CMP

COMMERCE MENTORSHIP PROGRAM

FINAL REVIEW SESSION

COMM 205

Prepared by: Benjamin Gerochi

TABLE OF CONTENTS

1. R Programming Basics

2. R Data Types

3. R Data Frame

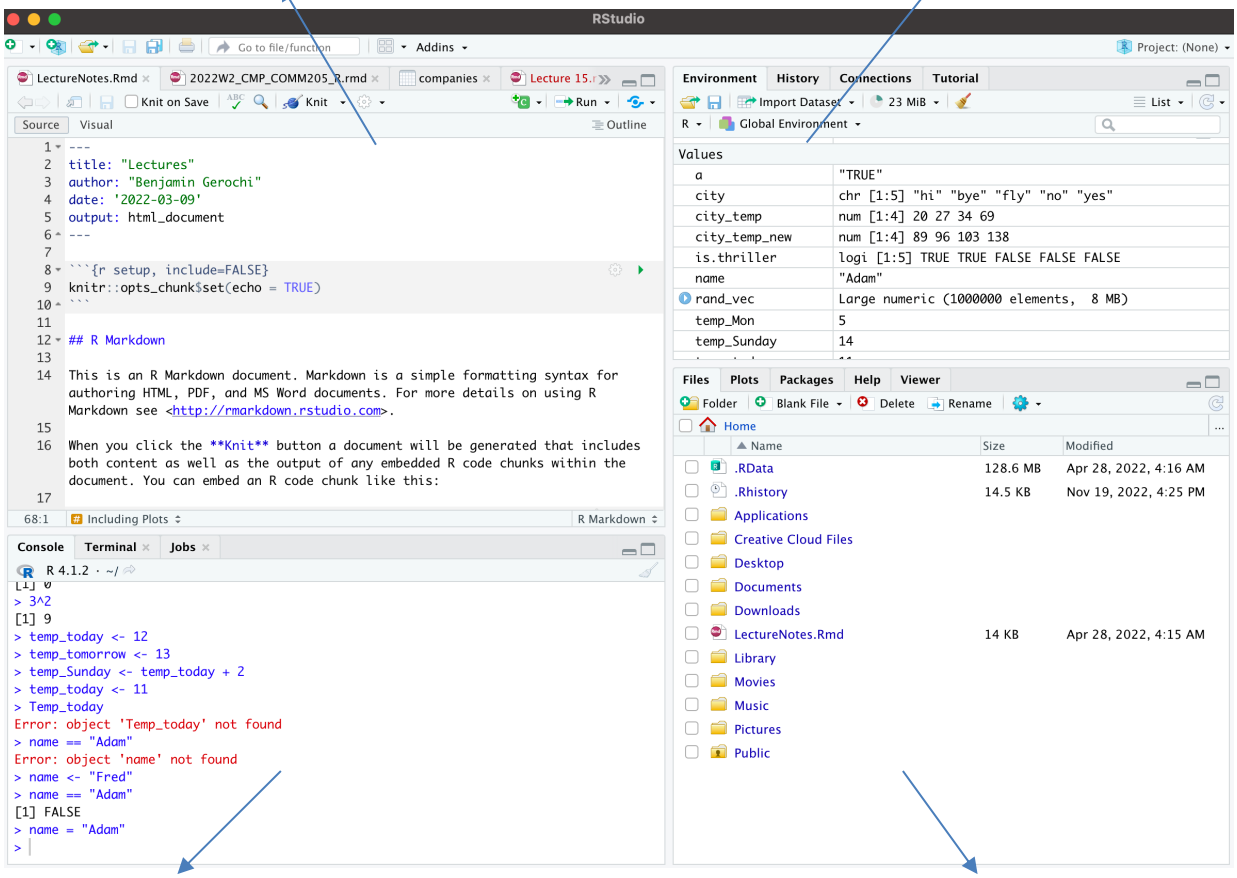
4. Data Wrangling

5. Merging Datasets

R Programming Basics

R Extension

Environment



Console

Files/Plots/etc.

