# R Workshop, Day 1

Toby Trotta, Applied Research Lab

Spring 2024

```
# This two-part workshop was created while employed as a Research Associate at IUP's
# Applied Research Lab. Examples created are all proprietary, but
# most information regarding R/RStudio IDE, programming concepts,
# and computer lingo is all more or less "public knowledge", or are from
# recycled class notes/materials from Edinboro University of PA. It is a (hopefully)
# straightforward crash course in the iceberg of R's capabilities.
#
# If this makes you more intrigued about R, RStudio, or the wonders of statistical
# language programming, then my goal has been met. Enjoy!
#
# Special thanks to:
#     Dr. Paul Hawkins (Director, IUP ARL);
#     Dr. Russ Stocker (IUP Thesis Advisor)
#     Dr. Marc Sylvester (EUP Academic Advisor)
#     Dr. John Hoggard (EUP Professor)*
#                 * that of whom I owe most thanks for
#                   building my foundation of R/RStudio! :D
#
#   Thank you all for your commitment to me and my success!
#
# -Toby Trotta, 2024
```

## Day 1: Introduction to R and RStudio, Directory Management, and Dataframes:

### What is R?

R is a **programming language** used for statistical analysis and data visualization. RStudio, on the other hand, is an open-source **integrated development environment (IDE)** for the R programming language; IDEs provide the necessary tools to *efficiently* develop code. We use "R" and "RStudio" synonymously, but they are different in theory.

(If you might like to work/take notes in RMarkdown, the file type we are using currently, you can *italicize* or **bold** using single or double asterisks.)

RStudio is comprised of four main windows:

- `Source`, which is where this is written
- `Console`, a text-based interface to execute (run) code – this is also where our output will populate. (includes `Terminal`, `Render` and `Background Jobs`; these are not important to the workshop and can be disregarded)
- `Environment`, which stores objects created during a **session** (more info!!!!)

- `Files`, show the available directories and all files (includes `Plots`, `Packages`, `Help`, `Viewer`, `Presentation`; the latter two will not be discussed)

## RProjects: mapping file explorer to Rstudio

RProject files are used to quickly open the currently stored settings, directory, and environment of a session. They do not require "saving", as it acts as the supervisor of all the moving parts. Creating an Rproj file can be done by: `File > New Project` ** Note: This is the "File" in the top banner, not the "Files" pane to the right.

## Directory Management:

Every file on your computer is stored in a *directory*. The directory acts as a file system cataloging structure allowing users to reference computer files and other directories. Directories can be simply associated with the folders in your File Explorer. (File Explorer, find Mac version)

The *working (current) directory* is the directory (ie. location) where the system will default to for accessing files read ("referenced", "called") in your Script or Markdown document. The working directory can be changed by command using `setwd(path)` – this method is much more challenging for novice programmers. However, simplified by RStudio, we may instead access `Session > Set Working Directory > Choose Directory...` in the top banner or using the keyboard shortcut `CTRL + SHIFT + H`.

Once the directory is set using any of the above methods, the `Files` pane to the right will update to show all files (photos, Word documents, R scripts, etc.) in the chosen directory.

Using `setwd("path")` can be easily referenced in RStudio under the `New Folder` icon in the `Files` pane. The path, for example, would look something like "C:/Users/tobyt/OneDrive/Documents/ARL/RWorkshop", with the complete command being `setwd("C:/Users/tobyt/OneDrive/Documents/ARL/RWorkshop")`. This takes a bit more practice, but isn't incredibly necessary if you have everything you need stored in a *project*.

## Basic Arithmetic:

Rstudio is powerful enough to run 2 + 2. Let's try it:

```
2 + 2
```

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

** A quick way to create a **chunk**, where the code can be executed, by using `CTRL + ALT + I` (`CMD + OPTION + I` on MacOS).

