Preparing data for analysis and visualization in R

Entering or loading data into R

Jenine Harris
Brown School

Creating vectors for different data types

- Usually when social scientists collect information to answer a question, they collect more than one number or word since that would be extremely inefficient.
- One commonly used object type for storing this type of information is a **vector**.
- A **vector** is a set of data elements that are saved together as the same type (numeric, logical, etc.).
- Each entry in a vector is called a *member* or *component* of the vector. Vectors are commonly used to store variables.

Using c() to create a vector

- The format for a vector uses the `c()` function for concatenate.
- The parentheses are filled with the elements of the vector separated by commas.
- If the members of the vector are meant to be saved as character type variables, use single or double quotes around each member.

```
# creates character vector char.vector
char.vector <- c('Oregon','Vermont','Maine')
# prints vector char.vector
char.vector
```

```
## [1] "Oregon"  "Vermont" "Maine"
```

```
nums.1.to.4 <- c(1,2,3,4)
nums.1.to.4
```

```
## [1] 1 2 3 4
```

```
logic.vector <- c(TRUE, FALSE, FALSE, TRUE)
logic.vector
```

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

Shortcut to creating and printing a vector

- A coding trick for creating new objects and printing them at the same time is adding parentheses around the code that creates the object:

```
# create and print vectors
( char.vector <- c('Oregon', 'Vermont', 'Maine') )
```

```
## [1] "Oregon"  "Vermont" "Maine"
```

```
( nums.1.to.4 <- c(1, 2, 3, 4) )
```

```
## [1] 1 2 3 4
```

```
( logic.vector <- c(TRUE, FALSE, FALSE, TRUE) )
```

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

Math with vectors

- Vectors can be combined, added to, subtracted from, subsetted, and other operations.

```
# add 3 to each element in the nums.1.to.4 vector
nums.1.to.4 + 3
```

```
## [1] 4 5 6 7
```

```
# add 1 to the 1st element of nums.1.to.4,
# 2 to the 2nd element, etc
nums.1.to.4 + c(1, 2, 3, 4)
```

```
## [1] 2 4 6 8
```

```
# multiply each element of nums.1.to.4 by 5
nums.1.to.4 * 5
```

```
## [1]  5 10 15 20
```

More math with vectors

```
# subtract 1 from each element and then divide by 5
(nums.1.to.4 - 1) / 5
```

```
## [1] 0.0 0.2 0.4 0.6
```

```
# make a subset of the vector including numbers > 2
nums.1.to.4[nums.1.to.4 > 2]
```

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

Saving new objects from vector math

- The results of these operations are printed in the Console but not saved in the Environment pane.
- To save a vector, assigned the operations to a new vector name using the assignment arrow.

```
# add three to number vector and save
# as new vector
( nums.1.to.4.plus.3 <- nums.1.to.4 + 3 )
```

```
## [1] 4 5 6 7
```

```
# divide number vector by 10 and save
# as new vector
( nums.1.to.4.div.10 <- nums.1.to.4 / 10 )
```

```
## [1] 0.1 0.2 0.3 0.4
```

Multiple computations on a single vector

- It is possible to do multiple computations on a single vector:

```
# add 3 and divide by 10 for each vector member
( nums.1.to.4.new <- (nums.1.to.4 + 3) / 10 )
```

```
## [1] 0.4 0.5 0.6 0.7
```

Creating a matrix to store data in rows and columns

- In addition to the **vector** format, R also uses the **matrix** format to store information. A matrix is information, or data elements, stored in a rectangular format with rows and columns.
- Coders can perform operations on matrices, or more than one matrix, like with vectors.
- The R command for producing a matrix is, surprisingly, `matrix()`.
- This function takes arguments to enter the data, `data =`, and to specify the number of rows, `nrow =`, and columns, `ncol =`.

