Exploratory Data Analysis with R

Introduction to R - Part II

Xuemao Zhang
East Stroudsburg University

September 9, 2022

Outline

- R Programming
  - R condition statements
  - R loops
  - R functions
- Data Input
  - Working directory
  - Data input
  - Data Summaries
- Data output

R condition statement: `if`

- Condition statemants allow you to specify the execution of your code. They are extremely useful if you want to run a piece a code if a certain condition is met. The syntax of `if` statement is

```
if (test_expression) {statements;}
```

- The `any` function takes in logical vectors and returns `TRUE` if one or more elements if `TRUE`.
- The `which` functions takes in logical vectors and returns the index for the elements where the logical value is `TRUE`.

R condition statement: if

```
x = c(8, -3, 2, -6);
#any() checks if any of the elements of a vector are TRUE
if(any(x < 0)) print("x contains negative numbers");
```

```
## [1] "x contains negative numbers"
```

```
if(any(x < 0))
  {print("x contains negative numbers");
  print(x[which(x<0)]);
#which() function returns the positions of the elements
}
```

```
## [1] "x contains negative numbers"
## [1] -3 -6
```

R condition statement: `if...else`

- The conditional `if ... else` statement is used to test an expression similar to the `if` statement. However, if the `test_expression` is FALSE, the `else` part of the function will be evaluated. The syntax is

```
if (test_expression) {
        statement 1;
} else {
        statement 2;
}
```

R condition statement: if...else

```
x = c(8, -3, 2, -6);
if(any(x < 0))
  {print("x contains negative numbers");
  print(x[which(x<0)]);
} else
{print("x contains all positive numbers")}
```

```
## [1] "x contains negative numbers"
## [1] -3 -6
```

```
x = c(8, 3, 2, 6);
if(any(x < 0))
  {print("x contains negative numbers");
  print(x[which(x<0)]);
} else
{print("x contains all positive numbers")}
```

```
## [1] "x contains all positive numbers"
```

R condition statement: `if...else`

- We can also nest as many `if...else` statements as required (or desired).

```
k = 21;
if(k > 20){
  print("The number is greater than 20");
} else if (k < 20){
  print("The number is less than 20");
} else {
  print ("The number is equal to 20");
}
```

```
## [1] "The number is greater than 20"
```

R condition statement: `if...else`

- `ifelse()` function is a shorthand function to the traditional `if...else` statement. The syntax is

```
ifelse(test_expression, x, y)
```

- The `test_expression` must be a logical vector (or an object that can be coerced to logical).
- This returned vector has element from `x` if `test_expression` is `TRUE` or from `y` if `test_expression` is `FALSE`.

```
x = c(9,4,0,-4,-9);
sqrt(x);   #it gives warning
```

```
## Warning in sqrt(x): NaNs produced
```

```
## [1]   3   2   0 NaN NaN
```

```
sqrt(ifelse(x >= 0, x, NA))  # no warning
```

```
## [1]  3  2  0 NA NA
```

R loops: `for` loop

- Loops are used in programming to repeat a specific block of code.
- The `for` loop is used to execute repetitive code statements for a particular number of times. The syntax is

```
for (val in sequence)
{
statements;
}
```

```
for(i in 1:5)
{
  print(i);
}
```

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

R loops: `while` loop

- `While` loops begin by testing a condition. If it is `TRUE`, then they execute the statement. Once the statement is executed, the condition is tested again, and so forth, until the condition is `FALSE`, after which the loop exits.
- The syntax is

```
while(condition)
{expressions;}
```

```
i=0;
while (i < 5)
  {
    print(paste("i is", i));
    i=i+1;
  }
```

```
## [1] "i is 0"
## [1] "i is 1"
## [1] "i is 2"
## [1] "i is 3"
## [1] "i is 4"
```

R loops: `repeat` loop

- A `repeat` loop is used to iterate over a block of code multiple number of times. There is test expression in a repeat loop to end or exit the loop. Rather, we must put a condition statement explicitly inside the body of the loop and use the break function to exit the loop. Failing to do so will result into an infinite loop.
- The syntax is

```
counter = 1;
repeat {
        statements;
        if(test_expression){
                break;
        }
        counter = counter + 1;
}
```

R loops: `repeat` loop

```
x = 1;
repeat {
print(x);
x = x+1;
if (x == 6){
break;
}
}
```

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

R loops: `break` and `next`

- A `break` statement is used inside a loop (`repeat`, `for`, `while`) to stop the iterations and flow the control outside of the loop. It is used to exit a loop immediately if the `test_expression` is `TRUE`.

```
if (test_expression) {break;}
```

```
for (i in 1:100) {
if (i == 4){
break;
}
print(i);
}
```

