaversine()` is from the geosphere package, they may not install and load it
  - It is our job as the programmer to address these types of concerns

Functions for Function safety

- `require()` — works similarly to `library()`
  - Designed to be used within other functions
  - Returns `FALSE` if a package fales to load
  - Can use this with if/else statements
- `warning()` — prints a warning message that we specify but DOES NOT stop the program
- `stop()` — prints an error message and stops the program

Function safety in Action

```
prop_metros_dist <- function(prop_long_lat, metro_data){

  if(!require(geosphere)){ # what does require do?
      warning("geosphere package is not installed")
    } else{
  output_distances <- c()
  for(i in 1:nrow(metro_data)){
        metro_long_lat <- c(metro_data$Longitude[i],
                            metro_data$Latitude[i])
        distance <- distHaversine(p1 = prop_long_lat,
                                  p2 = metro_long_lat)
        output_distances <- c(output_distances, distance)
    }

    return(min(output_distances)/1609.344)
    }
}
```

Wrapper Functions

- Need a two-number vector with longitude and latitude but we have two columns in our data
- A function that calls another function
  - Generally pre-process inputs for other functions
  - Improve readibility, ease of use, and our ability to implement changes
  - `prop_metros_dist()` calls `dist_Haversine()`
- Want to write a function that turns `LATITUDE` and `LONGITUDE` into a vector

Our Wrapper Function

```r
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)
    }
}

distance_function(joined_data$LONGITUDE[3],
                  joined_data$LATITUDE[3],
                  metro_data)
```

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

Finding the Closest Metro Station

- **Combine our function with a loop to find the closest metro station to each property (Hint: Look at one of our previous functions)**

```
joined_data$metro_distance <- NA #initialize the column
for()){

  ## Your code here

}
```

Graphing the Relationship