Code to create and print a matrix

- The `byrow =` argument tells R whether to fill the data into the matrix by filling across first (fill row 1, then fill row 2, etc) or by filling down first (fill column 1 first, then fill column 2, etc).
  - This example uses `byrow = TRUE` so the data fills across first.
  - For the columns to fill first, she would have to use `byrow = FALSE` instead.

```
# create and print a matrix
( policies <- matrix(data = c(1, 2, 3, 4, 5, 6), # data in the matrix
                     nrow = 2,                   # number of rows
                     ncol = 3,                   # number of columns
                     byrow = TRUE) )             # fill matrix by rows
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Naming matrix rows and columns

- The `policies` matrix included the number of states with policies legalizing medical, recreational, and both types of marijuana that were in effect in 2013 and 2014.
- Naming the rows and columns would make it more useful; R uses `dimnames()` to assign names to rows and columns.
- Names are entered in vectors inside a list, with the first vector being the row names and the second vector being the column names.

Adding row and column names

- In this case, the row names were `c("2013", "2014")` for the two years of data and the column names were `c("medical", "recreational", "both")` for the three types of policy.

```
# add names to the rows and columns of the matrix
dimnames(x = policies) <- list(
  c("2013", "2014"),                                  # row names
  c("medical", "recreational", "both")                # column names
)

# print the policies matrix
policies
```

```
##      medical recreational both
## 2013       1            2    3
## 2014       4            5    6
```

Creating a data frame

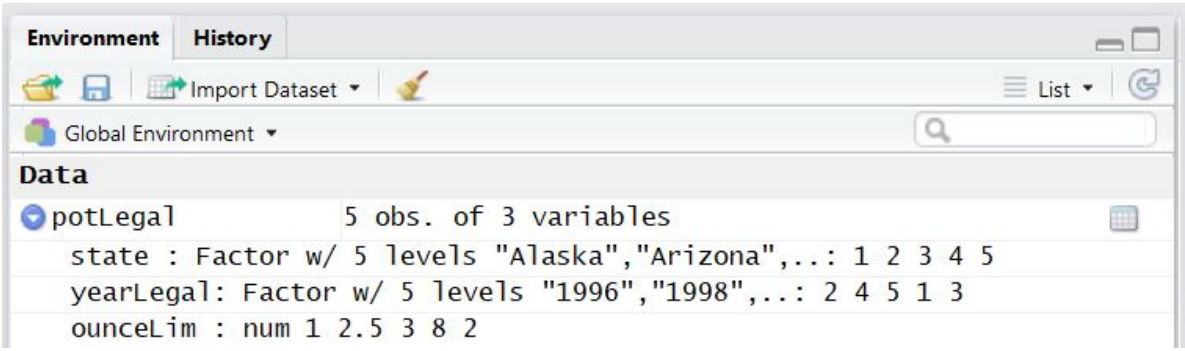
- Similar to a matrix format, the **data frame** format has rows and columns of data.
- In the data frame format, rows are observations and columns are variables.
- Data frames are often entered outside of R into a spreadsheet or other type of file and then imported into R for analysis.
- However, R users can also make their own data frame by entering data in vectors and combining them into a data frame using the `data.frame()` function, like this:

```
# state, year enacted, personal oz limit medical marijuana
# create vectors
state <- c('Alaska', 'Arizona', 'Arkansas', 'California', 'Colorado')
year.legal <- c('1998', '2010', '2016', '1996', '2000')
ounce.lim <- c(1, 2.5, 3, 8, 2)

# combine vectors into a data frame
# name the data frame pot.legal
pot.legal <- data.frame(state, year.legal, ounce.lim)
```

Checking the new data frame

- Just like in the `matrix()` function, the `data.frame()` function reads in multiple arguments.
- The `data.frame()` function has three arguments: state, year.legal, and ounce.lim.
- This time all of the arguments are objects, but that will not always be the case.
- After entering and running these code lines, check the Environment pane to see a new entry called `pot.legal`.
    - To the right of the label `pot.legal` it will show "5 obs. of 3 variables" indicating five observations and three variables.
    - Clicking the blue and white circle with a triangle in it to the left of `pot.legal` expands this entry to see more information about what is contained in the `pot.legal` object.



Accessing variables within a data frame

- The Environment window shows the `state` variable in the `pot.legal` data frame was assigned the variable type of factor, which is incorrect.
- In this case, the names of states are unique and not categories; change the `state` variable to a character variable using the `as.character()` function.
- Because the state variable is now part of a data frame object, identify both the data frame and the variable in order to change it.
- Enter the name of the data frame first, a `$` to separate the data frame from the variable, and the variable name, like this:

```
# change state variable from pot.legal data frame
# to a character variable
pot.legal$state <- as.character(x = pot.legal$state)

# check the variable type
class(x = pot.legal$state)
```

```
## [1] "character"
```

Summarizing a data frame

- A data frame has many options open for data management and analyses.
- For example, examine basic information about the variables by using the `summary()` function.
    - The `summary()` function requires at least one argument that identifies the object that should be summarized, like this:

```
# summarize the data frame
summary(object = pot.legal)
```

```
##     state            year.legal          ounce.lim
##  Length:5           Length:5           Min.   :1.0
##  Class :character   Class :character   1st Qu.:2.0
##  Mode  :character   Mode  :character   Median :2.5
##                                        Mean   :3.3
##                                        3rd Qu.:3.0
##                                        Max.   :8.0
```

Importing data frames from outside sources

- While typing data directly into R is possible and sometimes necessary, most of the time analysts will open data from an outside source.
- R is unique among statistical software packages because it has the capability of importing and opening data files saved in most formats.
- Some formats open directly in the base version of R.
- Other data formats require the use of an **R package**, which is a program written to do something specific in R.
- To know what format a data file is saved in, examine the file extension; common file extensions for data files include:
    - **.csv**: comma separated values
    - **.txt**: text file
    - **.xls** or **.xlsx**: Excel file
    - **.sav**: SPSS file
    - **.sasb7dat**: SAS file
    - **.xpt**: SAS transfer file
    - **.dta**: Stata file

Importing a comma separated values (csv) file

- In addition to the kind of file, opening a file requires the location of the file.
- R can open files saved locally on a computer, in an accessible shared location, or directly from the internet.
- To open a csv file, the most straightforward way is with the `read.csv()` command, however, this command may sometimes result in misreading of variable names or row names, so be sure to review newly opened data.

Example of importing a csv file

- For example, try loading the data set named **legal_weed_age_GSS2016_ch1.csv** that was imported from the General Social Survey (GSS) website
  - Start by making a *data* folder inside the folder where the code is saved
  - Download and save the csv data file there save the downloaded data there
  - Use the `read.csv()` function to import the data from that folder location like this:

```
# read the GSS 2016 data
gss.2016 <- read.csv(file = "data/legal_weed_age_GSS2016_ch1.csv")

# examine the contents of the file
summary(object = gss.2016)
```

```
##     grass                age          
##  Length:2867        Length:2867       
##  Class :character   Class :character  
##  Mode  :character   Mode  :character
```

Features of the data importing

- Notice that the location of the data was inside quote marks and did not include the entire file path.
    - This is one benefit of saving the data in the same folder as the code or in a subfolder.
    - If the data were saved in another location on the local computer or online, the full path to the data file would also work to import a csv file with `read.csv()`.
- The `summary()` command output shows two column headings, "grass" and "age".
- These two column headings are the two variables in the data set.

Other ways to import a csv file

- The `fread()` function in the **data.table** package or the `read_csv()` command in the **tidyverse** package might be useful for opening csv files saved from online sources if `read.csv()` doesn't work well.
- To install a package, go to the Tools menu in R Studio and select **Install Packages...**.
    - Type "data.table" or the name of whichever package to install in the dialog box that opens.
- Once a package is installed, there are two ways to open it and use it.

Using packages

- When using a function from a package one time, it is not necessary to open the package and leave it open.
- Instead, there is another way to open a package temporarily just to use a particular function.
- To temporarily open a package in order to use a command from the package, add the package name before the command name and separate with two colons, like this: `package.name::function()`.
- Try using this way to open the **data.table** package and used the *Fast and friendly file finagler* `fread()` function from the package to open the GSS data file.

Importing a csv file with fread()