See in the console below, we have the expression that was executed `> 2 + 2 # CTRL + ENTER to run the current line` and the **output**: `[1] 4`

```
# Let's create some variables:
x <- 2
y <- 5
z <- x + y
```

See in the Environment to your right, we now have variables `x` and `y` stored for this session that we can use in any chunk throughout the document. Both `<-` and `=` can be used for assigning variables, but `<-` is the R standard. The keyboard shortcut is `ALT + -` for Windows (`OPTN + -` for Mac). Note that you have to be in a chunk for the shortcut to work correctly. (how to store changes between programs)

We can remove unwanted objects stored in the environment by using `rm(object_name)` in the console. Or, conveniently, right-click the variable in the environment and select "Delete" – these are the pros of IDEs.

```
# To see the output:
z
```

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

```
# or:
print(z)
```

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

We can of course apply the other arithmetic operators:

```
x - y # Subtraction
```

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

```
x * y# Multiplication
```

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

```
x / y # Division
```

```
## [1] 0.4
```

```
x ^ y # Exponentiation
```

```
## [1] 32
```

- If we want to run this entire chunk at once, `CTRL + SHIFT + ENTER` (Windows) or `CMD + SHIFT + ENTER` (MacOS).

Or we can apply logical operators:

```
x < y # Less than
```

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

```
x <= y # Less than or equal to
```

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

```
x >= z # Greater than or equal to
```

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

```
x == z # Is equal to
```

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

## Vectors:

We introduce an essential topic in mathematics, statistics, data analysis – a **vector**. Vectors are defined differently by discipline, but for purposes of this workshop, consider the following definition:

A *vector* is a one-dimensional array. To supplement the definition of "vector," an *array* is a collection of items or data, which can be stored in memory (via `<-` in R).

[[ In history, there wasn't a separate operator for equality testing '=='; so to mitigate any issues between the variable assignment operator, the arrow was elected as the most consistent operator for assignments. ]]

```
v1 <- c(1, 2, 3, 4, 5) # c() creates a list (can be either row or column)
v2 <- seq(5, 9, 1) # Both endpoints are included in the vector
```

Similar to before, we can apply both arithmetic and logic to vectors:

```
10 * v1
```

```
## [1] 10 20 30 40 50
```

```r
v1 < v2
```

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

In the event we need to remove an item from a vector, there are two primary methods:

```r
v1[-4] # Removes 4th item
```

```
## [1] 1 2 3 5
```

```r
v2[v2 != 5] # Removes all 5s using logic, read "items in v2 not equal to 5"
```

```
## [1] 6 7 8 9
```

```r
v2[seq(1, length(v2), 2)] # Remove every second item
```

```
## [1] 5 7 9
```

## Data Types

We can partition data into three primary categories:

- `numeric` (1.2, -35, 2.343e3)
  - `integer` (-3, 10, 43)
  - `double` (-7.18, 3.14, 5)
  - `float` (-7.18, 3.14, 5)
  - `complex` (2 + 2i, 1 - 3i)
- `character`, aka `string` ("t", "I love R", "2")
- `boolean`, aka `logical` (binary, dichotomous) (TRUE, T, FALSE, F)

**Q:** What's the difference between a `double` and a `float`?

**A:** They are both used to represent real number values (ie. not complex, including the imaginary term i). A `double` eliminates computational errors when rounding; it is the preferred numeric type.

**Q:** What's the difference between an `integer` and a `double`?

**A:** An integer is a whole number, both positive and negative. A double can be any positive or negative whole number or decimal.

**Q:** If the data is a `double`, then it is an `integer`. True or False?

**A:** False. 4.25 is a double but not an integer.

**Q:** If the data is an `integer`, then it is a `double`. True or False?

**A:** True. 7, an integer, can be expressed as 7.0.

What type of data is the vector?

```r
class(v1)
```

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

## Creating Data Frames

(You likely wont do this LOL)

We can either create a dataframe in R or import an existing data frame (with Excel extension `.csv` or a `.txt` file).

I find the simplest way (although there are numerous methods to creating an empty data frame) is first to create an empty **matrix** then convert to a data frame.

```
mat <- matrix('', ncol = 3) # '' is the empty string
emptydf <- data.frame(mat) # advantages of df vs matrix

# The functions matrix and data.frame are included in base R (no special packages need installed or loa
```

Before we continue, let's discuss how to navigate the cells within a matrix (can be applied exactly the same way for data frames):

When programming, we access a dataframe's (or matrix's) cells by describing the location of the value. For example, consider the following matrix (this can be associated with selecting a cell in Excel)

```
mat <- matrix(c(1:9), nrow = 3)
df <- data.frame(mat)
```

*Note:* By default, when using the `matrix()` function, items are stored vertically and move across columns. If this is how you wish to format your data, you can stop here. However, it's more convenient for users to input data (using `c()`) horizontally and have it presented the same way:

