Submodule 1.2

Getting started: managing data

Spring 2024

Load script for submodule #1.2

1. Click here to download the script! Save the script to the project directory you set up in the previous module.
2. Load your script in RStudio. To do this, open RStudio and click on the folder icon in the toolbar at the top to load your script.

Let's get started with data management in R!

Working directory

The working directory is the first place R looks 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
# Working directories --------------------

# 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 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 set a new working directory using the “setwd()” function.

```
# for example...

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

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()    # uncomment this to run it...
```

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 data to/from R. But here are the basics.

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

- read.csv or read_csv
  - fields are separated by a comma (this is 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!

NOTE: you can also read data from Excel (.xlsx files) directly using the ‘readxl’ package.

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/Export data files into R----------------------

# read_csv to import textfile with columns separated by commas
data.df <- read_csv("data.csv")
names(data.df)

# Remove redundant objects from your workspace
rm(data.txt.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!

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

head(mtcars)    # inspect the first few lines

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

Many packages come with built-in datasets as well. For, instance, ggplot2 comes with the "diamonds" package:

```
ggplot2::diamonds   # note the use of the package name followed by two colons- this is a way to make su
```

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 objects --------------------------

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

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```r
str(data.df)
```

```
## spc_tbl_ [20 x 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ Country: chr [1:20] "Bolivia" "Brazil" "Chile" "Colombia" ...
##  $ Import : num [1:20] 46 74 89 77 84 89 68 70 60 55 ...
##  $ Export : num [1:20] 0 0 16 16 21 15 14 6 13 9 ...
##  $ Product: chr [1:20] "N" "N" "N" "A" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   Country = col_character(),
##   ..   Import = col_double(),
##   ..   Export = col_double(),
##   ..   Product = col_character()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

```r
names(diamonds)
```

```
##  [1] "carat"   "cut"     "color"   "clarity" "depth"   "table"   "price"   "x"       "y"       "z"
```

And we can use the "summary()" function to get a brief summary of the contents of each column in our data frame:

```r
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 drive as well. 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 as table

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

# ?write_csv: writes a CSV file to the working directory

newdf <- data.df[,c("Country","Product")]

write_csv(newdf, file="data_export.csv")   # export a subset of the data we just read in.
```

Saving (and loading) R data objects (binary)

The data loaded in your **R environment** are stored in your computer's memory as **binary** representations that are efficient but not 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 save data we can use the "saveRDS()" and "save()" functions. To load data we can use the "readRDS()" and "load()" functions:

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

```
#  Saving and loading

# saveRDS: save a single object from the environment to hard disk
# save: saves one or more objects from the environment to hard disk. Must be read back in with the same

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

saveRDS(b, "Myobject1.rds")    # use saveRDS to save individual R objects
save(a,b,file="Myobjects1.RData")    # use 'save' to save sets of objects
save.image("Myworkspace.RData")      # use 'save.image' to save your entire workspace

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

new_b <- readRDS("Myobject1.rds")
load("Myworkspace.RData")   # load these objects back in with the same names!
```

Clearing the environment

Sometimes your environment can get cluttered with objects. In these cases, it can help to clear the environment. You can just use the 'broom' icon in your environment window in RStudio to clear your environment, or use this command:

```
# Clear the environment

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

data.df <- read_csv("data.csv")  # read the data back in!
```

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 can ask a question about the data to a Boolean operator and the operator will return a TRUE or a FALSE (logical) result.

First, let's meet the boolean operators.

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

```
# Working with data in R ------------------------

# <- assignment operator
# =  alternative assignment operator

a <- 3     # assign the value 3 to the object named "a"
b = 5      # assign the value 5 to the object named "b"
a == 3     # answer the question: "does the object "a" equal "3"?
```

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

```
a == b
```

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

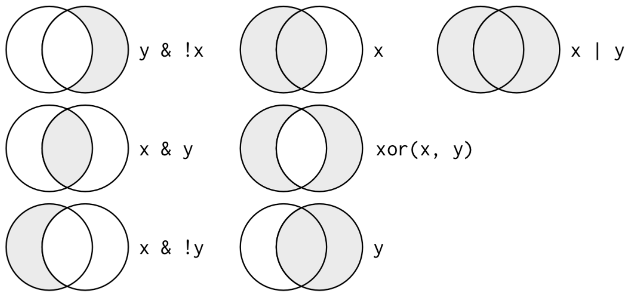
```
# what happens if you accidentally typed 'a = b'?
```

```
#  Boolean operations

# Basic operators

# <    less than
# >    greater than
# <=   less than or equal to
# >=   greater than or equal to
# ==   equal to
# !=   not equal to
# %in% belongs to a set

# Combining multiple conditions

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

```
# explore Boolean operators -----------------------


Y <- 4   # first define a couple new objects
Z <- 6

Y == Z  # is Y equal to Z?  (T/F)
Y < Z   # is Y less than Z?

