ECO 274 LAB: Data Wrangling: Subsetting Data Frames, Vectors, Matrices, and Lists

Learning Objectives

- To understand how to import external data set
- Construct data structures to store external data in R.
- Inspect data structures in R
- Filtering data
- Demonstrate how to subset data from data structures, exporting data to the directory.

Reading data into R

Regardless of the specific analysis in R we are performing, we usually need to bring data in for the analysis. The function in R we use will depend on the type of data file we are bringing in (e.g. text, Stata, SPSS, SAS, Excel, etc.) and how the data in that file are separated, or delimited. The table below lists functions that can be used to import data from common file formats.

Data type	Extension	Function	Package
Comma separated values	csv	read.csv()	utils (default)
		read_csv()	readr (tidyverse)
Tab separated values	tsv	read_tsv()	readr
Other delimited formats	txt	read.table()	utils
		read_table()	readr
		read_delim()	readr
Stata version 13-14	dta	readdta()	haven
Stata version 7-12	dta	read.dta()	foreign
SPSS	sav	read.spss()	foreign
SAS	sas7bdat	read.sas7bdat()	sas7bdat
Excel	xlsx, xls	read_excel()	readxl (tidyverse)

- For example

```
## First let's check out what are the datasets available within the R-environment, usually called as ## Built in data sets

data()## see all the available data sets
data(ChickWeight)
View(ChickWeight)

# Next, ##Importing Data in R Script
# Import the brandChoice.csv dataset
```

```
df_brandChoices_method1 <- read.csv(file.choose(), header=T)
df_brandChoices_method2 <- read.csv(file = "brandChoices.csv", header = TRUE, stringsAsFactors=FALSE)
```

Now we import xlsx data set using readxl/tidyverse package

```
# Import the flower xls|xlsx dataset
my_data <- read_excel("flower.xls")

# Import the flower.txt dataset
flowers <- read.table(file = 'flower.txt', header = TRUE, sep = "\t",
stringsAsFactors = TRUE)
```

Importing dataset from online data archive:

```
#Import data from online

df <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data",header = FALSE)
View(df)
```

Importing datasets that are STATA, SAS, or SPSS format:

```
# Read Stata data into R
mydata <- read.dta("affairs.dta")

# Read the SPSS data
mySPSSData <- read.spss("dataFileName.sav", to.data.frame=TRUE,
use.value.labels=FALSE)


# Activate the `sas7bdat` library
library(sas7bdat)
# Read in the SAS data
mySASData <- read.sas7bdat("dataFileName.sas7bdat")
```

Inspecting data structures

There is a wide selection of base functions in R that are useful for inspecting your data and summarizing it.

List of functions for data inspection

We already saw how the functions head() and str() can be useful to check the content and the structure of a data.frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of data.

- All data structures - content display:
    - **str():** compact display of data contents (env.)

    - **class():** data type (e.g. character, numeric, etc.) of vectors and data structure of dataframes, matrices, and lists.
    - **summary():** detailed display, including descriptive statistics, frequencies
    - **head():** will print the beginning entries for the variable
    - **tail():** will print the end entries for the variable
- Vector and factor variables:
    - **length():** returns the number of elements in the vector or factor
- Dataframe and matrix variables:
    - **dim():** returns dimensions of the dataset
    - **nrow():** returns the number of rows in the dataset
    - **ncol():** returns the number of columns in the dataset
    - **rownames():** returns the row names in the dataset
    - **colnames():** returns the column names in the dataset