If we instead wanted 1 2 3 across the first row, we include the argument `byrow = TRUE`:

```
mat1 <- matrix(c(1:9), nrow = 3, byrow = TRUE)
df1 <- data.frame(mat1)
```

---

How can I access the `1` entry in `df1`? We see it is located at the intersection of the first row and first column. To "retrieve" the value, we use brackets: `dataframe[row#, col#]`

***Note:*** If you are familiar with programming, it is common for matrix/dataframe indices to start at `0`. With R, indices start at `1`. One will argue `0` is superior, but we don't start counting at 0, do we?... LOL

```
df1
```

```
##   X1 X2 X3
## 1  1  2  3
## 2  4  5  6
## 3  7  8  9
```

```
df1[1, 1]
```

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

What if we want the entire first row?

```
df1[1, ] # In this case, we don't specify which column we want. It will grab all entries in the first r
```

```
##   X1 X2 X3
## 1  1  2  3
```

And similarly for the first column?

```
df1[, 1] # Now we don't specify the row number. Note: It is presented as a "row," but stored as a colum
```

```
## [1] 1 4 7
```

Given the values stored in `df1`, at what location is `3` stored? Try and get the output:

```
df1[1, 3]
```

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

Try printing the second column:

```
df1[, 2]
```

```
## [1] 2 5 8
```

**Challenge:** Use the values stored in `df1` to find 3 * 2:

```
df1[1, 3] * df1[1, 2]
```

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

Now we have a 3 (row) by 3 (column) data frame. Of course the default headers `X1`, `X2`, and `X3` are not very descriptive. These could be any data type and represent many different things. That said, we can change our column headers to accurately represent what is stored in the column:

We want to change the column names for data frame 1:

```
colnames(df1)
```

```
## [1] "X1" "X2" "X3"
```

```
colnames(df1) <- c('RandomNum1', 'RandomNum2', 'RandomNum3')
df1
```

```
##   RandomNum1 RandomNum2 RandomNum3
## 1          1          2          3
## 2          4          5          6
## 3          7          8          9
```

**Note:** We can find the data types listed under the variable names. `<int>` represents "integer", a type of numeric value.

To store by row, we state which row to add data to (`df[1, ]` represents the first row, all columns):

```
emptydf[1, ] <- c('Toby', 24, 'Applied Mathematics')
```

We obviously don't want to enter data row by row – that disregards the efficiency of RStudio. Instead, we can create separate lists of data and compile them into one data frame:

First, create lists to be included in your desired column:

Let's add some students:

```
names <- c('Toby', 'Hope', 'Grace')
ages <- c(24, 24, 20)
majors <- c('Applied Math.', 'Speech Pathology', 'Undeclared')
```

Now, instead of using the matrix approach, we can just store the data directly in the data frame:

```
df <- data.frame(names, ages, majors)
```

In this case, we don't have to worry about specifying which direction the data will be stored; each list will serve as a column.

## Importing Data Frames

If you have secondary data or data that is in a tabular format (something from Excel or Google Sheets), we can use the `tidyverse` library to import the data into R for analysis.

### What are "libraries"?

Libraries (or "packages") are collections of functions and compiled code in a well-defined format. The directory where the packages are stored is called the library. Packages can be installed using `install.packages("name`

of package"). Installing only needs done once per operating system. (Reference similarities in SPSS, Excel, etc.)

```
install.packages('tidyverse')
```

```
## Installing package into '/cloud/lib/x86_64-pc-linux-gnu-library/4.3'
## (as 'lib' is unspecified)
```

To use the package, use the command `library(name of package)` – note that with `library()`, we do not use quotations, but when installing, we do. Alternatively, we may use the `Packages` pane in the Files window in the bottom right and check the box of the package we'd like to use.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.0     v tibble    3.2.1
## v lubridate 1.9.3     v tidyr     1.3.1
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Libraries expand your capabilities in R tremendously! For now, we will just use `tidyverse` – the main package for importing and tidying data, but will be installing and using different packages later for our analyses. (What's included?)