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

R loops: `break` and `next`

- A `next` statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering `next`, the R parser skips further evaluation and starts **next iteration** of the loop. The syntax is

```
if (test_condition) {next;}
```

```
for (i in 1:5) {
if (i == 3){
next;
}
print(i);
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
```

R functions

- A function is a set of statements organized together to perform a specific task.
- Functions are used to logically break our code into simpler parts which become easy to maintain and understand.
- R has a large number of in-built functions.

```
x = 1:50;
x;
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
```

```
sum(x);  #Find the sum of the numbers
```

```
## [1] 1275
```

```
mean(x); #Find the average of the numbers
```

```
## [1] 25.5
```

R functions

- One of the great strengths of R is the user's ability to add functions.user can create their own functions. In fact, many of the functions in R are actually functions of functions. The structure of a function is

```
myfunction <- function(arg1, arg2, ... )
{
statements;
return(object);
}
```

- Function Name (myfunction): This is the actual name of the function. It is stored in R environment as an object with this name.
- Arguments (arg1, arg2, . . .): An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- Function Body (statements): The function body contains a collection of statements that defines what the function does.
- Return Value (return(object)): The return value of a function is the last expression in the function body to be evaluated.

R functions

```
# sum of the first n integers
summ<-function(n)
{
a=0;
for(i in 1:n)
{a=a+i;}
return(a);
}
summ(10)
```

```
## [1] 55
```

R functions

```
# power of a vector or matrix
getPower<-function(x ,power)
{
   out<-x^power;
   return(out);
}
getPower(x=3,power=2);
```

```
## [1] 9
```

```
getPower(x=c(1,2,3),power=2);
```

```
## [1] 1 4 9
```

Data Input

- 'Reading in' data is the first step of any real project/analysis
- R can read almost any file format, especially via add-on packages
- We are going to focus on simple delimited files first
  - tab delimited (e.g. '.txt')
  - comma separated (e.g. '.csv')
  - Microsoft excel (e.g. '.xlsx')

Common new user mistakes

1. **Working directory problems: trying to read files that R "can't find"**
   - RStudio can help, and so do RStudio Projects
2. Lack of comments in code
3. Typos (R is **case sensitive**, `x` and `X` are different)
   - RStudio helps with "tab completion" which can help correct your typos
4. Data type problems (is that a string or a number?)
5. Open ended quotes, parentheses, and brackets
6. Different versions of software

Working Directories

- R "looks" for files on your computer relative to the **working** directory
- Many people recommend not setting a directory in the scripts
  - assume you're in the directory the script is in
  - If you open an R file with a new RStudio session, it does this for you.
- If you do set a working directory, do it at the beginning of your script.
- Example of getting and setting the working directory:

```
getwd(); #obtain the current working directory
here::here();
setwd("../Lecture05");  #set (change) the working directory
```

Setting a Working Directory

- Setting the directory can sometimes be finicky
  - **Windows**: Default directory structure involves single backslashes (\), but R interprets these as "escape" characters. So you must replace the backslash with forward slashes ("/") or two backslashes ("\")
  - **Mac/Linux**: Default is forward slashes, so you are okay
- Typical directory structure syntax applies
  - ".." - goes up one level
  - "./" - is the current directory
  - "~" - is your "home" directory

Working Directory

Note that the `dir()` function interfaces with your operating system and can show you which files are in your current working directory.

You can try some directory navigation:

```
dir("./") # shows directory contents
```

```
##  [1] "Lecture05_R_Introduction2.aux" "Lecture05_R_Introduction2.r
##  [3] "Lecture05_R_Introduction2.pdf" "Lecture05_R_Introduction2.R
##  [5] "Lecture05_R_Introduction2.snm" "Lecture05_R_Introduction2.t
##  [7] "Lecture05_R_Introduction2.vrb" "mtcars.csv"
##  [9] "mtcars.txt"                    "mtcars_data.rda"
## [11] "mtcars2.rds"
```

```
dir("..")
```

```
##  [1] "88x31.png"   "data"        "Lecture01"   "Lecture02"   "Lec
##  [6] "Lecture04"   "Lecture05"   "Lecture06"   "Lecture07"   "Lec
## [11] "Lecture09"   "Lecture10"   "Lecture11"   "Lecture12"   "Lec
## [16] "Lecture14"   "Lecture15"   "Lecture16"   "Lecture17"   "Lec
## [21] "Lecture19"   "Lecture20"   "Lecture21"   "Lecture22"   "Lec
## [26] "Lecture24"   "Lecture25"   "Lecture26"   "Lecture27"   "Lec
```

Relative vs. absolute paths (From Wiki)

*An **absolute or full path** points to the same location in a file system, regardless of the current working directory. To do that, it must include the **root directory**.*

This means if I try your code, and you use absolute paths, it won't work unless we have the exact same folder structure where R is looking (bad).

