Module 1.2

Getting started: managing data

TWS-WS 2022

NOTE: this module borrows heavily from an R short course developed by a team at Colorado State University. - Thanks to Perry Williams for allowing us to use these materials!!

Load script for module #1.2

1. Click here to download the script! Save the script to your working directory (project directory you set up in the previous submodule).

2. Load your script in RStudio. To do this, open RStudio (load the RStudio project you created in the previous submodule) and click on the folder icon in the toolbar at the top and load your script.

Let's get started with data management in R!

Working directory

The working directory is the first place R will look for any files you would like to read in (e.g., code, data). It is also the first place R will try to write any files you want to save.

It makes things a lot easier if you put all your data files into a single folder and tell R to make that folder your **working directory**. That way, you won't need to wrangle complex directory names!

Every R session has a working directory, whether you specify one or not. Let's find out what my working directory is right now:

```r
# Find the directory you're working in
getwd()          # note: the results from running this command on my machine will differ from yours!
```

```
## [1] "C:/Users/Kevin/Documents/GitHub/R-Bootcamp"
```

What's yours? Usually the default working directory is your "Documents" folder.

Since you should already be working in an RStudio Project, your working directory will automatically be set as your project directory (directory that contains the .Rproj file for your project). This is convenient, because:

1. That's most likely where the data for that project live anyway

2. If you're collaborating with someone else on the project (e.g., in a shared Dropbox folder), you can both open the project and R will instantly know where to read and write data without either of you having to reset the working directory (even though the directory path is probably different on your two machines). This can save a lot of headaches!

Reminder: to start a new RStudio Project, just click on "File->New Project" in RStudio's menu bar.

Set your working directory

You can easily set a new working directory using the "setwd()" function.

```
# for example...

# setwd("E:/GIT/R-Bootcamp")   # note the use of forward slash- backslashes for file paths, as used by
```

NOTE: when you put file paths in R, they need to use forward slashes ("/"; or double backslashes, "\\") – single backslashes ("\", as seen in Windows) do not work for specifying file paths in R.

Alternatively, you can (in RStudio) use the dropdown menus at the top to set the working directory (Session->Set Working Directory->Choose Directory).

Once you have set the working directory, you can use the "list.files()" function to see what's in the directory. If I ran this, we would see the contents of the directory that I'm using to create this website!

```
# Contents of working directory
list.files()
```

What's in your working directory?

Importing data into R!

There are many ways data can be imported into R. Many types of files can be imported (e.g., text files, csv, shapefiles). And people are always inventing new ways to read and write to/from R. But here are the basics.

- read.table
  - reads a data file in any major text format (comma-delimited, space delimited etc.), you can specify which format (very general)

- read.csv
  - fields are separated by a comma (this is still probably the most common way to read in data)

Before we can read data in, we need to put some data files in our working directory!

1. Download the following data files and store them in your working directory (i.e., the folder where your scripts are already!)

   - Whitespace delimited data file

   - Comma delimited data file

2. Make sure the data files are stored in your working directory (project directory)!

Once you have saved these files to your working directory, open one or two of them up (e.g., in Excel or a text editor) to see what's inside.

Now let's read them into R!!

```
####
####  Import data files into R
####

# ?read.table    # some useful functions for reading in data

# read.table to import textfile (default is sep = "" (one of several common delimiters, including any t
data.txt.df <- read.table("data.txt", header=T, sep="")

# read.table with csv file
data.csv.df <- read.table("data.csv", header=T, sep=",")

# built-in default for importing from CSV (easiest and most widely used)
data.df <- read.csv("data.csv")
names(data.df)

# Remove redundant objects from memory
rm(data.txt.df)
rm(data.csv.df)
```

Using R's built in data

R has many useful built-in data sets that come pre-loaded. You can explore these datasets with the following command:

```
# Built-in data files
data()
```

Let's read in one of these datasets!

```
# read built-in data on car road tests performed by Motor Trend
data(mtcars)

str(mtcars)    # examine the structure of this data object

# ?mtcars        # learn more about this built-in data set
```

Basic data checking

To learn more about the 'internals' of any data object in R, we can use the "str()" (structure) function:

```
####
#### Check/explore data object
####


# ?str: displays the internal structure of the data object
str(data.df)


## 'data.frame':    20 obs. of  4 variables:
##  $ Country: chr  "Bolivia" "Brazil" "Chile" "Colombia" ...
```

```
##  $ Import : int  46 74 89 77 84 89 68 70 60 55 ...
##  $ Export : int  0 0 16 16 21 15 14 6 13 9 ...
##  $ Product: chr  "N" "N" "N" "A" ...
```