**For Excel extensions:** `.csv` files are the most flexible and are widely used, making `read_csv()` likely the most useful to you. Data can be "read in" or "called" into use by using command: `data_csv <- read_csv("fileName.csv")`

`.xlsx` files are exclusive to Microsoft Excel and allow for different tabs and pages in the overall document. Some use `.csv` files for raw data and `.xlsx` for presented/sorted/cleaned data. In the event that your data is attached with the `.xlsx` extension, one can either `Save As` and change the extension to `.csv` or read in the data using the `readxl` package – which you may need to install before use: `data_xlsx <- read_excel("fileName.xlsx")`

**For `.txt` extensions:**

Text documents can be formatted many different ways. In a text file, we don't have the convenience of placing items into cells. Instead, we separate data using some form of punctuation: `,`, `.`, `:`, or `;`. or with a set distance between items. The punctuation and/or distance between data in a `.txt` file is called a **delimiter**.

See https://www.datacamp.com/tutorial/r-data-import-tutorial for steps to import other types of data (Databases, JSON, or HTML).

Try importing the file `testData.csv` into your workspace:

```
testData <- read_csv('testData.csv')
```

```
## Rows: 200 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (5): X1, X2, X3, X4, X5
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

**Q:** What data types are `x1`, `x2`, `x3`, `x4`, and `x5`? (Try and find it in the output above!)

**A:** All of the values in our testData dataframe are doubles.

Let's take a look at the data:

```
# Can either use:
testData
```

```
## # A tibble: 200 x 5
##       X1    X2    X3    X4    X5
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  150.  192.     1     1    17
##  2  151.  205.     1     1    18
##  3  151.  247.     1     1    18
##  4  151.  162.     1     1    17
##  5  151.  248.     1     1    17
##  6  153.  218.     1     1    17
##  7  153.  104.     1     1    17
##  8  153.  155.     1     1    18
##  9  154.  236.     1     1    17
## 10  154.  134.     1     1    18
## # i 190 more rows
```

```
# or
# View(testData) # This will open a separate window in the source pane and is easier to examine with la
```

Obviously our column headers aren't very informative. Using what we learned above, rename the column headers:

- x1 to be Height
- x2 to be Weight
- x3 to be Gender
- x4 to be Class
- x5 to be Age

```
colnames(testData) <- c('Height', 'Weight', 'Gender', 'Class', 'Age')
testData
```

```
## # A tibble: 200 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl>  <dbl> <dbl> <dbl>
##  1   150.   192.      1     1    17
##  2   151.   205.      1     1    18
##  3   151.   247.      1     1    18
##  4   151.   162.      1     1    17
##  5   151.   248.      1     1    17
##  6   153.   218.      1     1    17
##  7   153.   104.      1     1    17
##  8   153.   155.      1     1    18
##  9   154.   236.      1     1    17
## 10   154.   134.      1     1    18
## # i 190 more rows
```

## Data Manipulation:

Let's quickly pivot to the concept of **factors** (nominal, categorical, ordinal, boolean). A factor is a type of variable that allows us to classify and organize groups. For example, a clinical study may be testing the effects of a certain medication against a placebo. We could include a factor variable indicating `0` for patients

receiving the tested medication and 1 for patients who have received the placebo. As we'll discuss in Part 2 of this workshop, factors are useful in many statistical tests and models: two-sample T-tests, ANOVA, etc.

Factors are a special type of variable that we did not mention above, and we can specify which variables are factors. If we do this manually, variable by variable, we reference each column using `$` and apply `as.factor(df$Var)`:

```
testData$Gender <- as.factor(testData$Gender)
```

We've introduced some new notation: `$`. The dollar sign, included immediately after the name of the dataframe, allows us to reference an entire column, the name of which we tie to the dollar sign:

```
testData$Gender # Now that we have Gender as a factor, we have someting additional in our output: Level
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 1 0 0 0
##  [75] 1 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [112] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 1 1 1 1 0
## [149] 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1
## [186] 1 1 1 1 0 0 0 1 0 0 1 1 0 0 0
## Levels: 0 1
```

(***Note:*** We also use `$` to assign new variables not included in the original dataframe.)

Now, try turning our `Class` variable into a factor instead of a double. Check if you did it correctly!

