Introduction to R for Research Methodology

Pramudita Satria Palar, Ph.D.

15/8/2022

1 Introduction

This tutorial is intended for the "Research Methodology" Course currently taught in the Graduate Program at the Faculty of Mechanical and Aerospace Engineering, Bandung Institute of Technology. Teaching statistics is not yet complete without computing, especially for research purposes. Since researchers will heavily use software for statistics, it makes sense that statistical computing should be taught complementary to theory. In this course, R is used as a medium for teaching statistics in the context of research. R is designed for statistics; hence, it suits the purpose of this course. So why not R or MATLAB? Besides the reasons mentioned above, the ease of installation for R is also one deciding factor. I previously used Python for the course, but I observed that Python is not suitable for students who want to learn basic (or advanced) statistics. I can use MATLAB, but I think it is better to teach programming languages (or call it software if you want) that are free and can be used by the students even after they graduate from the university.

This tutorial covers important topics that I believe will help students understand the next modules. As such, some important topics that are not so essential for this course are not covered. For example, this tutorial does not discuss looping (*loop* and *while*) because most of the time, we will not use looping; if it exists, its role would not be essential.

This document is written by using R Markdown in R studio.

2 Installation

The first step is to install R in your local system, which can be downloaded from https://cran.r-project.org/. Follow the procedure until R is successfully installed in your system. Next, please download R studio (https://www.rstudio.com/) because it greatly helps us in working with R. R Studio is a nice *Integrated Development Environment* for R, and it has a free version of it (which is already very nice). Now open R studio and try typing something in the R console, for example:

```
5+5
```

```
## [1] 10
```

Congratulations! You just typed your first line of code in R. The result will be automatically shown in your Console (i.e., `10`). We will heavily use R console to type short lines of code. R Script is typically used for longer code, especially if we want to make specific programs. Now try using R as a simple calculator for subtraction, addition, etc.:

```
2-5 # Substraction
```

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

```r
2*6 # Multiplication
```

```
## [1] 12
```

```r
9/1 # Division
```

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

```r
2^5 # Exponent
```

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

Please note that the symbol `#` indicates a comment. A comment will not be executed by R and is only shown to help readers to understand the code.

3 Variable assignment

Variable assignment in R is very important, just like in other programming languages. Variable assignment is essential because the value assigned to the variable can be called anytime it is needed. There are several methods to assign variables in R, that is, by using leftwards assignment (`=`, `<-`, `<<-`) or rightwards assignment (`->>` or `->`). For example,

```r
A <- 5
B = 6
7 -> C
```

In the above example, we assign a numerical value of `5`, `6`, and `7` to `A`, `B`, and `C` respectively. The method to assign the variable might differ, but all are equally valid. However, there is a convention in R practitioners that the symbol `<-` should be used for variable assignment. Make sure to get yourself used to `<-`, because for Python and MATLAB users, this seems to be an unconventional way for variable assignment. It is also not common to use both `<<-` and `->>` for variable assignment. The value can then be easily called by typing the name of the variable in the R console, for example

```r
A
```

```
## [1] 5
```

Another useful trick is to assign the same value for several variables at once:

```r
X_1 <- X_2 <- X_3 <- 6
```

You have to follow several rules because not all variable names are valid. Here are some important rules:

- A variable name must start with a letter and can be a combination of letter, number, period (.), and underscore(_).
- A variable name that starts with dot (.) should not be followed by a number.
- A variable name cannot start with an underscore or a number
- Variable naming in R is *case sensitive* (e.g., `yoi`,`YOI`, and `Yoi` are three different variables names).

- There should not be a space in a variable name.
- Some words are already reserved within R and cannot be used. For example, `TRUE`, `FALSE`, `NULL`, and `if` are not allowed.

Here are some examples of invalid names:

```
var a <- 5
_var <- 5
.1var <- 5
```

4 R Data Types

There are several data types in R that can be used for various purposes. At least six types exists: `logical`, `numeric`, `integer`, `complex`, `character`, and `raw`. In this course, only several data types are used (`complex`, which is for complex number, and `raw` are excluded). The type of a variable can be checked by using the function `class()`. For example,

```r
A <- 5
class(A) # Should return "numeric"
```

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

Let us delve into some of the data types one at a time.

4.1 Logical

There are only two possible values for `logical`, namely, `TRUE` or `FALSE`. Logical is the output of relational operators (i.e., "is A higher than B?") and can also be used for filtering values, such as to pick only values that are higher than a certain threshold. We will study relational operators later, but it is worth giving a simple example here:

```r
var_a <- 5
var_a > 7 # This operation will yield FALSE
```

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

4.2 Numeric and integer

The `numeric` data type is extremely important because statistical computing essentially deals with this type of data. It is possible for `numeric` to be a decimal or not. See the following example (both variables are `numeric`):

```r
var_n1 <- 132.445
class(var_n1)
```

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

```r
var_n2 <- 132
class(var_n2)
```

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

In contrast to `numeric`, `integer` needs to be defined by adding `L` at the end of the digit. For example:

```r
var_int <- 155L
class(var_int)
```

```
## [1] "integer"
```

```r
var_num <- 155
class(var_num)
```

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

Notice that `class(var_num)` will yield `numeric`, while `class(var_int)` will yield `integer`.