As we've already seen, we can use the "head()" function to see the first few rows of our data (or the "tail()" function to see the last few rows):

```
# ?head: displays first n elements of object (default=6)
head(data.df)
head(data.df,2)

# ?tail: displays last n elements of object (default=6)
tail(data.df)
```

And we can use the "summary()" function to get a brief summary of what's in our data frame:

```
summary(data.df)
```

```
##     Country              Import          Export          Product
##  Length:20          Min.   :35.0   Min.   : 0.00   Length:20
##  Class :character   1st Qu.:66.0   1st Qu.: 3.00   Class :character
##  Mode  :character   Median :74.0   Median : 8.00   Mode  :character
##                     Mean   :72.1   Mean   : 9.55
##                     3rd Qu.:84.0   3rd Qu.:15.25
##                     Max.   :91.0   Max.   :23.00
```

Exporting and saving data in R

Reading in data is one thing, but you will probably also want to write data to your hard disk. There are countless reasons for this- you might want to use an external program to plot your data, you might want to archive some simulation results.

Again, there are many ways to write data to a file. Here are the basics!

Exporting data

```
####
#### Exporting data (save to hard drive as data file)
####

# ?write.table: writes a file to the working directory

write.table(data.df[,c("Country","Product")], file="data_export.csv", sep=",", col.names=TRUE, row.names
```

Saving (and loading) data

The data loaded in your **R environment** are stored in your computer's memory as **binary** representations that are efficient but not particularly human-readable (this is how computers store and manage data). But sometimes the lack of human readability isn't a problem. For example: what if all we want to do is save the

```

data so that we can load it back into an R session at a later date? In this case we never need to look at the data *outside* of R.

To do this we can use the "save()" and "load()" functions in R:

NOTE: all binary R data files should be stored with the extension ".RData":

```
####
Saving and loading
####

?save: saves particular objects to hard disk

a <- 1
b <- data.df$Product

save(a,b,file="Module1_2.RData")

rm(a,b)   # remove these objects from the environment

load("Module1_2.RData")   # load these objects back in!
```

NOTE: If you're using an RStudio Project (.Rproj), then the contents of your environment are saved when you close the project (if you choose that option when closing the project), and automatically loads again when you open the R Project file. This allows you to pick up where you left off!

**Clearing the environment**

Sometimes your environment can get cluttered with objects. In these cases, it can help to clear the environment. Let's try this!

```
##############
# Clear the environment (and load it back in!)

save.image(file="Module2.RData")     # ?save.image: saves entire environment (we don't necessarily want

rm(list=ls())    # clear the entire environment. Confirm that your environment is now empty!

load(file="Module2.RData")   # load the objects back!
```

Working with data in R

Now let's start seeing what we can do with data in R. Even without doing any statistical analyses, R is very a powerful environment for doing data transformations and performing mathematical operations.

Boolean operations

Boolean operations refer to TRUE/FALSE tests. That is, we ask a question about the data that either is true or false.

First, let's meet the boolean operators.

NOTE: don't get confused between the equals sign ("="), which is an alternative *assignment operator* (same as "<-"), and the double equals sign ("=="), which is a boolean operator:

```
# <- assignment operator
# =  alternative assignment operator

a <- 3      # assign the value "3" to the object named "a"
a = 3       # assign the value "3" to the object named "a"
a == 3      # answer the question: "does the object "a" equal "3"?
```

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

```
####
####  Boolean operations
####

#####
# Basic operators

# <    less than
# >    greater than
# <=   less than or equal to
# >=   greater than or equal to
# ==   equal to
# !=   not equal to
# %in% matches any one of a specified group of possibilities


#####
# Combining multiple conditions

# &    must meet both conditions (AND operator)
# |    must meet one of two conditions (OR operator)

Y <- 4
Z <- 6

Y == Z  #I am asking if Y is equal to Z, and it will return FALSE
Y < Z

!(Y < Z)  # the exclamation point reverses any boolean object (read "NOT")

# Wrong!
data.df[,2]=74     # sets entire second column equal to 74! OOPS!!!

data.df <- read.csv("data.csv")  ## correct our mistake in the previous line (revert to the original da