```
testData$Class <- as.factor(testData$Class)
```

Aside from re-classifying our variable types (called **coercion**), three important and very useful commands are `select()`, `filter()`, and `mutate()` from the `dplyr` package.

Try installing and loading the `dplyr` package, if you have not already done so/have it installed:

```
install.packages('dplyr')
```

```
## Installing package into '/cloud/lib/x86_64-pc-linux-gnu-library/4.3'
## (as 'lib' is unspecified)
```

```
library(dplyr)
```

Now we have those manipulation commands available to us. What do they do?

- `select()` is pretty straightforward: this is what we use to select specific variables/columns.
- `filter()` is used to subset a data frame, retaining all rows that satisfy our conditions.
- `mutate()` is a bit more complicated, but allows us the freedom to perform arithmetic or apply logical operators in order to change (or create!) variables.

To implement these efficiently, we use something called a **pipe**. We can "pipe" dataframes into the command to specify which dataframe to perform the command on. The pipe operator is defined as `%>%` or `%<%` , two percent symbols sandwiching a directed arrow (less-than or greater-than sign). This works seamlessly with the three mutation functions above, allowing for cleaner, more compact code.

**Note:** Instead of typing the pipe out each time, we use the keyboard shortcut `CTRL + SHIFT + M` for Windows in a chunk (`CMD + SHIFT + M` for MacOS). This will, by default, use the right-ward facing pipe, `%>%`, which is typically the standard form.

### `select()`

Let's see how the `select()` function works:

```
# df %>% select(VariableName)
testData %>% select(Class)
```

```
## # A tibble: 200 x 1
##    Class
##    <fct>
##  1 1
##  2 1
##  3 1
##  4 1
##  5 1
##  6 1
##  7 1
##  8 1
##  9 1
## 10 1
## # i 190 more rows
```

If we want to select multiple variables, we create a list within the `select()` function:

```
testData %>% select(c(Class, Age))
```

```
## # A tibble: 200 x 2
##    Class   Age
##    <fct> <dbl>
##  1 1        17
##  2 1        18
##  3 1        18
##  4 1        17
##  5 1        17
##  6 1        17
##  7 1        17
##  8 1        18
##  9 1        17
## 10 1        18
## # i 190 more rows
```

**Note:** We do not include quotation marks around the variables. Some programming languages require this; R does not.

Try selecting the `Weight` and `Class` columns from our `testData` dataframe:

```
testData %>% select(c(Weight, Class))
```

```
## # A tibble: 200 x 2
##    Weight Class
##     <dbl> <fct>
##  1   192. 1
##  2   205. 1
##  3   247. 1
##  4   162. 1
##  5   248. 1
##  6   218. 1
##  7   104. 1
##  8   155. 1
##  9   236. 1
## 10   134. 1
## # i 190 more rows
```

### `filter()`

Let's see how the `filter()` function works:

Say we want all observations having a value in our Weight column less than or equal to 150:

```
testData %>% filter(Weight < 150)
```

```
## # A tibble: 68 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
##  1   153.   104. 1      1        17
##  2   154.   134. 1      1        18
##  3   157.   121. 1      1        17
##  4   157.   107. 1      1        18
##  5   159.   135. 1      1        18
##  6   160.   114. 1      1        17
##  7   160.   111. 1      1        18
##  8   162.   145. 1      1        18
##  9   163.   136. 1      1        17
## 10   164.   127. 1      1        18
## # i 58 more rows
```

If we classify our `Gender` variable as `0` being `Males` and `1` being `Females`, filter out all data for our imaginary female participants:

*Remember, we use logical operators in our `filter()` function. If we want to filter cases where a certain variable equals our criteria, use `==`:*

```
testData %>% filter(Gender == 0)
```

```
## # A tibble: 97 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
##  1   165.   123. 0      1        18
##  2   167.   112. 0      1        18
##  3   167.   138. 0      1        17
##  4   169.   138. 0      1        18
##  5   170.   196. 0      1        18
##  6   170.   189. 0      1        18
##  7   171.   150. 0      1        18
##  8   171.   142. 0      1        17
##  9   172.   175. 0      1        17
## 10   172.   153. 0      1        18
## # i 87 more rows
```

If we have multiple conditions that need satisfied simultaneously, we use `&` (and). For example, if we wanted all females in class 2, our syntax would be:

```
testData %>% filter(Gender == 1 & Class == 2)
```

```
## # A tibble: 15 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
##  1   153.   189. 1      2        20
##  2   157.   106. 1      2        18
##  3   159.   206. 1      2        20
##  4   160.   162. 1      2        19
##  5   161.   199. 1      2        19
```

