Module 1.2

Getting started: managing data

February 2019

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 a convenient folder on your laptop.
2. Load your script in RStudio. To do this, open RStudio 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

To be able to import and export data to/from your R workspace, it makes things a lot easier if you can 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 with complex directory names!

A working directory is the folder that R uses for reading or writing files if no other directory information is specified. Every R session has a working directory, whether you specify one or not. Let's find out what my working directory is right now:

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

```
## [1] "E:/GIT/R-Bootcamp"
```

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

ASIDE: RStudio Projects

If you're using an **RStudio Project** (.Rproj extension) then the default working directory is the directory where the .Rproj file lives- which 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 immediately read/write data to that directory (without having to set the working directory), even though the directory path of the project is probably different on your two machines. This can save a lot of headaches!

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

Back to basics!

You can easily set a new working directory using the "setwd()" function. To select the directory manually (using standard file navigation) you can use the "file.choose()" function:

```r
# Setting a new working directory in a PC (same as File->Change dir...)
#setwd("filepath with forward slashes")

# for example...

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

# Open finder to set directory
setwd(file.choose())
```

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 top dropdown menus to set the working directory (Session->Set Working Directory->Choose Directory).

Once you have set the working directory, you can use the "dir()" 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 R bootcamp website!

```r
# Names of files in working directory
dir()
```

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)
- read.csv2
  - fields are separated by semicolons
- read.delim
  - tab delimited files
- read.delim2
  - tab delimited with periods as commas

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 a convenient file location (e.g., the R bootcamp folder where your scripts are already!)

- Tab delimited data file
- Whitespace delimited data file
- Comma delimited data file

2. Set the directory where the data files are as your working directory! In my case, the R command would look like this:

```
setwd("E:/GIT/R-Bootcamp")   # set working directory to the folder with my data!

# note: since I use an R project (.Rproj), this is automatically my working directory when I open the R
```

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.csv
# ?readLines

# read.table with tab delimited file (default is sep = "" (one of several common delimiters, including
data.tab.df <- read.table("data.dat", header=TRUE, sep="")

head(data.tab.df)    # display the first few lines of the data frame

# read.table to import textfile
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=",")

# ?names: lists names of an object (columns, name of list elements)
names(data.csv.df)

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

# Remove objects we won't be using
rm(data.tab.df)
rm(data.txt.df)
rm(data.csv.df)
```

You can also import data from the internet directly!

Here's how:

```
########
# Import data from the internet directly

brain.df <- read.table("http://www.oup.com/us/static/companion.websites/9780195089653/Spreadsheets/brai
head(brain.df)
dim(brain.df)
```

Using R's built in data

R has many useful built-in data sets. 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 the passengers of the titanic
data(Titanic)

str(Titanic)    # examine the structure of this data object
```

Aside: Tips on good data management in R

1. No spaces within object names (same for column names in data files, use . or _ 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 so many functions in base R!
3. Can't start object names with a number

Basic data checking

As you've already seen, data objects can be vectors, matrices, arrays, data frames, and lists.

Elements of vectors, matrices, etc. can be coded as numbers (numeric), characters, logicals (TRUE/FALSE), factors etc.

You can use the "class()" function to see what kind of object you're working with:

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

# ?class: tells you what type of data object you have
class(data.df)
```

```
## [1] "data.frame"
```

Okay, we have a data frame. We know that! What about the internals of the data frame? For this we can use the "str()" function, which tells us some more about what's IN our data object:

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

```
## 'data.frame':    20 obs. of  4 variables:
##  $ Country: Factor w/ 20 levels "Bolivia","Brazil",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ 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: Factor w/ 2 levels "A","N": 2 2 2 1 1 1 1 1 1 1 ...
```

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):

```r
# ?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)
```

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

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

# ?write.table: writes a file to the working directory
# ?writeLines: writes to a file, one line at a time

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

Saving (and loading) data

The data loaded in your **R workspace** are stored in your computers RAM as **binary** representations that are efficient but not particulary 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()" (or "save.image()") and "load()" functions in R:

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

```r
####
####  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 workspace

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

```r
save.image(file="Module2.RData")    # ?save.image: saves entire workspace

load(file="Module2.RData")  # load the workspace from the working directory
```

NOTE: If you're using an RStudio Project (.Rproj), then the contents of your workspace are automatically saved when you close the project, and load again when you open the project file. This allows you to easily pick up where you left off!