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

data.df[,2]=74     # (for each element in the second column of data.df) is it equal to 74?

# OOPS! sets entire second column equal to 74! OOPS WE GOOFED UP!!!
```

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

# Let's do it right this time
data.df[,2]==74    # tests each element of column to see whether it is equal to 74

data.df[,2]<74|data.df[,2]==91   # combine two questions using the logical OR operator
```

This image from Wickham and Grolemund's R for Data Science book can help conceptualize the combination operators:



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 (e.g., where [some condition] is TRUE).

```r
#  Subsetting data

data.df[data.df[,"Import"]<74,]     # select those rows of data for which second column is less than 74

#  or alternatively, using tidyverse syntax (and the pipe operator):

data.df %>%
  filter(Import<74)    # use "filter" verb from 'dplyr' package

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

data.df %>%
  filter(Country %in% sub.countries)   # subset the dataset for only that subset of countries we're int
```

Practice data processing!

Let's switch to a different data file. Download the Turtle Data and save to your 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

```r
# Subsetting data ----------------------
#  Practice subsetting a data frame
turtles.df <- read_delim(file="turtle_data.txt",delim="\t")   # tab-delimited file
turtles.df

# Subset for turtles that weigh greater than or equal to 10g

subset.turtles.df <- turtles.df %>%
  filter(weight >= 10)

subset.turtles.df


# Subset for only females

fem.turtles.df = turtles.df %>%
  filter(sex=="female")

# Here we want to know the mean weight of all females
mean(fem.turtles.df$weight)
```

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

```
# or we can summarize the mean weight for males and females at the same time using the following tidyve

turtles.df %>%
  group_by(sex) %>%
  summarize(meanwt = mean(weight))

# OOPS, looks like we just caught an error!
```

Fixing data using subsetting

Let's fix the error we just noticed in the way sex was represented in the turtle dataset.

```
unique(turtles.df$sex)  # note the two ways of representing females...
```

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

```
turtles.df$sex[turtles.df$sex=="fem"] <- "female"  # correct the error

# or alternatively
turtles.df = turtles.df %>%
  mutate(
    sex= fct_collapse(sex,
              female=c("fem","female"),
              male=c("male")))


# or alternatively
turtles.df = turtles.df %>%
  mutate(sex = replace(sex,sex=="fem","female"))
```

```r
turtles.df %>%          # summarize weight by sex (check that it's fixed)
  group_by(sex) %>%
  summarize(meanwt = mean(weight))
```

Selecting and renaming columns

Sometimes we only want to work with a small subset of columns from a larger dataset. Or we have column names that are not informative or don't make sense to us.

```r
# select only the x y and z columns from the diamonds dataset

diamonds2 <- diamonds %>%
  select(x,y,z)

# in base R (non tidyverse) you can do this:
diamonds2 <- diamonds[,c("x","y","z")]

# to change the column names, use the "names()" function

names(diamonds2)  # extract the column names
```

```
## [1] "x" "y" "z"
```

```r
names(diamonds2)  <- c("length", "width","depth")

# or rename specific variables using 'dplyr' from the 'tidyverse;

diamonds2 <- rename(diamonds2, len = length)
```

Sorting/ordering data

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

*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" (i.e., to sort this vector in increasing order, you would need to take the second element, then the first, and then the third).

In the 'tidyverse' we can use the "arrange" verb to do this!

```r
#  Sorting

# The 'order' function returns the indices of the original (unsorted) vector in the order that would so
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
```

```r
# To sort a data frame by one vector, you can use "order()"
turtles.df[order(turtles.df$tag_number),]
```

```
# And in decreasing order:

turtles.df[order(turtles.df$tag_number,decreasing=T),]

# or we can use the "arrange" verb in the 'tidyverse':
turtles.df %>%
  arrange(carapace_length)

# or if we want in descending order...

turtles.df %>%
  arrange(desc(carapace_length))

# Sorting by 2 columns
turtles.df %>%
  arrange(sex,weight)
```

Challenge exercises

1. Save the "comm_data.txt" file to your working directory. Read this file in as a data frame. Select only the following columns: Hab_class, C_DWN, C_UPS (discard the remaining columns). Finally, rename these columns as: "Class","Downstream", and "Upstream" respectively. [hint 1: use read_table to read in the file as a data frame] [hint 2: use the "select" verb to select the columns you want] [hint 3: use the names() function to rename the columns]

2. Read in the file "turtle_data.txt". Create a new version of this data frame with all missing data removed (discard all rows with one or more missing data). Save this new data frame to your project directory as a comma delimited text file. [hint 1: use the "na.omit()" function to remove rows with NAs] [hint 2: use the write_csv function to write to your working directory]

3. Read in the file "turtle_data.txt". Create a new data frame with only male turtles. Use this subsetted data set to compute the mean and standard deviation for carapace length of male turtles.