```
##  6   164.   112. 1      2        19
##  7   167.   144. 1      2        18
##  8   168.   117. 1      2        20
##  9   169.   202. 1      2        19
## 10   174.   191. 1      2        20
## 11   174.   114. 1      2        20
## 12   175.   148. 1      2        20
## 13   176.   164. 1      2        20
## 14   176.   102. 1      2        20
## 15   178.   209. 1      2        20
```

For "and" conditions, both must be true. We have another logical operator `|` (or). If we wanted to filter all participants in classes 1 or 2, our syntax would be:

```
testData %>% filter(Class == 1 | Class == 2)
```

```
## # A tibble: 157 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
##  1   150.   192. 1      1        17
##  2   151.   205. 1      1        18
##  3   151.   247. 1      1        18
##  4   151.   162. 1      1        17
##  5   151.   248. 1      1        17
##  6   153.   218. 1      1        17
##  7   153.   104. 1      1        17
##  8   153.   155. 1      1        18
##  9   154.   236. 1      1        17
## 10   154.   134. 1      1        18
## # i 147 more rows
```

**Challenge:** Apply the logic where we filter intersection (and) of being in class 1 or class 2 and being female.

```
testData %>% filter((Class == 1 | Class ==2) & Gender == 1)
```

```
## # A tibble: 86 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
##  1   150.   192. 1      1        17
##  2   151.   205. 1      1        18
##  3   151.   247. 1      1        18
##  4   151.   162. 1      1        17
##  5   151.   248. 1      1        17
##  6   153.   218. 1      1        17
##  7   153.   104. 1      1        17
##  8   153.   155. 1      1        18
##  9   154.   236. 1      1        17
## 10   154.   134. 1      1        18
## # i 76 more rows
```

In the event we want to leave a class out but keep every other one, instead of having `filter(Class == 2 | Class == 3 | Class == 4)`, if we wanted to leave Class 1 out, we use `!` (exclamation point) to

```
testData %>% filter(Class != 1)
```

```
## # A tibble: 72 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl> <fct>  <fct> <dbl>
```

```
##  1   153.   189. 1      2        20
##  2   157.   106. 1      2        18
##  3   159.   206. 1      2        20
##  4   160.   162. 1      2        19
##  5   161.   199. 1      2        19
##  6   164.   112. 1      2        19
##  7   167.   144. 1      2        18
##  8   168.   117. 1      2        20
##  9   168.   114. 0      2        19
## 10   169.   123. 0      2        18
## # i 62 more rows
```

**Challenge:** Combining `filter()` and `select()`. Filter the observations for male participants with `Weight` less than or equal to 170. Then, select the Height and Weight.

*Hint:* You will need to use two pipe operators. The order in which you use `filter()` and `select()` matters! Why?

```
testData %>% filter(Gender == 0 & Weight <= 170) %>% select(c(Height, Weight))
```

```
## # A tibble: 54 x 2
##    Height Weight
##     <dbl>  <dbl>
##  1   165.   123.
##  2   167.   112.
##  3   167.   138.
##  4   169.   138.
##  5   171.   150.
##  6   171.   142.
##  7   172.   153.
##  8   173.   118.
##  9   174.   169.
## 10   174.   156.
## # i 44 more rows
```

### mutate()

The mutate command/function is useful in many applications. The options are limitless, but consider you wanted to standardize your data or re-group your observations (this would be called "re-coding your variables").

To start, let's do a simple example. Let's create a new column "Random" that scales our Weight column by 10:

```
# Syntax: df %>% mutate(NewVarName = Equation)
testData %>% mutate(Random = 10 * Weight)
```

```
## # A tibble: 200 x 6
##    Height Weight Gender Class   Age Random
##     <dbl>  <dbl> <fct>  <fct> <dbl>  <dbl>
##  1   150.   192. 1      1        17  1921.
##  2   151.   205. 1      1        18  2049.
##  3   151.   247. 1      1        18  2469.
##  4   151.   162. 1      1        17  1617.
##  5   151.   248. 1      1        17  2482.
##  6   153.   218. 1      1        17  2182.
##  7   153.   104. 1      1        17  1043.
```

```
##  8   153.   155. 1      1        18  1553.
##  9   154.   236. 1      1        17  2356.
## 10   154.   134. 1      1        18  1340.
## # i 190 more rows
```

```
# You'd generally want to store this new data frame (with the mutated column) into a new dataframe:
testData2 <- testData %>% mutate(Random = 10 * Weight)
```

## Recoding Variables:

If we wanted to re-code our variables (let's say we want `Male` to be 1 and `Female` to be 2, instead of the previous `0-1` definition). It's a bit more complicated, but if we include some logic, it simplifies our code:

### Recode into Same Variable

