Data Wrangling Part 1

Workspace / Project

- RStudio will save your current workspace to `.Rdata` file. This includes:
- All the objects loaded on environment
- Your file on text editor
- You can jump back to your project by opening `.Rdata` file
- This can be useful, but do not rely too much! Your script should do these works too.

Coding Style

- Having and maintaining a consistent coding style is always helpful
  - Readability
  - Help you revising code

Few Rules

- Put spaces between and around variable names and operators (=+−\∗/)
- Break up long lines of code
  - If 2-3 line is enough, break
  - If more than that, break lines for function arguments
- Use meaningful variable names composed of 2 or 3 words" (avoid abbreviations unless they're very common and you use them very consistently)
- Naming: snake_case or CamelCase
- Indentation (2 or 4 spaces)
- Comments!
- Google's R Style Guide https://google.github.io/styleguide/Rguide.html
- Tidyverse Style Guide https://style.tidyverse.org

Working Directory

- Each time you open R, it links itself to a directory -> Working Directory
- WD is the base directory for R to browse files.
  - If you want to use relative path, that should be based on WD
- Workspace / Project -> where your `.Rdata` is
- `getwd()`
- Default -> where your script is
  - `~/Documents/Rcamp2026/test.R` -> WD is `~/Documents/Rcamp2026/`

Change the working directory

- You might want to change WD
- `setwd(new_path)`
- Personally, I do not recommend, but you may use this to ensure WD
- `setwd()` may not work properly for Rmarkdown
  - It only **temporarily** changes working directory for *a code chunk*
  - `knitr::opts_knit$set(root.dir = '{PATH}')`
  - The best practice: Do not change if you really have to

List files in the working directory

- You can see what files are in your working directory
- `list.files()`
- This can be useful when you want to do file level operations; open each file inside a loop
  - I would do it in Python
- If you are familiar with UNIX commands, `ls` has a different role here: View environment

Create a new directory

- `dir.create(path)`

```
# This creates a new directory inside the current directory at path = /User
dir.create("./pset2")
```

In-class exercises:

1. Get your current working directory.

Loading and Saving Data

- Usually, we start by loading data
  - Surveys, downloaded from an organization, etc.
- Different file formats require different functions, so we briefly touch one few of them

Load: Built-in Datasets

- Base R and some packages come with default data sets
- You can see a list of base R's data sets

```r
help(package = "datasets")
```

- To use these default data set, just type its name

```r
iris
mtcars
```

Load: Native R Data Formats

- R provides two file formats of its own for storing data: `.RDS` and `.RData`
- Smaller spaces than csvs and
- Faster to read and write.
- RDS files can store *a single R object*
- RData files can store *multiple R objects*.
- `readRDS()` and `load()`
- `(load("file.RData"))`: sneak peek the objects contained

```
school_loc <- readRDS("stuff.RDS")
load("stuff.RData") # This will load the entire workspace including environ
```

Load: CSV Files

- `read.csv()`
- `read_csv()` from `readr` package (a member of `tidyverse`)

```
school_loc <- read.csv("school_loc.csv")
```

Load: HTML Links

- R can take files from urls
- For example:

```
school_loc <-
  read.csv(
    "https://data-nces.opendata.arcgis.com/datasets/
    a15e8731a17a46aabc452ea607f172c0_0.csv?outSR=%7B%
    22latestWkid%22%3A4269%2C%22wkid%22%3A4269%7D"
  )
```

Load: Stata file .dta

- To load it, we need external package called `haven`
- We will touch on external package the other day, but here is a sample code

```r
# install.packages("haven") # install it first if it has not been
library("haven") # load the pacakge

df <- read_dta('./path/file.dta')
```

Save: Native R Data Formats

- `saveRDS()` and `save()`
- Do not forget to put file extensions!

```r
a <- 1
b <- 2
c <- 3
saveRDS(a, file = "stuff_a.RDS")
save(a, b, c, file = "stuff.RData")
load("stuff.RData")
a2 <- readRDS("stuff_a.RDS")
```

Save: CSV Files

- `write.csv()`
- Again, extensions!
- `row.names` = argument decides to save row index / names as a separate column

```
# write.csv(r_object, file = filepath, row.names = FALSE)
write.csv(school_loc, "./data/school_loc.csv", row.names = FALSE)
```

In-class exercises:

1. Load the world total fertility rate dataset from dataset.csv. Name the dataframe TFR.
2. Create a new directory named data.
3. Save the dataframe as “newdata.csv” into the new directory that you just created.

Data Wrangling Part 1

- We have covered
  - how to create R object (assign operator, <-)
  - Major data types (numeric, integer, character, boolean, and factor)
  - Major data structures (vector, list, dataframe, and matrix)
- Let's learn how we can select and manipulate values of those!

