Chapter 4 Data Wrangling

4.1 Libraries

Up until now, we haven't used packages - only functions available in base R. However, the power of R truly exists in the libraries, which are sets of functions that any user can create and share.

As of June 2019, there were over 14,000 packages available on the Comprehensive R Archive Network, or CRAN, the public clearing house for R packages. CRAN lists all libraries here: https://cran.r-project.org/web/packages/available_packages_by_name.html

To install a library, you can either:

- Packages `-->` Install `-->` Name
- Use the function install.packages()

Once a package is installed, you need to load it using the library or require function.

These are the libraries we'll be using today. Install if you don't have:

```
install.packages(c("data.table","dplyr","stringr"))
```

Now, import the libraries:

```
library(data.table)
library(dplyr)
library(stringr)
```

4.2 Importing & Exporting Data

Before importing data, we generally want to set the working directory. This is a folder where we will be accessing or storing data and outputs:

```
setwd("C:/Users/avillan/Documents/Class 2022/Class_Code")
```

Another useful function is `getwd()`. This function returns an absolute filepath representing the current working directory of the R process, or `NULL` if the working directory is not available:

```
getwd()
```

To import data, we can use the `read.` function

```
df_auto <- read.csv("Auto_MPG.csv", stringsAsFactors = FALSE)
```

We can check the data set:

```
class(df_auto)
View(df_auto)
dim(df_auto)
nrow(df_auto)
summary(df_auto)
```

Note that Horsepower is a character class

```
typeof(df_auto$Horsepower)
df_auto$Horsepower
summary(df_auto$Horsepower)
```

Upon inspection, it read NAs as question marks (a character), so everything became a character. We can fix this by reassigning as numeric:

```
df_auto$Horsepower <- as.numeric(df_auto$Horsepower)
summary(df_auto)
df_auto$Horsepower
summary(df_auto$Horsepower)
```

Recall if you want to fix a column name you can use the colnames() function:

```
colnames(df_auto)
colnames(df_auto)[8] <- "Origin"
colnames(df_auto)
```

If we want to export the data, we can use write.csv. The row.names = FALSE argument will stop the writer from outputting a row number:

```
write.csv(df_auto, file="Auto_MPG_Clean.csv", row.names = FALSE)
```

In addition to importing the data using code, one can also import data into RStudio with point-and-click. Go to `Import Dataset` in the Environment panel and select the relevant format. From there, you can set the datatype of each column, as well as specify whether to include or to skip a column or row. RStudio automatically generates the corresponding code that you can copy and paste for later reproducibility.

4.3 dplyr and tidyverse

One of the most difficult (yet, interesting!) parts of open-source software is the many ways to achieve the same task. This can make recalling functions and syntax difficult, increasing the time to write or read code. Thus, people have started creating packages that structure coding tasks into a simple but comprehensive set of functions. You'll see this referred to as a "grammar". This framework attempts to simplify the majority of required functions into a simple set of "verbs" to remember.

If you're familiar with SQL - think about how easy it is to read and write:

`SELECT * FROM table WHERE column = 'some_value'`
In SQL, `SELECT`, `FROM`, `WHERE` are "verbs" that provide the framework for numerous tasks.

The packages plyr, dplyr, ggplot2, stringr (and others) work within this framework. "tidyverse" is a collection of packages that encompases all of these https://www.tidyverse.org/

4.4 Data manipulation - dplyr basics

Select: select columns by name or helper function (choose which variables you want to look at):

```
  select(df_auto, MPG, Cylinders)
  select(df_auto, -Origin)
  select(df_auto, contains("Model"))
```

The base R equivalents are:

```
  df_auto$MPG
  df_auto[,c('MPG','Cylinders')]
  df_auto[,-c('Origin')]
  df_auto[,grepl("Model", names(df_auto))]
```

Slice: Select rows by position (choose which rows of data you want):

```
  df_auto$row_number <- 1:nrow(df_auto) #adding a new variable called row_number
to help us see the index

  slice(df_auto,1:5)
  slice(df_auto, 150:n()) #all the rows between 150 and the end
  slice(df_auto, -1:-5) #removes rows between values, equivalent to slice(df_auto,
6:n())

  df_auto <- select(df_auto, -row_number) #remove the row_number column
```

The base R equivalents are:

```
  df_auto[1:5,]
  df_auto[150:nrow(df_auto),]
  df_auto[-c(1:5),]
```

Rename: Rename the columns of a data frame

```
  colnames(df_auto)

  df_auto <- rename(df_auto, Miles_Per_Gallon=MPG) #rename MPG to Miles_Per_Gallon
  colnames(df_auto)

  df_auto <- rename(df_auto, MPG=Miles_Per_Gallon) #rename back Miles_Per_Gallon
to MPG
  colnames(df_auto)
```