```
# bring in GSS 2016 data
gss.2016 <-
  data.table::fread(input = "data/legal_weed_age_GSS2016_ch1.csv")

# examine the contents of the file
summary(object = gss.2016)
```

```
##     grass                age
##  Length:2867        Length:2867
##  Class :character   Class :character
##  Mode  :character   Mode  :character
```

Using :: to open a package temporarily

- Like with the `read.csv()` function, both the variables came into R as character variables, so they might have to use `as.factor()` and `as.numeric()` to fix the data types before using these variables.

- An important benefit of the `::` way of opening a package for use is that, occasionally there are function names that are the same in two different packages.

    - If two packages containing function names that are the same are opened at the same time in an R file, there will be a **namespace** conflict where R cannot decide which function to use.
    - One example is the function `summarize()`, which is included as part of the **dplyr** package and the **Hmisc** package.
    - When both packages are open, using the `summarize()` command results in an error.

Using library() to open a package

- The `library()` function is the second (and more common) way to open a package.
    - When the package will be used for more than once or twice, use the `library()` function.
    - Once the package is opened using `library()` it stays open until R is closed.
- The **dplyr** package is loaded with the **tidyverse** package.
- To demonstrate the error, try loading **Hmisc** and **tidyverse**:

```
# load Hmisc and tidyverse
library(package = "tidyverse")
library(package = "Hmisc")
```

Use summarize() to examine a variable

- Try the `summarize()` function:

```
# use the summarize command
gss.2016 %>%
  summarize(length.age = length(x = age))
```

```
Error in summarize(., length.age = length(x = age)) : argument "by" is
 missing, with no default
```

Show Traceback

Conflicts when function names repeat across packages

- This is relatively rare, but a good thing to keep in mind when a function does not run and you've checked the code a lot.
- There are a couple of ways to check to see if a **namespace** conflict is occurring.
- The first is to use the `conflicts()` function:

```
# check for conflicts
conflicts()
```

```
##  [1] "%>%"          "%>%"          "src"          "summarize"
##  [5] "%>%"          "%>%"          "as_tibble"    "contains"
##  [9] "ends_with"    "everything"   "last_col"     "matches"
## [13] "num_range"    "one_of"       "starts_with"  "tibble"
## [17] "tribble"      "%>%"          "add_row"      "as_data_frame"
## [21] "as_tibble"    "data_frame"   "data_frame_"  "frame_data"
## [25] "glimpse"      "lst"          "lst_"         "tbl_sum"
## [29] "tibble"       "tribble"      "trunc_mat"    "type_sum"
## [33] "enexpr"       "enexprs"      "enquo"        "enquos"
## [37] "ensym"        "ensyms"       "expr"         "quo"
## [41] "quo_name"     "quos"         "sym"          "syms"
## [45] "vars"         "%>%"          "mask"         "filter"
## [49] "lag"          "body<-"       "format.pval"  "intersect"
```

Using :: to address "conflicts in the namespace"

- The easiest thing to do to address the conflict is to use the `::` and specify which package to get the `summarize()` function from.
- To use `summarize()` from **dplyr**, the code would look like this:

```
# use summarize from dplyr
gss.2016 %>%
  dplyr::summarize(length.age = length(x = age))
```

```
##   length.age
## 1       2867
```

Use environment() to look for namespace conflicts

- Another way to check and see if a function is in conflict after an error message is to use `environment()` to check which package is the source for the `summarize()` function.