- **R Extension:** where to write/execute code, e.g. in Markdown
- **Console:** see outputs of executed code (can also code here)
- **Environment:** see objects that are created
- **Files/Plots/etc:** navigate directories, help documents

- **R is case sensitive** (functions must be lower case)
- **Comments:** put a **#** before the words/phrase (these will not be executed as code)
- Highlight code, hit **command/ctrl + enter** to execute
- Execute **library(tidyverse)** for the functions we will use

R Programming Basics

- **Operators**
  - **arithmetic:** +, -, *, /, ^
  - **logical:** ==, !=, >, <, >=, <=
  - **other:** & (and), | (or), ! (not)
- **Assigning values to Objects**
  - use a left arrow **<-** to assign objects (= also works)
  - these will appear on the **environment**

```
rating_movie <- 15
rating_show <- 12
rating_Thor <- rating_movie + 5
```

rating_movie	15
rating_show	12
rating_Thor	20

- Executing an object on Markdown/Console outputs its value
- Code is executed **in order** and values are **dependent on what was executed**
  - changing rating_movie to 16 will **NOT** automatically change rating_Thor to 21
- **Naming Objects**
  - combination of letters, periods, digits, or underscores
  - must start with a **letter or period**; if starting with a period, cannot be followed by a digit
  - **cannot use** reserved words (e.g. TRUE, FALSE, NA, if, etc.)
    - use ?reserved to see the list of reserved words
- **Functions**
  - function(argument/input) = output
  - use ?function_name to get information on it
  - example: **rm(object)**
    - removes an object from the Environment

R Data Types

- **Double:** default numeric type, can include decimals
- **Integer:** whole number type
- **Character:** any combination of characters/symbols (has quotation marks in R language)
- **Logical:** TRUE or FALSE

- **typeof(x) function:** returns the data type of the argument
  - e.g. typeof(7) = double
  - argument can be an object

- **Implicit Coercion**
  - R coerces inappropriate data types to appropriate ones
  - If there are multiple data types within a vector, R will coerce them to a **character** data type
    - order of coercion: logical -> integer -> double -> character
  - Logical data types are coerced to double data types
    - **TRUE** is equivalent to **1**
    - **FALSE** is equivalent to **0**

- **Examples**
  - vector1 <- c("Benjamin", 5, FALSE)
    - typeof(vector1) = **"character"**
  - TRUE + 8 = **9**, TRUE + FALSE + 17 = **18**
  - cannot add a character and a double
    - "Benjamin" + 8 will result in an **error**

R Data Types

- **Vectors**
  - set of multiple values of the **same** data type
    - can be assigned to an object
    - arguments in a c() function
- **length(x)** function: determines how many elements are within a vector
  - **Examples:**
    - **doublevector <- c(7, 8, 9, 10)**
    - **charactervector <- c("A", "B", "C", "D")**
    - length(doublevector) = 4
- **Subsetting**
  - retrieving specific elements from a vector
  - use of indexing (numerical position in a vector)
    - doublevector[1] = **7**
    - doublevector[1:3] = **7, 8, 9**
    - doublevector[c(2, 4)] = **8, 10**
  - can also specify elements **NOT** to retrieve
    - doublevector[-2] = **7, 9, 10**
    - doublevector[-c(2, 4)] = **7, 9**
  - a subset of a vector can be assigned to a new object
    - sub_vector <- doublevector[2:3]
    - sub_vector = **8, 9**
  - can also alter the elements in a vector
    - doublevector[1] <- 0
    - doublevector = **0, 8, 9, 10**
  - or add to an existing vector
    - doublevector[5:7] <- c(11, 12, 13)
    - doublevector = **7, 8, 9, 10, 11, 12, 13**

R Data Types

- **Calculations with Vectors**
  - performing arithmetic operations on numerical vectors applies it to every element
  - **Examples:**
    - doublevector + 10 = **17, 18, 19, 20**
- **which(x)** function: returns the indexes (numerical positions) in a vector which satisfy the given condition (x)
  - where x is a logical argument, e.g. doublevector > 8
  - can see a vector as a column and its elements as the values in its rows
  - **Examples:**
    - which(doublevector > 8) = **3, 4**
  - you can subset a which function to get values in those positions
    - charvector[which(doublevector > 8)] = **C, D**

- **Basic R functions** (can be performed on vectors)
  - max(x) – maximum value of x
  - min(x) – minimum value of x
  - range(x) – minimum and maximum values of x
  - length(x) – number of elements in x
  - sum(x) – sum of all the elements in x
  - mean(x) – average value of all the elements in x
  - sd(x) – standard deviation of the elements in x
  - var(x) – variance of the elements in x
  - sqrt(x) – gets the square root of every element in x
  - median(x) – median of the elements in x
  - n_distinct(x) – count of distinct elements in x