The base R equivalents are:

```
  colnames(df_auto)[1] <- 'Miles_Per_Gallon'
  colnames(df_auto)

  colnames(df_auto)[1] <- 'MPG'
  colnames(df_auto)
```

Filter: Extract rows that meet logical criteria (filter the data frame by one, or multiple, conditions)

```
  filter(df_auto, MPG>20)
  filter(df_auto, (MPG>20 & Acceleration>20))
  filter(df_auto, MPG==max(MPG))
  filter(df_auto, MPG==min(MPG))
```

The base R equivalents are:

```
  df_auto[df_auto$MPG>20,]
  df_auto[(df_auto$MPG>20 & df_auto$Acceleration>20),]
  df_auto[which.max(df_auto$MPG),]
  df_auto[which.min(df_auto$MPG),]
```

Sample: Randomly select a sample of n size or a proportion

```
  sample_n(df_auto, 5)
  sample_frac(df_auto, 0.05)
```

The base R equivalents are:

```
  df_auto[sample(1:nrow(df_auto),5),]
  df_auto[sample(1:nrow(df_auto),ceiling(0.05*nrow(df_auto))),]
```

Mutate: Compute and append one or more new columns (create a new variable - returns the data frame)

```
  mutate(df_auto, HP_Per_Cylinder=Horsepower/Cylinders)
  mutate(df_auto, Avg_HP_Per_Cylinder = mean(Horsepower/Cylinders, na.rm=TRUE))
```

The base R equivalents are:

```
  df_auto$HP_Per_Cylinder <- df_auto$Horsepower / df_auto$Cylinders
  df_auto$Avg_HP_Per_Cylinder <- mean(df_auto$Horsepower / df_auto$Cylinders, 
na.rm = TRUE)
```

About the na.rm=TRUE argument. Some arithmatic functions sum, avg, count, etc. will NA the entire column if they encounter a single NA within that column/vector. Setting na.rm=TRUE will tell the interpreter to “remove” the NA and carry forth with the calculation.

Summarise: Summarise data into single row of values (aggregates the data into a single result. think avg, mean, min, max, etc.)

```
  summarise(df_auto,
            Avg_HP_Per_Cylinder=mean(Horsepower/Cylinders, na.rm = TRUE),
            Min_HP_Per_Cylinder=min(Horsepower/Cylinders, na.rm=TRUE)
            )
```

The base R equivalents are:

```
  data.frame(
      Avg_HP_Per_Cylinder = mean(df_auto$Horsepower/df_auto$Cylinders, na.rm = 
TRUE),
      Min_HP_Per_Cylinder = min(df_auto$Horsepower/df_auto$Cylinders, na.rm=TRUE)
    )
```

4.5 dplyr - pipe operator

Consider the previous operations. If we wanted to manipulate our data set in order, we traditionally have to do a reassignment each time:

```
  df_auto_hold <- rename(df_auto, Miles_Per_Gallon=MPG) #rename

  df_auto_hold <- mutate(df_auto_hold, HP_Per_Cylinder=Horsepower/Cylinders) #then 
create

  df_auto_hold <- filter(df_auto_hold, HP_Per_Cylinder>20) #then filter

  df_auto_hold <- select(df_auto_hold, Car_Name, HP_Per_Cylinder) #then select

  View(df_auto_hold)
```

This is truely cumbersome the more complicated the data manipulation becomes. To avoid this, we can use the pipe operator `%>%`. The pipe "passes along" the results of one function to the next:

```
  df_auto_hold <- df_auto %>% #note the pipe operator
    rename(Miles_Per_Gallon=MPG) %>% #note the first argument is not required,
because the pipe is passing it
    mutate(HP_Per_Cylinder=Horsepower/Cylinders) %>%
    filter(HP_Per_Cylinder>20) %>%
    select(Car_Name, HP_Per_Cylinder)

  View(df_auto_hold)
```

4.6 stringr package

The package stringr is a similar grammar language for manipulating strings. All the functions start with str_

Consider that I want to extract the Car Make and Model from the Car Name:

```
  View(df_auto_hold)
```

We note that the first word in Car_Name is the make, and the second word is the model. The str_split function will split the string into a vector of individual tokens:

```
  df_auto_hold <- df_auto %>%
    mutate(Car_Make=str_split(Car_Name," "))

  View(df_auto_hold)
```

The str_split creates a list, which we can use to try to get the value:

```
  df_auto_hold <- df_auto %>%
    mutate(Car_Make=str_split(Car_Name," ")[[1]][1])
  View(df_auto_hold)
```