*By contrast, a **relative path starts from some given working directory**, avoiding the need to provide the full absolute path. A filename can be considered as a relative path based at the current working directory.*

Setting the Working Directory

In RStudio, go to `Session --> Set Working Directory --> To Source File Location`

RStudio should put code in the Console, similar to this:

```
setwd("~/Math318/Lecture05");
```

Help and Commenting

- For any function, you can write `?FUNCTION_NAME`, or `help("FUNCTION_NAME")` to look at the help file:

```
?dir
help("dir")
```

- Commenting in Scripts

Commenting in code is super important. You should be able to go back to your code years after writing it and figure out exactly what the script is doing. Commenting helps you do this.

Data Input

- Easy way: R Studio features some nice "drop down" support, where you can run some tasks by selecting them from the toolbar.
  - For example, you can easily import text datasets using the "File –> Import Dataset" command. Selecting this will bring up a new screen that lets you specify the formatting of your text file.
  - After importing a datatset, you get the corresponding R commands that you can enter in the console if you want to re-import data.
- Write your code directly.
  - Using base R functions like `read.table()` and `read.csv()`.
  - Utilizing functions in the `readr` package called `read_delim()` and `read_csv()`.
  - The readr data import tools are up to two times faster for reading in large datasets.

Data Input

`read_delim()`: Read a delimited file into a data frame.

```
read_delim(file, delim, quote = "\"", escape_backslash = FALSE,
  escape_double = TRUE, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  comment = "", trim_ws = FALSE, skip = 0, n_max = Inf,
  guess_max = min(1000, n_max), progress = interactive())

# for example: `read_delim("file.txt",delim="\t")`
```

Data Input

- The filename is the path to your file, in quotes
- The function will look in your "working directory" if no absolute file path is given
- Note that the filename can also be a path to a file on a website (e.g. 'www.someurl.com/table1.txt')

Data Input

There is another convenient function for reading in CSV files, where the delimiter is assumed to be a comma:

`read_csv`

- It is from package ‘readr“

```
read_csv(file, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  quote = "\"", comment = "", trim_ws = TRUE, skip = 0,
  n_max = Inf, guess_max = min(1000, n_max),
  progress = show_progress(), skip_empty_rows = TRUE)
```

Data Input

- Here would be reading in the data from the command line, specifying the file path:

```
library(readr);
quiz1 = read_tsv("../data/quiz1.txt");
```

```
## Rows: 40 Columns: 7
## -- Column specification --------------------------------------
## Delimiter: "\t"
## dbl (7): ID, Q1, Q2, Q3, Q4, Q5, Q6
##
## i Use `spec()` to retrieve the full column specification for this
## i Specify the column types or set `show_col_types = FALSE` to qui
```

```
str(quiz1); #missing values are represented by NA
```

```
## spec_tbl_df [40 x 7] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ ID: num [1:40] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Q1: num [1:40] 8 8 10 6 10 NA 10 10 10 8 ...
##  $ Q2: num [1:40] 9 8 7 5 6 9 5 10 10 10 ...
##  $ Q3: num [1:40] 10 8 10 9 8 10 9 10 6 10 ...
##  $ Q4: num [1:40] 9 5 10 10 8 6 10 8 9 7 4
```

Data Input

```
quiz2 = read_csv("../data/quiz2.csv");
```

```
## Rows: 40 Columns: 7
## -- Column specification --------------------------------------
## Delimiter: ","
## dbl (7): ID, Q1, Q2, Q3, Q4, Q5, Q6
##
## i Use `spec()` to retrieve the full column specification for this
## i Specify the column types or set `show_col_types = FALSE` to qui
```

```
#quiz2 = read.csv("../data/quiz2.csv",header=TRUE, sep=",");
str(quiz2);
```

```
## spec_tbl_df [40 x 7] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ ID: num [1:40] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Q1: num [1:40] 8 8 10 6 10 NA 10 10 10 8 ...
##  $ Q2: num [1:40] 9 8 7 5 6 9 5 10 10 10 ...
##  $ Q3: num [1:40] 10 8 10 9 8 10 9 10 6 10 ...
##  $ Q4: num [1:40] 9.5 10 10 8 6 10 8 9 7 4 ...
##  $ Q5: num [1:40] 10 9 10 5 NA 10 10 10 8 10 ...
##  $ Q6: num [1:40] 8 8 8 NA NA NA 7 7 10 7 ...
```