R Data Frame

- **Libraries**
  - some useful functions come in packages, which are libraries of code written by the R user community
  - to install a package, execute:
    - **install.packages("name_of_package")**
  - to load the contents of the package, execute:
    - **library(name_of_package)**
    - must be executed every time a new R session is run
  - We execute **library(tidyverse)** to use functions in the tidyverse package! (do this every time you run R)
- **Data Frame**
  - R object to store a collection of vectors (variables) that have elements (observations)
  - all vectors in a data frame must have the same number of elements
  - a table with:
    - **columns -> variables**
    - **rows -> observations**
  - to construct a data frame, execute:
    - **tibble(...[column_name1] = vector1, ...)**
  - **Examples:**
    - data_frame <- tibble(

      column1 = c("This", "is", "a", "sentence"),

      column2 = c(1, 2, 3, 4),

      column3 = c(TRUE, FALSE, FALSE, TRUE)

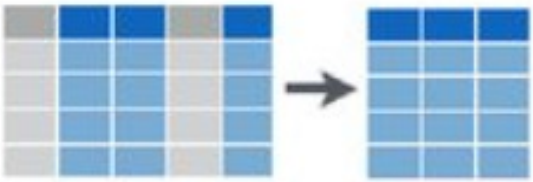
)
  - to view a data frame (seen as a table):
    - **view(dataframe_name)**

R Data Frame

- to access **a specific element** from the data frame using its row and column indexes
  - dataframe_name[row_position, column_position]
  - dataframe_name[row_position, "column_name"]
- to access **a column** from the data frame
  - The $ operator extracts a column of a data frame as a vector
  - **dataframe_name$column_name**
  - **Examples:**
  - you can perform calculations on them
    - data_frame$column1[1:3] = **"This", "is", "a"**
    - data_frame$column2 > 2 = **FALSE, FALSE, TRUE, TRUE**
    - data_frame$column * 2 = **2, 4, 6, 8**
  - you can also **add** another column or **alter** an existing one
    - data_frame$column4 <- c(5, 6, 7, 8)
    - data_frame$column2 <- data_frame$column2 * 2
    - data_frame$column5 <- data_frame$column2 / data_frame$column2 (visualize the created column)
- **Missing Values**
  - represented as **NA** (Not Available) in R
  - can exist in any data type; is not in quotation marks
  - any arithmetic operations or functions done on NA elements will **result in NA**
  - **Examples:**
    - 1 + NA = NA
    - NA > 5 = NA
    - missing <- c(1, 2, 3, 4, NA)
    - **mean(missing) = NA**

R Data Frame/Data Wrangling

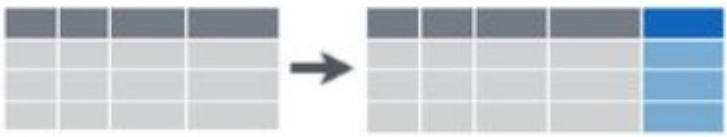
- Some of the basic R functions (mean, sum, length, etc.) have an option called **na.rm**
  - **na.rm** is a logical value indicating whether NA values should be ignored before proceeding with the calculation
  - they are set to FALSE by default – meaning NA values are not ignored
  - **function(x, na.rm = TRUE)**
  - **e.g.** mean(missing, na.rm = TRUE) = **2.5**
- **is.na(x)** function: checks for NA values
  - produces **TRUE (or 1)** if an element is NA
  - produces **FALSE (or 0)** if an element is not NA
  - **is.na(missing) =** FALSE, FALSE, FALSE, FALSE, TRUE
  - **sum(is.na(x))** can be used to check how many missing values are in x
- **na.omit(dataframe_name)**: removes **ALL** NA values from a data frame
- **Data Wrangling (dplyr functions)**
  - process of transforming and mapping data
  - load the companies dataset!
- **companies <- readRDS("~/Downloads/North_American_Stock_Market_1994-2013.rds")**
- **select(x)**: picks variables (columns) based on their names
- **select(dataframe_name, column_name1, …)**
  - e.g. select(data_frame, column1, column2)



R Data Wrangling

- **filter(x)**: picks observations (rows) based on logical conditions
- **filter(dataframe_name, condition1, condition2, …)**
  - TRUE rows are kept, FALSE/not TRUE are removed
  - you can create a new table with the operated values
  - e.g. **new_dataframe <- filter(data_frame, column1 == "sentence" & column2 > 2)**
  - can also use | (or) or , to separate
  - **!is.na(x)** can be used as a logical condition to filter out NA for certain variables
    - **e.g. filter(!is.na(missing))** = retain rows w/ non-missing values for missing