Selecting Values

- `[ row , column ]`
- 1-base indexing: Intuitive!
- Positive integers
- Negative integers
- Zero
- Blank spaces
- Logical values
- Names

Positive integers

- `ij` notation used in linear algebra:
  - `school_loc[i,j]` the ith row and the jth column

```
# Extract value at the 1st row and the 2nd column
school_loc[1,2]
```

```
## [1] 34.78337
```

- Again, 1 is the base here!
- Index by a *vector*

```
school_loc[1, c(1,2,3)]
```

```
##          X        Y OBJECTID
## 1 -86.5685 34.78337        1
```

Positive integers

- We can assign the subset of the data as a separate object

```
newdf <- school_loc[1, c(1,2,3)]
head(newdf)
```

```
##           X        Y OBJECTID
## 1 -86.5685 34.78337        1
```

- Repeating a number in your index -> Duplicate

```
newdf <- school_loc[c(1,1), c(1,2,2,3)]
head(newdf)
```

```
##             X        Y      Y.1 OBJECTID
## 1   -86.5685 34.78337 34.78337        1
## 1.1 -86.5685 34.78337 34.78337        1
```

Positive integers

- A single column in a data frame -> a vector

```
school_loc[1:2, 1] # this is a vector
```

```
## [1] -86.56850 -86.79935
```

- Prefer a single column to be a data frame? `drop = FALSE`

```
school_loc[1:2, 1, drop = FALSE] # this is a dataframe
```

```
##           X
## 1 -86.56850
## 2 -86.79935
```

- You can use the same syntax to select values in any R object

```
vec <- c(2,4,6,8,10)
print(vec[1:3])
```

```
## [1] 2 4 6
```

Negative integers

- Exclusion

```
school_loc[-1, 1:3]
school_loc[-(2:52), 1:3]
school_loc[c(-1,-3), 1, drop=F]
```

- We cannot pair a negative integer with a positive integer in the *same index*

```
# Exclude the first column and then include it again??
school_loc[c(-1, 1), 1]
```

Zero

- creates an empty object

```
# data frame with 0 columns and 0 rows
x <- school_loc[0, 0, drop = F]
dim(x) # 0 0
```

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

Blank spaces

- Include ALL

```
# This extracts the first row
school_loc[1, ]
# This extracts the first column
school_loc[ ,1, drop=F]
```

Logical values

- A vector of TRUEs and FALSEs as index
- R will then return rows / columns that correspond to TRUE.

```
dim(school_loc) # [1] 7012   26
```

```
## [1] 7012   26
```

```
# The first row, The first and the third cols
school_loc[1, c(TRUE, FALSE, TRUE, rep(FALSE, 23))]
```

```
##          X OBJECTID
## 1 -86.5685        1
```

Names

- If there are names, you can call them

```
# If you do not remember the exact names of each column,
# you can first use the function colnames() to list all the column names.
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

```
school_loc[1:10, c("X", "Y", "NAME", "SCHOOLYEAR")]
```

```
##            X        Y                                   NAME SCHOOLYEAR
## 1  -86.56850 34.78337             Alabama A & M University  2020-2021
## 2  -86.79935 33.50570 University of Alabama at Birmingham  2020-2021
## 3  -86.17401 32.36261                   Amridge University  2020-2021
## 4  -86.64045 34.72456 University of Alabama in Huntsville  2020-2021
## 5  -86.29568 32.36432             Alabama State University  2020-2021
## 6  -87.52959 33.20702 University of Alabama System Office  2020-2021
## 7  -87.54598 33.21188            The University of Alabama  2020-2021
```

In-class exercises:

Now that you know the basics of R's notation system, let's put it to use.

1. Extract all values from the first ten rows from school_loc
2. Extract the 100th row from school_loc
3. Extract the 200th to 300th rows from school_loc, but only from the following columns: NAME, STREET, CITY, STATE, ZIP.

Dollar Signs and Double Brackets

- dataframes and lists accepts $
- Select a column from a data frame

```r
sub_school_loc$NAME
```

- Useful for column level operations

```r
mean(school_loc$X)
```

```
## [1] -90.35801
```

```r
median(school_loc$Y)
```

```
## [1] 38.54617
```

```r
max(school_loc$Y)
```

```
## [1] 71.3247
```

Dollar Signs and Double Brackets

- You can use the same $ notation with the elements of a list, if they have names
- This works as double brackets, [[]]

```
l <- list(numbers = c(1, 2), logical = TRUE, strings = c("a", "b", "c"))

l[[1]] # access by index
```

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

```
l$numbers # same
```

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

```
l[1] # not same
```

```
## $numbers
## [1] 1 2
```

Changing Values in Place

- Select the values then assign new values

```
vec <- c(1:10) # initial value
vec <- "text" # new value
```

And here is how you can modify it:

```
vec <- 1:10
vec[1] <- 999
vec
```

```
##  [1] 999   2   3   4   5   6   7   8   9  10
```

```
vec[-1] <- 0
vec
```

```
##  [1] 999   0   0   0   0   0   0   0   0   0
```

Changing Values in Place

- Vectors? Of course!