```
# Syntax: df %>% mutate(SameVar = ifelse(test = (Criteria), yes = Condition if True, no = Condition if
testData %>% mutate(Gender = ifelse(test = (Gender = 0), yes = 1, no = 2))
```

```
## # A tibble: 200 x 5
##    Height Weight Gender Class   Age
##     <dbl>  <dbl>  <dbl> <fct> <dbl>
##  1   150.   192.      2 1        17
##  2   151.   205.      2 1        18
##  3   151.   247.      2 1        18
##  4   151.   162.      2 1        17
##  5   151.   248.      2 1        17
##  6   153.   218.      2 1        17
##  7   153.   104.      2 1        17
##  8   153.   155.      2 1        18
##  9   154.   236.      2 1        17
## 10   154.   134.      2 1        18
## # i 190 more rows
```

**Note:** When using `mutate()`, we must be careful. Note that the data type has been changed back into a `dbl` with this manipulation. We would need to coerce this, using `as.factor(df$Var)`, back into a factor.

### Recode into New Variable:

We recommend this so you don't lose any previous data. This is important for large-scale projects, and is especially important if you are tasked with reporting and documenting every manipulation.

```
# Syntax: df %>% mutate(NewVar = ifelse(test = (Criteria), yes = Condition if True, no = Condition if F
testData %>% mutate(Gender2 = ifelse(test = (Gender = 0), yes = 1, no = 2))
```

```
## # A tibble: 200 x 6
##    Height Weight Gender Class   Age Gender2
##     <dbl>  <dbl> <fct>  <fct> <dbl>   <dbl>
##  1   150.   192. 1      1        17       2
##  2   151.   205. 1      1        18       2
##  3   151.   247. 1      1        18       2
##  4   151.   162. 1      1        17       2
##  5   151.   248. 1      1        17       2
##  6   153.   218. 1      1        17       2
##  7   153.   104. 1      1        17       2
##  8   153.   155. 1      1        18       2
##  9   154.   236. 1      1        17       2
```

```
## 10   154.   134. 1      1        18       2
## # i 190 more rows
```

The syntax would be read as: "If Gender is 0, change that number to 1. If it is not 0, change it to 2." Again, this mutation coerced the new data column to a double, so we would need to change this again.

Of course the more conditions you have, or if you have more variables you'd like to change at once, we would include nested `ifelse()` statements. I think this is a bit beyond the scope, but if you were curious, we would add another `ifelse()` function under the `no =` argument in the previous `ifelse()` statement.

### Renaming Factors:

First, we must check the pre-defined order of the factors (ie. our factors order may not be sequential). To do so, we use the command `levels(df$FactorVariable)`:

```
levels(testData$Class)
```

```
## [1] "1" "2" "3" "4"
```

We see from the output that our factors are ordered 1, 2, 3, 4.

Note: The sequential ordering may not be the case; you may have performed some manipulation that ends up reordering the factors, so it is imperative we check the ordering with `levels()`.

To rename, store a list into `levels(df$FactorVariable)`in the same order they are listed. If we define 1 to be Freshman, 2 to be Sophomore, 3 to be Junior, and 4 to be Senior:

```
levels(testData$Class) <- c('Freshman', 'Sophomore', 'Junior', 'Senior')
```

Now we see that all the factors now reflect the newly defined factor names:

```
levels(testData$Class)
```

```
## [1] "Freshman"  "Sophomore" "Junior"    "Senior"
```

```
testData
```

```
## # A tibble: 200 x 5
##    Height Weight Gender Class      Age
##     <dbl>  <dbl> <fct>  <fct>    <dbl>
##  1   150.   192. 1      Freshman    17
##  2   151.   205. 1      Freshman    18
##  3   151.   247. 1      Freshman    18
##  4   151.   162. 1      Freshman    17
##  5   151.   248. 1      Freshman    17
##  6   153.   218. 1      Freshman    17
##  7   153.   104. 1      Freshman    17
##  8   153.   155. 1      Freshman    18
##  9   154.   236. 1      Freshman    17
## 10   154.   134. 1      Freshman    18
## # i 190 more rows
```

Try doing the same with our Gender variable; recall 1 is now Male and 2 is now female. Check the ordering to make sure we will not accidentally swap Males and Females!