Data Input

- The `read_delim()` and related functions returns a "tibble" is a `data.frame` with special printing, which is the primary data format for most data cleaning and analyses.
- Tibbles are a modern take on data frames. They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors).
- Mostly, `tbl` (tibbles) are the same as `data.frames`, except they don't print all lines.
- For more information, see Tibbles

Data Input

- Skimming a dataset

We will learn to do lots of Exploratory Data Analysis (EDA) in the course, one overview function, `skim()` like the R base function `summary()`, is very useful in data cleaning. To use this function, we need to load the package it comes from, `skimr`.

```
library(skimr)
```

- Let's import a dataset from a website

```
url='https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2021/2021-02-02/hbcu_all.csv'
```

```
hbcu=readr::read_csv(url)
skimr::skim(hbcu)
```

Data Input

- clean up the names of the variables using package `janitor`
  - Resulting names are **unique** and consist only of the _ character, numbers, and letters.

```
library(janitor)
hbcu <- clean_names(hbcu)
```

Data Input with `tbl_dfs`

- `read_csv` is popular but the base function `read.csv` is still largely used.
- When using the dropdown menu in RStudio, it uses `read_csv`, which is an improved version of reading in CSVs. It returns a `tbl` (tibble), that is a `data.frame` with improved printing and subsetting properties:

```
head(quiz2);
```

```
## # A tibble: 6 x 7
##      ID    Q1    Q2    Q3    Q4    Q5    Q6
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     8     9    10   9.5    10     8
## 2     2     8     8     8  10       9     8
## 3     3    10     7    10  10      10     8
## 4     4     6     5     9   8       5    NA
## 5     5    10     6     8   6      NA    NA
## 6     6    NA     9    10  10      10    NA
```

```
class(quiz2);
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

Data Input

```
quiz2;
```

```
## # A tibble: 40 x 7
##       ID    Q1    Q2    Q3    Q4    Q5    Q6
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     1     8     9    10   9.5    10     8
##  2     2     8     8     8  10       9     8
##  3     3    10     7    10  10      10     8
##  4     4     6     5     9   8       5    NA
##  5     5    10     6     8   6      NA    NA
##  6     6    NA     9    10  10      10    NA
##  7     7    10     5     9   8      10     7
##  8     8    10    10    10   9      10     7
##  9     9    10    10     6   7       8    10
## 10    10     8    10    10   4      10     7
## # ... with 30 more rows
```

Base R: Data Input

- There are also data importing functions provided in base R (rather than the `readr` package), like `read.delim` and `read.csv`.
- These functions have slightly different syntax for reading in data, like `header` and `as.is`.
- However, while many online resources use the base R tools, the latest version of RStudio switched to use these new `readr` data import tools,
- But you can use whatever function you feel more comfortable with.

Base R: Data Input

```
quiz1 = read.csv("../data/quiz1.txt",header=TRUE, sep="");
str(quiz1);
```

```
## 'data.frame':    40 obs. of  7 variables:
##  $ ID: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Q1: int  8 8 10 6 10 9 10 10 10 8 ...
##  $ Q2: int  9 8 7 5 6 10 5 10 10 10 ...
##  $ Q3: int  10 8 10 9 8 10 9 10 6 10 ...
##  $ Q4: num  9.5 10 10 8 6 10 8 9 7 4 ...
##  $ Q5: int  10 9 10 5 NA NA 10 10 8 10 ...
##  $ Q6: int  8 8 8 NA NA NA 7 7 10 7 ...
```

```
head(quiz1);
```

```
##   ID Q1 Q2 Q3   Q4 Q5 Q6
## 1  1  8  9 10  9.5 10  8
## 2  2  8  8  8 10.0  9  8
## 3  3 10  7 10 10.0 10  8
## 4  4  6  5  9  8.0  5 NA
## 5  5 10  6  8  6.0 NA NA
## 6  6  9 10 10 10.0 NA NA
```

Base R: Data Input

```
quiz2 = read.csv("../data/quiz2.csv",header=TRUE, sep=",");
str(quiz2);
```

```
## 'data.frame':    40 obs. of  7 variables:
##  $ ID: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Q1: int  8 8 10 6 10 NA 10 10 10 8 ...
##  $ Q2: int  9 8 7 5 6 9 5 10 10 10 ...
##  $ Q3: int  10 8 10 9 8 10 9 10 6 10 ...
##  $ Q4: num  9.5 10 10 8 6 10 8 9 7 4 ...
##  $ Q5: int  10 9 10 5 NA 10 10 10 8 10 ...
##  $ Q6: int  8 8 8 NA NA NA 7 7 10 7 ...
```