```
vec <- 1:10
vec[c(1, 3, 5)] <- c(1000, 3000, 5000)
vec
```

```
##  [1] 1000    2 3000    4 5000    6    7    8    9   10
```

```
vec[4:6] <- vec[4:6] + 1
vec
```

```
##  [1] 1000    2 3000    5 5001    7    7    8    9   10
```

Changing Values in Place

- You can also create values that do not yet exist in your object.
- R will expand the object to accommodate the new values:

```
vec <- c(1:10)
vec[12] <- 0
vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 NA  0
```

Changing Values in Place

- Create new col in a dataframe

```
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

```
school_loc$indices <- 1:7012 # This creates a new col
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR" "indices"
```

Changing Values in Place

- Remove columns from a data frame by assigning them the symbol NULL:

```
school_loc$indices <- NULL
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

Changing Values in Place

- Change values of a subset of a dataframe

```
school_loc$newcol <- 1:7012
school_loc$newcol[c(1,3,5,7,9)] <- c(111, 333, 555, 777, 999)
school_loc$newcol[c(2,4,6,8,10)] <- 1000
head(school_loc$newcol, 10)
```

```
##  [1]  111 1000  333 1000  555 1000  777 1000  999 1000
```

Logical Subsetting

Operator	Syntax	Tests
`>`	`a > b`	Is a greater than b?
`>=`	`a >= b`	Is a greater than or equal to b?
`<`	`a < b`	Is a less than b?
`<=`	`a <= b`	Is a less than or equal to b?
`==`	`a == b`	Is a equal to b?
`!=`	`a != b`	Is a not equal to b?
`%in%`	`a %in% c(a, b, c)`	Is a in the group c(a, b, c)?

Figure 1: R's Logical Operators

Logical Subsetting

- *element-wise comparisons* and return a logical vector

```
1 > c(0, 1, 2)
```

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

```
c(1, 2, 3) == c(3, 2, 1)
```

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

Logical Subsetting

- `%in%` is the only basic operator that does 'not' do normal element-wise execution

```
1 %in% c(3, 4, 5)
```

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

```
c(1, 2) %in% c(3, 4, 5)
```

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

```
c(1, 2, 3) %in% c(3, 4, 5)
```

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

```
c(1, 2, 3, 4) %in% c(3, 4, 5)
```

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

Logical Subsetting

- `%in%` is the only basic operator that does 'not' do normal element-wise execution

```
# works for strings
"a" %in% c("a", "b", "c")
```

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

```
"A" %in% c("a") # case sensitive
```

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

```
"a" %in% c("abc") # %in% checks exact matches
```

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

Logical Subsetting

- You can compare any two R objects with a logical operator;
- If you compare objects of different data types, R will use its coercion rules
- To sum up:
  - `object[ row_condition, column_condition ] <- new_values`

In-class exercises:

Let's practice with the dataset TFR.

1. Find the dimension of TFR. How many observations and variables?
2. Add a new column called index to TFR, where the values are the indices for the rows.
3. Select the first row of TFR.
4. Select the first column of TFR.
5. Get the last row of TFR.
6. How many values in the column TFR are greater than 8?
7. How many values in the column ChildBearing are over 25?
8. What does this code do?

```
TFR$ChildBearing[TFR$ChildBearing > 25]
```

9. Set all values in the column ChildBearing that are less than 25 to 0.

In-class exercises:

```r
TFR <- read.csv("dataset.csv")
nrow(TFR[TFR$TFR > 8,])
```

```
## [1] 67
```

Boolean Operators

Operator	Syntax	Tests
`&`	`cond1 & cond2`	Are both `cond1` and `cond2` true?
`|`	`cond1 | cond2`	Is one or more of `cond1` and `cond2` true?
`xor`	`xor(cond1, cond2)`	Is exactly one of `cond1` and `cond2` true?
`!`	`!cond1`	Is `cond1` false? (e.g., `!` flips the results of a logical test)
`any`	`any(cond1, cond2, cond3, ...)`	Are any of the conditions true?
`all`	`all(cond1, cond2, cond3, ...)`	Are all of the conditions true?

Figure 2: Boolean Operators

Boolean Operators

- R will execute each logical test and then use the Boolean operator
- element-wise execution

```
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 4)
a == b
```

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