# Right
data.df[,2]==74     # tests each element of column to see whether it is equal to 74
```

Data subsetting using boolean logic

Boolean operations are great for subsetting data - that is, we can select only those rows/observations that meet a certain condition (where [some condition] is TRUE).

The "which()" function returns the indices of a vector (i.e., the specific elements) that meet a certain specified condition. Here's how it works:

```
####
####  Subsetting data
####

##########
# The "which()" function -- return indices of a vector that meet a certain condition (condition is TRUE

which(data.df[,2]==74)        # elements of column 2 that are equal to 74


## [1]   2 17

which(data.df[,2]!=74)        # elements of column 2 that are NOT equal to 74


##  [1]   1   3   4   5   6   7   8   9 10 11 12 13 14 15 16 18 19 20

which(data.df[,2]<74)         #  and so on...


## [1]   1   7   8   9 10 11 12 15 18

which((data.df[,2]<74)|(data.df[,2]==91))    # use the logical OR operator


##  [1]   1   7   8   9 10 11 12 15 18 20

#########
# Alternatively, save the indices as an R object as an intermediate step

indices <- which(data.df[,2]<74)
data.df[indices,2]                # same as above!


## [1] 46 68 70 60 55 35 51 68 73

#########
# Alternatively, you can omit the "which()" function entirely-
#     R will know that you're trying to omit "FALSE" elements and keep "TRUE" elements

data.df[data.df[,2]<74,2]     # alternative simpler syntax without using "which()"...


## [1] 46 68 70 60 55 35 51 68 73

sub.countries<-c("Chile","Colombia","Mexico")     # create vector of character strings

which(data.df$Country=="Chile")


## [1] 3

# This command doesn't work because sub.countries is a vector, not a single character string.
which (data.df[,1]==sub.countries)
```

```
## Warning in data.df[, 1] == sub.countries: longer object length is not a multiple of shorter
## object length
```

```
## integer(0)
```

When we want all elements of a vector that match one of a group (vector) of possibilities, we use the "%in%"
operator:

```
# Instead we use %in%
which(data.df[,1] %in% sub.countries)        # elements of data column that match one or more of the eleme
```

```
## [1]  3  4 14
```

```
which((data.df$Country %in% sub.countries) & (data.df$Product=="N"))    # use AND operator
```

```
## [1]  3 14
```

```
# What if we don't want the row number, but the actual row(s) of data that meet a particular condition?

indices <- which(data.df$Country %in% sub.countries)
data.df[indices,]              # print all data for a subset of observations
```

More data processing!

Let's switch to a different data file. Download the Turtle Data and save to your working directory (R project
directory).

Let's use this data set to review the summarizing and subsetting operations we have learned- and learn a
few new things along the way:

Subsetting data and correcting errors

```
####
####   Practice subsetting a data frame
####

turtles.df <- read.table(file="turtle_data.txt", header=T, stringsAsFactors = FALSE)
names(turtles.df)
```

```
## [1] "tag_number"      "sex"             "carapace_length" "head_width"      "weight"
```

```
head(turtles.df)

# Subset for females
fem.turtles.df <- turtles.df[which(turtles.df$sex =="female"),]

# Can subset just one factor based on another
# Here we want to know the mean weight of all females
mean(fem.turtles.df$weight)
```

```
## [1] 10.02857
```

```
mean(turtles.df$weight[which(turtles.df$sex =="female")])
```

```
## [1] 10.02857
```

```r
# Try using the "subset()" function, which can make subsetting a data frame even easier!
female.turtles <- subset(turtles.df,sex=="female")              # use the "subset()" function!
female.turtles <- turtles.df[which(turtles.df$sex=="female"),]    # alternative

subset.turtles.df <- subset(turtles.df, weight >= 10)
head(subset.turtles.df)

# Subsetting to certain individuals
bad.tags <- c(13,105)
good.turtles.df <- turtles.df[!(turtles.df$tag_number %in% bad.tags),]
bad.turtles.df <- turtles.df[turtles.df$tag_number %in% bad.tags,]
bad.turtles.df
```

Data manipulation using subsetting

We can use subsetting operations to manipulate (alter) data- for example, to correct some data elements:

```r
####
####  Data Manipulation using subsetting
####

# Using subset to correct data entry problems
unique(turtles.df$sex)
```

```
## [1] "male"   "female" NA       "fem"
```

```r
table(turtles.df$sex)    # do you notice the data entry problem here?
```

```
##
##    fem female    male
##      1      7       7
```

```r
turtles.df[which(turtles.df$sex=="fem"),2]<-"female"            # recode the sex variable