Note, this is treating the entire column, instead of by row. One way to address this is to rowise() by Car_Name, then create the variable:

```
  df_auto_hold <- df_auto %>%
    rowwise() %>%
    mutate(Car_Make=str_split(Car_Name," ")[[1]][1])

  View(df_auto_hold)
```

An alternative to this is the group_by() function:

```
  df_auto_hold <- df_auto %>%
    group_by(Car_Name) %>%
    mutate(Car_Make=str_split(Car_Name," ")[[1]][1])

  View(df_auto_hold)
```

With the structure built, I can just pipe on more functions to get additional variables:

```
  df_auto <- df_auto %>%
    rowwise() %>%
    mutate(Car_Make=str_split(Car_Name," ")[[1]][1]) %>%
    mutate(Car_Model=paste(str_split(Car_Name," ")[[1]][-1], collapse=" "))
#everything but the first

  View(df_auto)
```

Now that we have Car Make and Model, we might want to summarize some information. We saw summarize the entire column earlier, but we can additionally summarize by a group. We use the group_by() to do so:

```
  # Average MPG by Make and Model Year
  df_auto %>%
    group_by(Car_Make, Model_Year) %>%
    summarise(Average_MPG=mean(MPG)) %>%
    arrange(Model_Year) %>%
    filter(Model_Year==70) %>%
    ungroup(Car_Make, Model_Year)
```

4.7 Combining Data - dplyr

Typically when working with data, especially relational databases, we have information stored in multiple tables. For instance, we have the Auto data set:

```
  View(df_auto)

  # But we also have data on when a brand started:
  Car_Make <- c("amc", "audi", "bmw", "buick", "chevrolet", "datsun", "dodge",
"ford")
  First_Year <- c(1954, 1910, 1916, 1903, 1911, 1931, 1900, 1903)

  df_auto_start <- data.frame(Car_Make=Car_Make,
                              First_Year=First_Year,
                              stringsAsFactors = FALSE)

  View(df_auto_start)
```

We are interested in the relationship between founding year and average MPG for the max year. For this, we need to get my First Year data combined with the Auto MPG Data. This is accomplished via joins.

Dplyr has different join types. The easiest way to think about this is as a Venn-Diagram.

- inner_join - only returns the data where both frames contain the join key(s), removes all non-matching rows
- left_join - maintains all of the data on base table (left/top), and joins matching data from the joining table. NAs created for joining table
- right_join - opposite of left. Maintains all of the data on the joining table and matches from the base table. NAs created for base table
- full_join - returns all data from both tables. NAs created for both tables.

Inner_join:

```
  df_auto_hold <- df_auto %>%
    inner_join(df_auto_start, by="Car_Make")

  View(df_auto_hold)

  # Note, we've effectively filtered a lot of data because the inner_join only
keeps matching records.
  dim(df_auto)
  dim(df_auto_hold)

  # Joins can pipe together with other functions just as we expect
  df_auto_hold <- df_auto %>%
    inner_join(df_auto_start, by="Car_Make") %>%
    group_by(Car_Make, First_Year) %>%
    summarise(Average_MPG=mean(MPG))

  View(df_auto_hold)

  plot(df_auto_hold$First_Year, df_auto_hold$Average_MPG)
```

Left_join:

```
  df_auto_hold <- df_auto %>%
    left_join(df_auto_start, by="Car_Make")

  # note, we've retained all of the records but it's placed "NA" in the column
where it couldn't find a matching record
  dim(df_auto)
  dim(df_auto_hold)

  is.na(df_auto_hold$First_Year)

  View(df_auto_hold)

  df_auto_hold <- df_auto %>%
    left_join(df_auto_start, by="Car_Make") %>%
    group_by(Car_Make, First_Year) %>%
    summarise(Average_MPG=mean(MPG))

  View(df_auto_hold)
```

4.8 Cheat Sheet

We’re not going to cover every function here, but what is great about these packages is the the cheatsheets. The community has made to provide reference. I highly recommend downloading them and keeping them close:

- dplyr: https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
- stringr: https://github.com/rstudio/cheatsheets/blob/master/strings.pdf
- a whole lot more… https://www.rstudio.com/resources/cheatsheets/

4.9 Extra - sqldf

If you like SQL, there is a package allows you to manipulate data in a in-memory, or permanent database:

```
install.packages("sqldf")
library(sqldf)

df_auto_hold <- sqldf("select * from df_auto where Car_Make='amc'")
View(df_auto_hold)
```

We will learn a lot more about SQL in the Fall semester.