```
# Check ordering:
levels(testData$Gender)
```

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

```
# Rename levels:
levels(testData$Gender) <- c('Male', 'Female')
```

```r
# Check the updated data:
head(testData)
```

```
## # A tibble: 6 x 5
##   Height Weight Gender Class      Age
##    <dbl>  <dbl> <fct>  <fct>    <dbl>
## 1   150.   192. Female Freshman    17
## 2   151.   205. Female Freshman    18
## 3   151.   247. Female Freshman    18
## 4   151.   162. Female Freshman    17
## 5   151.   248. Female Freshman    17
## 6   153.   218. Female Freshman    17
```

With this manipulation, the factors remain factors; we do not need to coerce. This is the preferred method for recoding **factors**.

## Exercises

For this section, instructions will be provided, but chunks will not be. (Do you remember how to create a chunk?) Let's try the following exercises!

1) Subset (either using `filter()`, `select()` or both) your dataset:

a) Create two datasets, one for the male participants and another for the female participants. Store these in appropriately named dataframes.

```r
# With factors that are characters (instead of 0 for Male or 1 for Female), we must include quotations.
male <- testData %>% filter(Gender == 'Male')
female <- testData %>% filter(Gender == 'Female')
```

b) Now, create 4 datasets, one for each class standing. Store these in appropriately named dataframes.

```r
freshmen <- testData %>% filter(Class == "Freshman")
sophomores <- testData %>% filter(Class == "Sophomore")
juniors <- testData %>% filter(Class == "Junior")
seniors <- testData %>% filter(Class == "Senior")
```

2) Manipulating Data:

a) Using the original dataset (testData.csv), assume the Height column is in centimeters. Use the appropriate `dplyr` function (`select()`, `filter()`, or `mutate()`) to convert the measurements to feet (1 ft = 30.48 cm). Store this in the original Height column.

```r
testData %>% mutate(Height = Height / 30.48)
```

```
## # A tibble: 200 x 5
##    Height Weight Gender Class      Age
##     <dbl>  <dbl> <fct>  <fct>    <dbl>
##  1   4.94   192. Female Freshman    17
##  2   4.94   205. Female Freshman    18
##  3   4.94   247. Female Freshman    18
##  4   4.96   162. Female Freshman    17
##  5   4.96   248. Female Freshman    17
##  6   5.01   218. Female Freshman    17
##  7   5.03   104. Female Freshman    17
##  8   5.03   155. Female Freshman    18
##  9   5.06   236. Female Freshman    17
## 10   5.06   134. Female Freshman    18
```

```
## # i 190 more rows
```

b) Consider the following scenario: The person who collected this data inappropriately reported the weight due to a faulty scale, over-reporting the individuals' weights by 25 lbs. Perform the appropriate mutation and store the corrected dataframe as "cleanedData".

```
cleanedData <- testData %>% mutate(Weight = Weight - 25)
```

## Exporting Data:

To wrap up the day, I want to share a nice tool for exporting data from RStudio. The syntax is relatively straightforward: `write.csv(dataFrame, "dataFrame.csv")`, where the first argument is the dataframe stored in your session's workspace, and the second argument is the exported name of the file. Once this is complete, you should see the new `x.csv` file in your Files pane to the right!

The third argument `row.names` is optional; it requires either TRUE (T) or FALSE (F). This is useful for not adding an additional column for just the row number. The row number is likely something you aren't interested in; you may have an ID column, for survey data, for example, but that will remain in the data set as it was a defined variable.

```
write.csv(cleanedData, 'cleanedData.csv', row.names = FALSE)
```

(I always include `row.names = FALSE`.)

And finally, we have the option to save our workspace environment. To do this, we can just hit the floppy "Save workspace as" button, to the left of "Import Dataset". This will automatically save all working variables and dataframes from today. It's generally good practice to save your workspace, perhaps every session or maybe once a week – depends how much work you do in that day.

Why do this? This isn't a necessary step. A convenient part of R is that it will prompt you upon exiting the application if you want to save your workspace. If you are using an R project to house your work, your workspace will be saved for your next session. However, your workspace environment can become lengthy and jumbled, especially if you have items or variables with similar names. It's all personal preference! For purposes of this workshop, let's save our workspace; we will use some of the variables and dataframes we created today in Part 2.