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

# 1: Save the "comm_data.txt" file to your working directory. Read this file in as a data frame. Select
#
# 2: Read in the file "turtle_data.txt". Create a new version of this data frame with all missing data
#
# 3: Read in the file "turtle_data.txt". Create a new data frame with only male turtles. Use this subse
```

–go to next submodule–

Bonus: dealing with missing data

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

In R, the code "NA" (not available) stands in for 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 missing data.

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

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

```
#  Missing Data

# NOTE: you need to specify that this is a tab-delimited file. It is especially important to specify th

missing.df <- read_delim(file="data_missing.txt",delim="\t")   # try replacing with "read_table"- it do

# Missing data are read as an NA
missing.df

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

```
##    Country              Import          Export         Product
##  Length:20          Min.   :35.00   Min.   : 0.00   Length:20
##  Class :character   1st Qu.:60.00   1st Qu.: 3.25   Class :character
##  Mode  :character   Median :74.00   Median :11.00   Mode  :character
##                     Mean   :70.53   Mean   :10.22
##                     3rd Qu.:84.00   3rd Qu.:15.75
##                     Max.   :89.00   Max.   :23.00
##                     NA's   :3       NA's   :2
```

```
# Omits (removes) rows with missing data
na.omit(missing.df)

# ?is.na   (Boolean test!)
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
```

```
complete.cases(missing.df)   # Boolean: for each row, tests if there are no NA values
```

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

```
# Replace all missing values in the data frame with a an interpolation function from tidyverse: replace

missing.df %>%
  mutate(Export = replace_na(Export,mean(Export,na.rm=T)),
         Import = replace_na(Import,mean(Import,na.rm=T)))

# or using tidyverse trickery (less code repetition)
missing.df %>%
  mutate(across(where(is.numeric),~replace_na(.,mean(.,na.rm=T))))
```

Aside: the pipe operator (%>%)

The 'tidyverse' set of packages takes advantage of the pipe operator `%>%`, which provides a clean and intuitive way to structure code and perform sequential operations in R.

Key advantages include:

- code is more readable and intuitive – reads left to right, rather than inside out as is the case for nested function
- perform multiple operations without creating a bunch of intermediate (temporary) datasets

This operator comes from the *magrittr* package, which is included in the installation of all of the tidyverse packages. The shortcut for the pipe operator is ctrl-shift-m ('m' is for Magritte). When reading code out loud, use 'then' for the pipe. For example the command here:

```
x %>% log() %>% round(digits=2)
```

can be interpreted as follows:

> Take "x", THEN take its natural logarithm, THEN round the resulting value to 2 decimal places

The structure is simple. Start with the object you want to manipulate, and apply actions (e.g., functions) to that object in the order in which you want to apply them.

Here is a quick example.

```r
#  ASIDE: using the pipe operator %>% (ctrl-shift-m) in R

# start with a simple example
x <- 3

# calculate the log of x
log(x) # form f(x) is equivalent to
```

```
## [1] 1.098612
```

```r
x %>% log() # form x %>% f
```

```
## [1] 1.098612
```

```r
# example of multiple steps in pipe
round(log(x), digits=2) # form g(f(x)) is equivalent to
```

```
## [1] 1.1
```

```r
x %>% log() %>% round(digits=2)
```

```
## [1] 1.1
```

Some more subsetting practice:

Returning to the turtle example, we can use subsetting operations to correct or alter data:

```r
#  Data Manipulation using subsetting

# list of tags we do not trust the data for
bad.tags <- c(13,105)

turtles.df = turtles.df %>%
  mutate(
    sex = replace(sex,tag_number%in%bad.tags,NA),
    carapace_length = replace(carapace_length,tag_number%in%bad.tags,NA),
    head_width = replace(head_width,tag_number%in%bad.tags,NA),
    weight = replace(weight,tag_number%in%bad.tags,NA)
  )

# or... use some more tidyverse helper functions and tricks!

turtles.df = turtles.df %>%
  mutate(across(c("sex","carapace_length","head_width","weight"),
              ~replace(.,tag_number%in%bad.tags,NA)))

# or use the following non-tidyverse syntax... which still seems easier to me!

turtles.df[turtles.df$tag_number%in%bad.tags,c("sex","carapace_length","head_width","weight")]  <- NA
turtles.df

 # make a new variable "size.class" based on the "weight" variable

turtles.df = turtles.df %>%
  mutate(size.class = case_when(
    weight < 3 ~ "juvenile",
    weight > 6 ~ "adult",
    is.na(weight) ~ NA_character_,
    .default = "subadult"
  ))

turtles.df$size.class
```

```
##  [1] "adult"    "adult"    "juvenile" "juvenile" "subadult" "adult"    "adult"    "adult"    "adult"
## [10] NA         "adult"    "juvenile" "subadult" "subadult" "adult"    "adult"    NA         "adult"
## [19] "adult"    "juvenile" "adult"
```