Helpful R Commands

R code allows you to do many things: get data, manipulate data, conduct statistical analyses, and save data. It can take years before you really feel comfortable writing your own syntax. Most people will memorize basic R functions, but they will not memorize everything. They will however keep samples of code that they can refer to and they learn where to look in the R help documentation for the code that they need. The more you write your own code the easier it will get.

Getting Help with R

1. Type a ? and the command name in the console window and press Ctrl+Enter (e.g., ?describe)
2. Use Google, Stack Overflow, or the R Studio community
3. Use Rcmdr to point and click and save script (or Rmarkdown) file.
4. Ask for help! Ask Dr. Rocconi, the course TA, or another student for help.

Things to Know About R Syntax

- A function is a command that requires arguments, often listed in the function's parentheses
- Commands can appear on multiple lines
- It is good practice to indent subcommands after the first line (e.g., 2 spaces)
- R is case sensitive, with most functions being in either lowercase, camel case (e.g. functionName), Pascal case (e.g., FunctionName), or snake case (e.g. snake_case)
- Hashtag/pound (#) signifies a comment line; very useful to organize your code
    - A comment followed by four dashes (----) will be listed in R's document outline (https://www.natedayta.com/2019/12/25/owning-outlines-in-rstudio/)
- You can run code by:
    - Highlighting and pressing play or Ctrl+Enter
    - Putting your mouse inside the code and pressing play or Ctrl+Enter
- When you get an error message look for mistakes: Missing or additional quotation marks; Missing periods; Missing parentheses marks; Invalid variable names or variable values; Forgetting to include Execute; Spelling mistakes in commands. *Errors will happen; try not to get too frustrated and think of it as a learning experience.*

Helpful Hints

- Organize your work in R Projects to avoid setting a working directory
- Name and date your files in a consistent manner and keep these recorded in a data codebook
- Resave your Datafile using a new name periodically (e.g., after a major set of analyses/ modifications). This way you can always go back to the original dataset if you made a mistake
- Become used to installing and loading multiple packages

Some Useful Commands

- Getting data into R
  - load()
    Loads an R dataset (.RData)
  - **`read.csv()`**
    Loads a csv file
  - The haven package imports and saves out SPSS, Stata, and SAS datasets
    - **`haven::read_sas()`**
      Opens SAS datasets (**`haven::write_sas`** saves data as SAS file)
    - **`haven::read_stata()`**
      Opens Stata datasets (**`haven::write_dta`** saves data as a Stata file)
    - **`haven::read_spss()`**
      Opens SPSS datasets (**`haven::write_sav`** saves data as a SPSS file)
  - The psychTools package will also load SPSS, .csv, .txt, .dat and data from the clipboard into R
    - **`psychTools:read.file()`**
- **`install.packages()`**
  Installs R packages. Package name must be in quotes (e.g., **`install.packages("psych")`**). This only needs to be done ONCE.
- **`library()`**
  Opens an R package. Package name does not need to be in quotes (e.g., **`library(psych`**).
- **`::`**
  Loads a specific function from a package (e.g., **`package::function`**)
- **`<-`**
  Assigns any file, command, or result to a data object.
- **`#`**
  Comment function. Useful to organize and label code.
- **`save(x, y, z, file="name.Rdata")`**
  Save objects as an .Rdata file.
- **`write.csv(x, file="name.csv")`**
  Save an object as a CSV file
- **`function(x, y){…code…}`**
  Used to create an R function. A function is useful for automating repeated tasks.

  There are two parts to a function:

(1) *Arguments*: x, y, z… the inputs to the function.
(2) *Body*: The body of the function contains R code for the tasks that are to be performed.

```
fahrenheit_to_celsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

Common Tidyverse Commands

- **`%>%`**
  The "pipe". It allows you to connect multiple functions together. It stands for "and then".

- **`rename()`**
  This command allows you to rename one or more variables in your dataset

  ```
  rename("new name" = "old name")
  ```

- **`mutate()`**
  Creates or changes a variable. With this function, you can add together the values of two or more variables; create composite variables that are summed or averaged from other variables in your dataset.

  ```
  data %>%
     mutate(variable_to_change = variable_to_change*100,
            new_variable = variable_1 * variable2)
  ```

- **`dplyr::na_if(x, y)`**
  Allows you to designate specific values in a variable as missing values. This should be used inside a mutate() function.

  ```
  data %>%
     mutate(variable = na_if(variable, 999)
  ```

- **`recode()`**
  Allows you to recode a variable into a new variable. Useful for changing string variables into numeric variables; reverse score (change lowest to highest, etc.) the values for a variable; replace missing data for a variable with a specific value (say a mean or a norm for that variable). For instance, recode is useful when you have negatively worded items and you need to reverse score prior to creating composites or conducting reliability tests. This should be used inside a mutate() function.

  ```
  data %>%
     mutate(new_variable = recode(variable, 1=0, 2=1))

  data %>%
     mutate(new_variable = recode(variable, "text"=0, "text2"=1))
  ```

- **`view() or dplyr::view()`**
  View a data set in a spreadsheet-like viewer. This can be added to the end of a dplyr chain to view the result of your work before assigning to an object

```
data %>%
   mutate(new_variable = recode(variable, “text”=0, “text2”=1)) %>%
   view()
```

- **`left_join()`**
  This command allows you to merge two or more datafiles by one or more similar variables that they both have in common (called a key variable). This is useful if you have two datafiles for the same participant (say a pretest and posttest file) and you want to merge by participant (id)
- There are different types of joins, including left, right, inner, and anti.

```
dataset_a %>%
  left_join(dataset_b, by=”id”)

dataset_a %>%
  left_join(dataset_b, by=c(“id” = “identification”) #when there are two
  similar variables but they have different names
```

- **`count`**
  Allows you to create a new variable that contains a count of a specific value for variable in your dataset. This is very useful if you want to quickly check to see how often “prefer not to answer” or “not applicable” was checked for a series of variables OR if you want to look to see if any participants checked all extreme responses for a series of questions.
- This will create a summary dataframe with two columns, the factors or numbers you are counting and *n*, the number

```
data %>% count(variable)
```

- **`filter()`**
  Used to select a subsample of cases. This command allows you to select only certain participants based on specific values of a variable or variables in the dataset. This can be useful when you want to conduct analyses on just a subset of your datafile.

```
data %>%
      filter(variable > 1000)
```

See Base R Cheat Sheet for common Base R commands (also available on Canvas)**.**

See Syntax Comparison Cheat Sheet for a comparison between Base R ($ syntax), Formula syntax (~ syntax), and Tidyverse syntax (%>%).

Note: Base R will be getting a native pipe operator in a future release “|>”