Clearing the workspace

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

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

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

rm(list=ls())   # clear the entire workspace. Confirm that your workspace 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 statistics, R is very a powerful environment for doing data transformations and mathematical operations.

Boolean operations

Boolean operations refer to TRUE/FALSE tests. That is, we impose a condition on 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:

```r
# <- assignment operator: required for functions
# =  alternative assignment operator

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

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

```r
a == 2
```

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

```r
####
####  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 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

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

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

# 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 useful "which()" function returns the indices of a vector that meet a certain spedified 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 the data column that are equal to 74
```

```
## [1]  2 17
```

```
which(data.df[,2]!=74)       # elements of the data column 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 OR operator
```

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

```
data.df[which(data.df[,2]<74),2]    # check to make sure these numbers are under 74- they had better be
```

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

```
#########
# 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 don't actually need to use the "which()" function

data.df[data.df[,2]<74,2]    # alternative 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[,1]=="Chile")
```

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

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

```
## Warning in is.na(e1) | is.na(e2): 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[,1] %in% sub.countries) & (data.df[,4]=="N"))   # use AND operator
```

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

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

```
## [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,]
```

Practice exercise!

Let's practice some of what we've learned. Here we will use a different data file. Take a couple minutes to make sure you understand this code.

First, make sure you download the Turtle Data and save to your working directory.

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"
## [5] "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
```

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

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

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

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

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

# 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 <- subset(turtles.df,sex!="male")                # alternative
```

```
female.turtles <- turtles.df[which(turtles.df$sex!="male"),]    # 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:

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

# Setting the length for all turtles with weight 5 or heavier to 48
turtles.df$carapace_length[which(turtles.df$weight >=5)] = 48

# Sets the TRUE indexes from above to 2
turtles.df$size.class[turtles.df$weight >= 6] <- 1        # make a new variable "size.class" based on t
turtles.df$size.class[turtles.df$weight < 6] <- 2

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
```

```
ncol(turtles.df)
```

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

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
# ?order

# 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 48.0 48.0 48.0 48.0 48.0 48.0
## [15] 48.0 48.0 48.0 48.0 48.0 48.0 48.0
```

```
# 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  1  2  6  7  8  9 11 15 16 17 18 19 21
```

```
# 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[rev(order(turtles.df$tag_number)),]

# 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 some 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. We will explore this later.

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!
# 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
```

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

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

```
# 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
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
##  Bolivia  : 1   Min.   :35.00   Min.   : 0.00   A:13
##  Brazil   : 1   1st Qu.:60.00   1st Qu.: 3.75   N: 7
##  Chile    : 1   Median :74.00   Median :10.22
##  Colombia : 1   Mean   :70.53   Mean   :10.22
##  CostaRica: 1   3rd Qu.:84.00   3rd Qu.:15.25
##  Cuba     : 1   Max.   :89.00   Max.   :23.00
##  (Other)  :14   NA's   :3
```

```
# 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
```

Working with headers

Sometimes you may want to change the headers (variable names) for your data, perhaps to improve readability. Here's how:

```
####
####  Manipulate Headers (variable names)
####

# ?names: used to manipulate the column names of a data frame

# Rename the columns of 'data'
names(data.df) <- c("CNTRY", "IMP","EXP", "PROD")
head(data.df)

# Another way to manipulate the names
colnames(data.df) <- c("Country", "Imports","Exports", "Products")
head(data.df)

# ?rownames
# Try to do similar manipulations with the row names!

rownames(data.df) <- paste("obs",1:nrow(data.df),sep="")

head(data.df)
```

Pitfalls to avoid

1. 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:

```
###############
# 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(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!!

2. When reading in data, R automatically codes character data (any column containing even one character element) as a **factor**. This often leads to problems, since factor data (while useful) are actually integers and can confuse novice and expert R users alike. This is one of the most common problems that new R users face!

My recommendation is that, when reading in data, use the function argument "stringsAsFactors=FALSE" so that R reads these variables in as character strings rather than as factors:

```
df <- read.csv("data.csv", header=T, stringsAsFactors=FALSE)   # avoid reading in character data as fac
```

3. 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"
```

```
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"
```

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!!!

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()"?

–go to next module–