 # make a new variable "size.class" based on the "weight" variable
turtles.df$size.class <- NA
turtles.df$size.class[turtles.df$weight >= 6] <- 1        # ths is the "adult" class
turtles.df$size.class[turtles.df$weight < 6] <- 2         # this is the "juvenile" class

turtles.df$size.class
```

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

Sorting/ordering data

Sorting is another common data operation, which helps to visualize and organize data. In R, sorting is typically accomplished using either the "order()" or the "sort()" functions:

sort: sorts a vector literally – i.e., "10,3,22" becomes "3,10,22" (increasing, by default)
order: returns the indices of the original (unsorted) vector in the order that they would appear if properly sorted (increasing, by default) – i.e., "10,3,22" becomes "2,1,3".

Here's how you do it in R, continuing with the turtle data example:

```
####
####  Sorting
####

# Sort works for ordering along one vector
sort(turtles.df$carapace_length)
```

```
##  [1] 24.3 28.0 28.7 29.2 32.0 32.0 35.0 35.1 40.0 40.0 41.0 42.3 42.8 43.0 44.0 44.0 44.0 45.0
## [19] 46.4 48.0 48.1
```

```
# Order returns the indices of the original (unsorted) vector in the order that they would appear if pr
order(turtles.df$carapace_length)
```

```
##  [1]  3 10  4 20  5 12 13 14  7 11  1 15  6 18  9 17 21  8  2 19 16
```

```
# To sort a data frame by one vector, use "order()"
turtles.tag <- turtles.df[order(turtles.df$tag_number),]

# Order in reverse
turtles.tag.rev <- turtles.df[order(turtles.df$tag_number,decreasing = TRUE),]

# Sorting by 2 columns
turtles.sex.weight <- turtles.df[order(turtles.df$sex,turtles.df$weight),]
```

Dealing with missing data

In many real-world datasets, observations are incomplete in some way- they are missing information.

In R, the code "NA" represents elements of a vector that are missing for whatever reason.

Most statistical functions have ways of dealing with NAs.

Let's explore a data set with NAs.

Download this Data with missing values and save to your working directory.

Now let's explore this data set in more detail:

```
####
####  Missing Data
####

# Try reading in a data file with missing values, w/o specifying how the text is delimited
# Will not read because of missing data, it does not know where the columns are!
```

```r
# missing.df <- read.table(file="data_missing.txt")

# If you specify the header and what the text is delimited by, it will read them as NA
missing.df <- read.table(file="data_missing.txt", sep="\t", header=T)

# Missing data is read as an NA
missing.df

# Omits (removes) rows with missing data
missing.NArm.df <- na.omit(missing.df)

# ?is.na
is.na(missing.df)
```

```
##       Country Import Export Product
##  [1,]   FALSE  FALSE  FALSE   FALSE
##  [2,]   FALSE  FALSE  FALSE   FALSE
##  [3,]   FALSE  FALSE  FALSE   FALSE
##  [4,]   FALSE  FALSE  FALSE   FALSE
##  [5,]   FALSE  FALSE  FALSE   FALSE
##  [6,]   FALSE  FALSE  FALSE   FALSE
##  [7,]   FALSE   TRUE  FALSE   FALSE
##  [8,]   FALSE  FALSE  FALSE   FALSE
##  [9,]   FALSE  FALSE  FALSE   FALSE
## [10,]   FALSE  FALSE  FALSE   FALSE
## [11,]   FALSE  FALSE  FALSE   FALSE
## [12,]   FALSE  FALSE   TRUE   FALSE
## [13,]   FALSE  FALSE  FALSE   FALSE
## [14,]   FALSE  FALSE  FALSE   FALSE
## [15,]   FALSE  FALSE  FALSE   FALSE
## [16,]   FALSE   TRUE  FALSE   FALSE
## [17,]   FALSE  FALSE  FALSE   FALSE
## [18,]   FALSE  FALSE   TRUE   FALSE
## [19,]   FALSE  FALSE  FALSE   FALSE
## [20,]   FALSE   TRUE  FALSE   FALSE
```

```r
# Get index of NAs
which(is.na(missing.df))
```

```
## [1] 27 36 40 52 58
```

```r
# Replace all missing values in the data frame with a 0
missing.df[is.na(missing.df)] <- 0

# Create another data frame with missing values
missing.mean.df<- read.table(file="data_missing.txt", sep="\t", header=T)