Let's use the affairs data that we imported above to test out data inspection functions.

```
#download affairs data from the web

affairs <- read.dta("http://fmwww.bc.edu/ec-p/data/wooldridge/affairs.dta")

#alternatively you can download data from the GitHub page
# or you can use package Wooldridge
# require(wooldridge)
# data(package="wooldridge")
# data("affairs")
# View(affairs)
View(affairs)

head(affairs)
tails(affairs)

str(affairs)
View(affairs$kids)

class(affairs$kids)

#create factors for kids and for marriage and attach labels

haskids <- factor(affairs$kids,labels = c("no","yes"))

#for ratmarr collum, create five labels and convert the col values

mlab <- c("very unhappy","unhappy","average","happy","very happy")
marriage <- factor(affairs$ratemarr,labels = mlab)
marriage
table(haskids)#frequencies for kids
prop.table(table(marriage)) #marriage ratings and check the share/proportions


#Now make a contingency table and counts(display and store variables)
```

```
(countstab <- table(marriage,haskids))

#now we will see the share with in marriage,i.e with in a row (1)
prop.table(countstab,margin = 1)

#next check share within "haskids",i.e with in a column
prop.table(countstab,margin = 2)

#lets make some grpah to depcit above information
pie(table(marriage),col = c("blue","green","yellow","red","grey"),main =
"Proportion of marriage couple")
table(marriage)

# x <- c(16,66,93,194,232)
# library(plotrix) # you need this package to draw 3D plot. it looks cool!!
# pie3D(x,labels=mlab,explode=0.1,
#       main="Distribution of marriage status ")

barplot(table(marriage),horiz = F,las=1,
        main = "Distribution of happiness",ylim = c(0,180), col =
c("blue","green","yellow","red","purple"))

barplot(table(haskids, marriage),horiz = T,las=1,
        legend=T, args.legend = c(x="bottomright"),
        main = "Happiness by kids",col = c("green","purple"))



barplot(table(haskids, marriage),beside = T,las=2,
        legend=T, args.legend = c(x="topleft"),
        main = "Happiness by kids",col = c("green","purple"))
```

When analyzing data, we often want to partition the data so that we are only working with selected columns or rows. A data frame or data matrix is simply a collection of vectors combined together. So let's begin with vectors and how to access different elements, and then extend those concepts to dataframes.

Let's start by creating a vector called age:

```
age <- c(15, 22, 45, 52, 73, 81)
```

Suppose we only wanted the fifth value of this vector, we would use the following syntax:

```
age[5]
```

If we wanted all values except the fifth value of this vector, we would use the following:

```
age[-5]
```

If we wanted to select more than one element we would still use the square bracket syntax, but rather than using a single value we would pass in a *vector of several index values*:

```
age[c(3,5,6)]   ## nested

# OR

## create a vector first then select
idx <- c(3,5,6) # create vector of the elements of interest
age[idx]
```

Let's select the *first four values* from age:

```
age[1:4]
```

if we wanted to know if each element in our age vector is greater than 50, we could write the following expression:

```
age > 50
```

Returned is a vector of logical values the same length as age with TRUE and FALSE values indicating whether each element in the vector is greater than 50.

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

We can use these logical vectors to select only the elements in a vector with TRUE values at the same position or index as in the logical vector.

Select all values in the `age` vector over 50 **or** `age` less than 18:

```
age > 50 | age < 18

age

age[age > 50 | age < 18]  ## nested

# OR

## create a vector first then select
idx <- age > 50 | age < 18
age[idx]
```

Indexing with logical operators using the `which()` function

While logical expressions will return a vector of TRUE and FALSE values of the same length, we could use the `which()` function to output the indices where the values are TRUE. Indexing with either method generates the same results, and personal preference determines which method you choose to use. For example:

```
which(age > 50 | age < 18)
```

```
age[which(age > 50 | age < 18)]  ## nested

# OR

## create a vector first then select
idx_num <- which(age > 50 | age < 18)
age[idx_num]
```

Notice that we get the same results regardless of whether or not we use the `which()`. Also note that while `which()` works the same as the logical expressions for indexing, it can be used for multiple other operations, where it is not interchangeable with logical expressions.