```
head(quiz2);
```

```
##    ID Q1 Q2 Q3   Q4 Q5 Q6
## 1   1  8  9 10  9.5 10  8
## 2   2  8  8  8 10.0  9  8
## 3   3 10  7 10 10.0 10  8
## 4   4  6  5  9  8.0  5 NA
## 5   5 10  6  8  6.0 NA NA
## 6   6 NA  9 10 10.0 10 NA
```

Data Input: Excel format

- Sometimes, a data set is in Excel format. We can use the **readxl** package to access Excel files

```
library(readxl);
quiz3 = read_excel("../data/quiz3.xls", sheet=1);
# read in the first worksheet
str(quiz3);
```

```
## tibble [40 x 7] (S3: tbl_df/tbl/data.frame)
##  $ ID: num [1:40] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Q1: num [1:40] 8 8 10 6 10 NA 10 10 10 8 ...
##  $ Q2: num [1:40] 9 8 7 5 6 9 5 10 10 10 ...
##  $ Q3: num [1:40] 10 8 10 9 8 10 9 10 6 10 ...
##  $ Q4: num [1:40] 9.5 10 10 8 6 10 8 9 7 4 ...
##  $ Q5: num [1:40] 10 9 10 5 NA 10 10 10 8 10 ...
##  $ Q6: num [1:40] 8 8 8 NA NA NA 7 7 10 7 ...
```

```
head(quiz3);
```

```
## # A tibble: 6 x 7
##      ID    Q1    Q2    Q3    Q4    Q5    Q6
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

Data Input - Other Software

- **haven** package (https://cran.r-project.org/web/packages/haven/index.html) reads in SAS, SPSS, Stata formats
- **sas7bdat** reads .sas7bdat files

Data Summaries

- `nrow()` displays the number of rows of a data frame
- `ncol()` displays the number of columns
- `dim()` displays a vector of length 2: # rows, # columns

```
dim(mtcars);
```

```
## [1] 32 11
```

```
nrow(mtcars);
```

```
## [1] 32
```

```
ncol(mtcars);
```

```
## [1] 11
```

Data Summaries

- `colnames()` displays the column names (if any) and `rownames()` displays the row names (if any)
- Note that tibbles do not have rownames

```
colnames(mtcars);
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
## [11] "carb"
```

```
rownames(mtcars);
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetw
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
```

Data output

While its nice to be able to read in a variety of data formats, it's equally important to be able to output data in the R workspace to your hard/usb drive.

`write.table()`: prints its required argument `x` (after converting it to a `data.frame` if it is not one nor a `matrix`) to a file or connection.

```
write.table(x,file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Data output

`x`: the R `data.frame` or `matrix` you want to write

`file`: the file name where you want to R object written. It can be an absolute path, or a filename (which writes the file to your working directory)

`sep`: what character separates the columns?

- `sep =","` = .csv - Note there is also a `write.csv()` function
- `sep ="\t"` = tab delimited

`row.names`: setting this to TRUE or FALSE

Data output

- Note that `row.names=TRUE` would make the first column contain the row names, which is not very useful for Excel.

```
write.table(mtcars, file = "mtcars.txt", sep = "\t",
            row.names = TRUE, col.names = NA);
#saved as tab-separated text file
write.csv(mtcars, file = "mtcars.csv",row.names = TRUE);
```

You will find that the two files mtcars.txt and mtcars.csv are saved on your current directory.

More ways to save: write_rds

- If you want to save **one** object, you can use `readr::write_rds` to save to an `rds` file:

```
write_rds(mtcars, file="mtcars2.rds");
```

- To read this back in to R, you need to use `read_rds`, but **need to assign it**:

```
mtcars3 = read_rds(file="mtcars2.rds");
identical(mtcars, mtcars3); # test if they are the same
```

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

More ways to save: save

- The save command can save **a set of** R objects into an "R data file", with the extension .rda or .RData.

```
x = 5;
save(mtcars, x, file = "mtcars_data.rda");
```

- The opposite of save is load. The ls() command lists the items in the workspace/environment and rm removes them:

```
load('mtcars_data.rda');
ls();
```

```
##  [1] "getPower" "i"         "k"         "mtcars"    "mtcars3"   "quiz
##  [7] "quiz2"    "quiz3"     "summ"      "x"
```

```
rm(x);
# rm(list=ls(all=TRUE));  #remove all objects from the workspace
```

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.