```
b == c
```

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

```
a == b & b == c
```

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

```
all(a == b & b == c)
```

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

```
any(a == b & b == c)
```

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

In-class exercises:

If you think you have the hang of logical tests, try converting these sentences into tests written with R code. To help you out, I've defined some R objects after the sentences that you can use to test your answers:

1. Is w positive?
2. Is x greater than 10 and less than 20?
3. Is object y the word February?
4. Is every value in z a day of the week?
5. What does this expression evaluate to and why?

In-class exercises:

```
w <- c(-1, 0, 1)
x <- c(5, 15)
y <- "February"
z <- c("Monday", "Tuesday", "Friday")
(TRUE + TRUE) * FALSE
```

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

In-class exercises:

6. Use logical operators to output only those rows of data in TFR where column Year is between 1950 and 1955 inclusively.
7. Use logical operators to output only those rows of data where column TFR is equal to 7.45 and column Year is greater than 1960.
8. Use logical operators to output only the even rows of the dataframe.
9. Use logical operators and change every 4th element in column LifeExpB to 0.

In-class exercises:

```r
data <- TFR
# 6
head(data[data$Year >= 1950 & data$Year <= 1955,])
```

```
##       Country Uncode Year  TFR InfMRateCME InfMRateUN U5MRateCME U5MRate
## 1 Afghanistan      4 1950 7.45          NA   304.5940         NA  438.01
## 2 Afghanistan      4 1951 7.45          NA   299.6836         NA  431.60
## 3 Afghanistan      4 1952 7.45          NA   294.7732         NA  425.20
## 4 Afghanistan      4 1953 7.45          NA   289.8628         NA  418.79
## 5 Afghanistan      4 1954 7.45          NA   284.9524         NA  412.39
## 6 Afghanistan      4 1955 7.45          NA   280.0420         NA  405.99
##   LifeExpB MtoFbirth    MtoF04  Pop1564 Pop1564Female GDPpc GDPpcGrowth
## 1  26.0690      1.06 0.9523352 56.08875      54.25678    NA          NA
## 2  26.5736      1.06 0.9524428 55.82908      54.06208    NA          NA
## 3  27.0782      1.06 0.9728808 55.74039      54.08136    NA          NA
## 4  27.5828      1.06 0.9952082 55.71038      54.13613    NA          NA
## 5  28.0874      1.06 1.0122050 55.71779      54.22245    NA          NA
## 6  28.5920      1.06 1.0222970 55.68319      54.21412    NA          NA
##   Yschooling YschoolF1549 GenrollPrim ChildBearing CountryCode
## 1      0.270   0.08679564          NA       29.835         AFG
## 2      0.278   0.08758149          NA       29.835         AFG
## 3      0.286   0.08836733          NA       29.835         AFG
```

Missing Data (NA)

- you don't know a value
- The `NA` character is a special symbol in R

```
1 + NA
```

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

```
NA == 3.14
```

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

Inside functions: na.rm argument

- Many functions provide argument na.rm = TRUE/FALSE

```
c(NA, 1:50)
```

```
##  [1] NA  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 2
## [26] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 4
## [51] 50
```

```
mean(c(NA, 1:50)) # Error
```

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

```
mean(c(NA, 1:50), na.rm = TRUE)
```

```
## [1] 25.5
```

```
mean(c(1:50)) # which returns the same value as above
```

```
## [1] 25.5
```

Test for NA: is.na()

- You may want to identify the NAs in your data set
- A crude way:

```
c(1, 2, 3, NA) == NA
```

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

- A better way: is.na()

```
vec3 <- c(1, 2, 3, NA)
is.na(vec3)
```

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

```
# What does this do?
school_loc$Y[school_loc$X < -100] <- NA
```

Omit the Entire Rows: `na.omit()`

- `na.omit()`: Remove *rows* containing `NA`'s in **any** column

```
dim(school_loc)
summary(school_loc) # 7012 obs.
new_df <- na.omit(school_loc)
dim(new_df)
summary(new_df) # 5470 obs
```

In-class exercises:

1. What is the length of X after the following operations?

```
X <- c (123,0,NA,8,NA,200)
na.omit(X)
```

```
## [1] 123   0   8 200
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "omit"
```

2. Can you find all occurrences of NA in X? How can you find the total number of NAs in X?

3. Can you remove all occurrences of NA in X?

4. Can you replace all occurrences of NA with 88?

In-class exercises:

5. We will use the TFR dataset again. Create a new dataframe TFR2 where you drop all NA's.
6. Find the dimension of TFR2. How many observations are left?
7. Create a new dataframe TFR3 where you remove all rows with NA values in the GDPpc column.

```
TFR3 <- TFR[!is.na(TFR$GDPpc), ]
```