Data filtering

```
### Using pipe operator
require(tidyverse)
data("starwars")
View(starwars)


starwars %>% filter(height>150 & mass<200) %>%
  mutate(height_in_meters=height/100) %>%
  select(height_in_meters,mass) %>%
  arrange(mass) %>%
  View()

plot()

## Exploring Data structure and variable
data("msleep")
View(msleep)
glimpse(msleep)
head(msleep)
class(msleep)
class(msleep$name)
length(msleep)
length(msleep$name)
names(msleep)
unique(msleep$vore)
missing <- !complete.cases(msleep)
msleep[missing,]


## Select variables

starwars %>%
  select(name,height,mass)

starwars %>% select(1:3)
```

```
## Changing variable name
starwars %>%
  rename("characters"="name") %>%
  head()

## filter rows
starwars %>%
  select(mass,sex) %>%
  filter(mass<55 & sex=="male")


##Recode data

starwars %>%
  select(sex) %>%
  mutate(sex=recode(sex,"male"="man", "female"="women"))


## Dealing with mi8ssing data

mean(starwars$height, na.rm = T)


## create or change a new variable
starwars %>%
  mutate(height_m=height/100) %>%
  select(name,height,height_m)


## conditional statement

starwars %>%
  mutate(height_m=height/100) %>%
  select(name,height,height_m) %>%
  mutate(tallness=if_else(height_m<1,"short","tall"))



## Describing data, use msleep data set

min(msleep$awake)
max(msleep$awake)
range(msleep$awake)
IQR(msleep$awake)
mean((msleep$awake))
median((msleep$awake))
var((msleep$awake))
summary(msleep$awake)
msleep %>% select(awake,sleep_total) %>%
  summary()
```

Visualization

```
## Visualization

## The Grammar of plot or graphics: ggplot2
## data, mapping, geometry
## data must in data frame format
## Bar plots
require(ggplot2)
ggplot(data = starwars, mapping = aes(x=gender))+
  geom_bar()

## Histogram

starwars %>%
  drop_na(height) %>%
  ggplot(mapping = aes(x=height))+
  geom_histogram()


## Box plots

starwars %>%
  drop_na(height) %>%
  ggplot(mapping = aes(x=height))+
  geom_boxplot(fill="steelblue")+
  theme_bw()+
  labs(title = "Boxplot of Height",x="Height of Characters")


# density plot


starwars %>%
  drop_na(height) %>%
  filter(sex %in% c("male","female")) %>%
  ggplot(mapping = aes(x=height, color=sex, fill=sex))+
  geom_density(alpha=0.2)+
  theme_bw()+
  labs(title = "Boxplot of Height",x="Height of Characters")

# Scatter plot


starwars %>%
  filter(mass<200) %>%
  ggplot(mapping = aes(height, mass,color=sex))+
  geom_point(size=5,alpha=0.5)+
  theme_bw()+
  labs(title = "Height and mass by sex")
```

Writing to file/save/export data file

Everything we have done so far has only modified the data in R; the files have remained unchanged. Whenever we want to save our datasets to file, we need to use a write function in R.

To write our matrix to file in comma separated format (.csv), we can use the write.csv function. There are two required arguments: the variable name of the data structure you are exporting, and the path and filename that you are exporting to. By default the delimiter is set, and columns will be separated by a comma:

```
write.csv(starwars, file="starwars2.csv")
```

Similar to reading in data, there are a wide variety of functions available allowing you to export data in specific formats. Another commonly used function is write.table, which allows you to specify the delimiter you wish to use. This function is commonly used to create tab-delimited files.

NOTE: Sometimes when writing a dataframe with row names to file, the column names will align starting with the row names column. To avoid this, you can include the argument col.names = NA when writing to file to ensure all of the column names line up with the correct column values.

Reference and acknowledgement:

1. The materials used in this lesson are adapted from work that is Copyright © Data Carpentry (http://datacarpentry.org/), data camp, data quest, Kaggle, and Harvard Chan Bioinformatics Core (HBC) under the open access terms of the Creative Commons Attribution license (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2. The Book of R: A First Course in Programming and Statistics by Tilman M. Davies