# Replace only the missing values of just one column with the mean for that column
missing.mean.df$Export[is.na(missing.mean.df$Export)] <- mean(missing.mean.df$Export,na.rm=T)
missing.mean.df

# Return only those rows with missing data
```

```r
missing.df<- read.table(file="data_missing.txt", sep="\t", header=T)
missing.df <- missing.df[!complete.cases(missing.df),]

# Can summarize your data and tell you how many NA's per col
summary(missing.mean.df)
```

```
##     Country              Import          Export          Product
##  Length:20          Min.   :35.00   Min.   : 0.00   Length:20
##  Class :character   1st Qu.:60.00   1st Qu.: 3.75   Class :character
##  Mode  :character   Median :74.00   Median :10.22   Mode  :character
##                     Mean   :70.53   Mean   :10.22
##                     3rd Qu.:84.00   3rd Qu.:15.25
##                     Max.   :89.00   Max.   :23.00
##                     NA's   :3
```

```r
# Summarize one individual col
summary(missing.mean.df$Export)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    3.75   10.22   10.22   15.25   23.00
```

ASIDE: don't use "attach"!

Never use the "attach([some data frame])" function in R. This "convenience" function allows you to access all variables in a data frame by referencing the variable name only (instead of using the dollar-sign notation). That is, when you attach a data frame, you can do things like this:

```r
##############
# COMMON PITFALLS
##############


#############
# The attach() function

# Note: some people like to use attach
# We DO NOT recommend using attach

df <- data.frame(    # build a data frame
  a=1:10,
  b=rnorm(10)
)

df$a       # pull out columns of the data frame
df$b

attach(df)   # "attach" the data frame  # attach make objects within dataframes accessible with fewer l

a          # now we can reference the columns without referencing the data frame
b
```

```
detach(df)      # remember to detach when you're done

#########
# BUT...

attach(data.df)

Import

Import<-rep(1,10) # What if we forget the names in our data frames and use that name for something else
Import

rm(Import) # But if we remove that object...

Import   # There is still the original object behind it. That's troubling!

detach(data.df)    # Remember to detach
```

BUT this just leads to problems in the end. Our recommendation: just don't use it!!

Challenge Exercises!

1: Create a new data frame by saving the file "comm_data.txt" to your working directory. For the new data frame, include only the following columns: Hab_class, C_DWN, C_UPS, and rename the columns as: "Class","Downstream", "Upstream".

2: Read in the file "turtle_data.txt', and explore the data. Create a new version of this data frame with all missing data removed. Save this new data frame as a comma delimited text file in your working directory.

3: I am trying to create a data frame with only male turtles using the "which()" function. Would the following command work?

```
male_turtles <- turtles[,which(turtles$sex=="male")]
```

Why or why not?

What other method could I use to create this without using "which()"?

```
############
### CHALLENGE EXERCISES
############

# 1: Create a new data frame by saving the file ["comm_data.txt"](comm_data.txt) to your working direct
#
# 2: Read in the file "turtle_data.txt', and explore the data. Create a new version of this data frame
#
# 3: I am trying to create a data frame with only male turtles using the "which()" function.  Would the

# male_turtles <- turtles[,which(turtles$sex=="male")]  # uncomment this to run it...
```

–go to next module–

Aside: Tips on naming objects (and data frame columns) in R

1. No spaces within object names (same for column names in data files) – use underscore instead!

2. Try not to name an object with a similar/same as a function (e.g. don't name your data "data.frame"). NOTE: this can be hard- there are lots of functions in base R!

3. Can't start object names (or column names in data frame) with a number

Aside: "dropping dimensions"   When subsetting matrices or arrays, R automatically "drops" dimensions. This is best understood by example:

```
##########
# Demo- using the "drop=TRUE" argument when subsetting higher-dimensional objects

newmat <- matrix(c(1:6),nrow=3,byrow = T)
class(newmat)
```

```
## [1] "matrix" "array"
```

```
newmat[1,]
```

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

```
class(newmat[1,])    # what? why is it no longer a matrix????
```

```
## [1] "integer"
```

```
newmat[1,,drop=FALSE]
```

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

```
class(newmat[1,,drop=FALSE])    # ahhh, now we retain a 2-D matrix!
```

```
## [1] "matrix" "array"
```

This default behavior in R can lead to enormous problems when programming in R- just remember that the "drop=FALSE" argument can be your friend!!!