```
# check source package for summarize
environment(fun = summarize)
```

```
## <environment: namespace:Hmisc>
```

- The output shows that the namespace for `summarize()` is the **Hmisc** package instead of the **dplyr** package.

Cleaning data types in an imported file

- The variable names look good after loading with `fread()`, but the variables both were **character** data type.
- Check the codebook for the General Social Survey (GSS) that was saved as **gss_codebook.pdf** to determine what data types these variables are.
- On page 304 of the codebook it shows the measurement of the variable grass, which has five possible responses:

Do you think the use of marijuana should be made legal or not?

- Should
- Should not
- Don't know
- No answer
- Not applicable

Using GSS Explorer to understand variables

- Variables with categories are categorical and should be factor type variables in R.
- The GSS Data Explorer website is also useful for finding variables (https://gssdataexplorer.norc.org/variables/vfilter).
- Age appears to be measured in years up to age 88 and then "89 OR OLDER" represents people who are 89 years old or older

Examining codebook information

- The codebook and data suggest `grass` should be a factor and `age` should be numeric.
- Since "89 OR OLDER" is not an actual number, trying to force the `age` variable with "89 OR OLDER" in it into a numeric variable will result in an error.
- Before converting `age` into a numeric variable, first **recode** anyone that has a value of "89 OR OLDER" to instead have a value of "89."
    - This will ensure that `age` can be treated as a numeric variable.
    - Be careful in using and reporting this recoded `age` variable since it would be innaccurate to say that every person with the original "89 OR OLDER" label was actually 89 years old.

Cleaning the grass variable

- Start with with first converting `grass` into a factor.
- Because this is *not* changing the contents of the variable, keeping the same variable name seems logical.
- Use the arrow to assign the variable with the new assigned type back to the same variable name.
- Notice that the data frame name and variable name on the left of the assignment arrow `<-` are exactly the same as on the right.
- When new information is assigned to an existing variable, it over-writes whatever was saved in that variable.

```
# change grass variable to a factor
# check the data type
gss.2016$grass <- as.factor(x = gss.2016$grass)
class(x = gss.2016$grass)
```

```
## [1] "factor"
```

Cleaning the age variable

- For the trickier bit of recoding `age`, start with:

```
# recode the 89 OR OLDER category to 89
gss.2016$age[gss.2016$age == "89 OR OLDER"] <- '89'

# change age to numeric
gss.2016$age <- as.numeric(x = gss.2016$age)

# check the work
summary(object = gss.2016)
```

```
##        grass           age       
##  DK        : 110   Min.   :18.00  
##  IAP       : 911   1st Qu.:34.00  
##  LEGAL     :1126   Median :49.00  
##  NOT LEGAL: 717   Mean   :49.16  
##  NA's      :   3   3rd Qu.:62.00  
##                    Max.   :89.00  
##                    NA's   :10
```

- This line of code can be read as: "in the `age` variable of the `gss.2016` data frame, find any observation that is equal to '89 OR OLDER', and assign those particular observations to be the character '89'".
- This particular line of code is tricky and will be covered in more detail later.

Examining the data cleaning code

The full code for importing and cleaning the data:

```
# bring in GSS 2016 data
gss.2016 <-
  data.table::fread(input = "data/legal_weed_age_GSS2016_ch1.csv")
```

```
# change the variable type for the grass variable
gss.2016$grass <- as.factor(x = gss.2016$grass)
```

```
# recode "89 OR OLDER" into just "89"
gss.2016$age[gss.2016$age == "89 OR OLDER"] <- "89"
```

```
# change the variable type for the age variable
gss.2016$age <- as.numeric(x = gss.2016$age)
```

Checking the work so far

```
# examine the variable types and summary to
# check the work
class(x = gss.2016$grass)
```

```
## [1] "factor"
```

```
class(x = gss.2016$age)
```

```
## [1] "numeric"
```

```
summary(object = gss.2016)
```

```
##        grass           age
##  DK        : 110   Min.   :18.00
##  IAP       : 911   1st Qu.:34.00
##  LEGAL     :1126   Median :49.00
##  NOT LEGAL: 717   Mean   :49.16
##  NA's      :   3   3rd Qu.:62.00
##                    Max.   :89.00
##                    NA's   :10
```

Check your understanding

Use `fread()` to open the GSS 2016 data set. Look in the environment pane to find the number of observations and the number and types of variables in the data frame.

Answer

```
# bring in GSS 2016 data
gss.2016 <-
  data.table::fread(input = "data/legal_weed_age_GSS2016_ch1.csv")
# 2867 observations and 2 variables in this data frame
# Both variables (grass, age) are character variable type
```