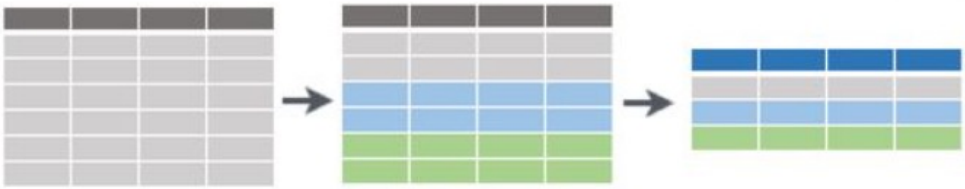
- **Pipe Operator (%>%)**
  - makes code more readable
  - when used, Data Wrangling functions take data from the previous step as their input
  - e.g. **filter(data_frame, column1 == "sentence")** is the same as **data_frame %>% filter(column1 == "sentence")**
  - works as one line; we put functions in the next for readability
- **mutate(x):** adds a new variable(s) based on existing data
- **mutate(data_frame_name, new_variable_name = operation or aggregate function(existing_variable_name), …)**
  - e.g. mutate(data_frame, column4 = column3/column2)



R Data Wrangling

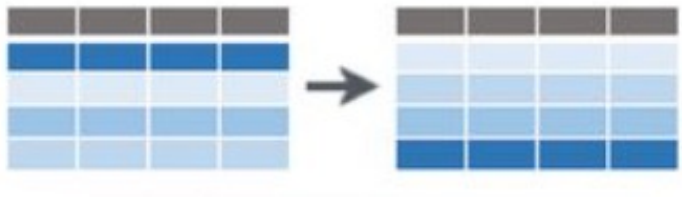
- **summarise(x):** creates an aggregate statistic over all observations (or subgroup, if specified with group_by function)
- **summarise(dataframe_name, new_variable_name = operation or aggregate_function(existing_variable_name))**
  - some examples of **aggregate functions** (not limited to): mean, median, sd, min, max, sum, n(), n_distinct()
  - **n(x) function**: same as length, but for use in DW functions
  - outputs one observation (# of groups if done by group)
  - no need to do a select() after summarise:
    - the summarise() function **drops all columns** except the new ones created in summarise() or the variable it is grouped by

- **round(x):** rounds numbers to the nth decimal place
  - **round(value_to_round, n)**
- **group_by(x)**: groups observations based on one or more variables
  - **group_by(dataframe_name, group_variable(s))**
  - allows user to perform functions on a subset of the data
  - converts it to a grouped dataframe, where subsequent operations are performed by group
  - mutate and summarise often used with group_by to perform aggregate functions in groups



R Data Wrangling/Merging Datasets

- **group_by with 2 variables**
  - groups by 2 variables and operates on that subset of data
    - e.g. "firms in the USA (loc) in each year (fyear)"
  - ideally, remove NA observations from variables that will be used in group_by
- **arrange(x)**: orders rows based on the specified variables
- **arrange(dataframe_name, arrange_variable(s))**
  - default is lowest to highest
  - works alphabetically (A-Z) or numerically (-inf – inf)
  - **desc(x)** function on a variable to arrange the variable in descending order
  - subsequent arrange variables are dependent of each other



- **Merging Datasets**
  - matching observations from two datasets based on matching values
  - variable names of the columns in both datasets should be the same for it to work
  - recommended: data types are also the same

ID	X1
1	a1
2	a2

ID	X2
2	b1
3	b2

Merging Datasets

- **inner_join(x, y)**: returns all rows from x where there are matching values in y, and all columns
  - where x and y are data frames
  - if specifying merged variables:
    - **merged_dataset <- inner_join(x, y, by = c("merge_variable1", "merge_variable2"))**

ID	X1	X2
2	a2	b1

- **left_join(x, y)**: returns all rows from x, and all columns from x and y
  - where x and y are data frames
  - to match the variables (if names are not equivalent):
    - **merged_dataset2 <- left_join(x, y, by = c("left_variable_name" = "right_variable_name")**
  - if variables to merge are not specified, R will merge common variables by default

ID	X1	X2
1	a1	NA
2	a2	b1

- **Other Commands**
  - **if_else(x)**: if function, similar to Excel!
    - **if_else(logical_condition, value_if_true, value_if_false)**
  - changing data types:
  - **as.character(x)**
    - converts a numeric object to a character object
  - **as.numeric(x)**
    - converts a character object to a numeric object
    - removes leading 0s