4.3 Character

This is how you save a letter or a string in a variable. A `character` is written with a *single quote* or a *double quote*. Both methods are valid, but the former is usually used for a letter while the former is a string. Take a look at the following example:

```r
var_char_1 <- 'C'
class(var_char_1)
```

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

```r
var_char_2 <- "Ciao"
class(var_char_2)
```

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

4.4 Complex

Complex variables are rarely used in R, but still mentioned here for the sake of completeness. A `complex` data type is defined by adding `i` to a number:

```r
var_com <- 4 + 2i # 4 is the real component and 2 is the imaginary component
class(var_com)
```

```
## [1] "complex"
```

5 Relational and logical operators

5.1 Relational operators

Relational operators test the relationship between two different entities. Consider the following inquiry: "Is A equals B?" (where both are numerical values). The answer to this question is either true or false; these are the only possible outputs from relational operators. The output is then `logical`, i.e., `TRUE` or `FALSE`. There are several relational operators in R:

- `<`, less than
- `>`, greater than
- `<=`, less than or equal to
- `>=`, greater than or equal to
- `==`, equal to
- `!=`, not equal to

Try the following example:

```r
8 > 2 # TRUE
```

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

```r
4 < 4 # FALSE
```

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

```r
4 <= 4 # TRUE
```

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

```r
8 >= 4 # TRUE
```

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

```r
9 == 1 # FALSE
```

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

```r
8 != 1 # TRUE
```

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

5.2 Logical Operators

There are three basic Boolean operators: **AND**, **OR**, and **NOT**. The aim is logical operators is to perform these Boolean operations. The output from logical operations is a logical `TRUE` or `FALSE`. The Boolean table is not discussed here, but the resource to study it is everywhere. So here we will focus only on how to perform logical operations in R:

- `!`: Logical NOT
- `&`: Element-wise logical AND
- `&&`: Logical AND
- `|`: Element-wise logical OR
- `||`: Logical OR

Let us see some examples. NOT `TRUE` is `FALSE`, then

```
!TRUE # Outputs FALSE
```

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

Next, let us try `&`:

```
TRUE & FALSE # & only yields TRUE when both are TRUE
```

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

Finally, let us try `|`:

```
FALSE | TRUE # | yields TRUE whenever one is TRUE
```

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

Since relational operators output `logical`, we can use them together with logical operators as in the following example:

```
(8 > 3) & (2 < 10) # TRUE & TRUE, outputting TRUE
```

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

The main difference between `&` and `&&` is that the former performs an element-wise operation

Actually, for the sake of simplicity, `TRUE` and `FALSE` can be written simply as `T` and `F`, respectively.

6 Functions in R

6.1 Built-in functions

A function consists of statements or multiple lines of code to perform a specific task. Our first example is `class()`, which is used to check the class type of a variable. There are many built-in functions in R, and more can be downloaded from the CRAN network (the user can even create custom functions). A function takes arguments and returns an output. The syntax of a generic function is as follows

`output <- function_name(arg_1,arg_2,...)`

That is, the arguments (`arg_1`,`arg2`,`...`) are passed to the function (`function_name`) to return a specific output. Most functions have default argument values that the user can change. A simple example is `mean()`, which computes the mean from a numerical vector.

```
vec_ex <- c(10,11,9,8,16)
result.mean <- mean(vec_ex)
print(result.mean)
```

```
## [1] 10.8
```

But how about if the data has empty inputs (i.e., `NA`). To that end, the function `mean()` can takes an extra argument `na.rm` to exclude entries with `NA`.

```
vec_ex_NA <- c(10,11,NA,9,8,16)
result.mean_NA <- mean(vec_ex,na.rm=TRUE)
print(result.mean_NA)
```

```
## [1] 10.8
```

6.1.1 Creating a user-defined function

The syntax to create a user-defined function is as follows:

```
function_name <- function(arg1, arg2, ...){
      # Statements
      return()
}
```

The purpose of `return()` is such that the function returns any output. Notice that all functions require the definition of output. The following simple example converts temperatures from Fahrenheit to Celcius:

```
fahrenheit_to_celsius <- function(temp_in_F) {
  temp_in_C <- (temp_in_F - 32) * 5 / 9
  return(temp_in_C)
}
```

To use this function, just call the function name as in the example below

```
C <- fahrenheit_to_celsius(50)
print(C)
```

```
## [1] 10
```

7 Vector, Matrix, Data Frame, and List

7.1 Vector

7.1.1 Creating a vector

Vector is one of the most useful data types in R. Notice that the "vector" here is not strictly a vector in a mathematical sense. Vector in R is a type of data structure that consists of a sequence of elements which share the same type. For example, a single vector is only of either `numeric` or `character` type. Combining values into a vector is done through a function `c()`. See one example below

```r
vec_num <- c(4,33.22,8,100) # Vector of `numeric`
print(vec_num)
```

```
## [1]   4.00  33.22   8.00 100.00
```

```r
vec_chr <- c("Taufiq","Matza","Ardanto") # Vector of `character`
print(vec_chr)
```

```
## [1] "Taufiq"  "Matza"   "Ardanto"
```

Use `class()` to check the type of these vectors. It is not possible to create a vector with multiple data types. R automatically converts all elements in a vector into a single data type. For example,

```r
vec_as <- c('A',3,34)
```

returns a `character` vector (use `class()`).

7.1.2 Accessing vector elements

Now that we have a vector, it is possible to call the elements in a vector by using a square bracket symbol `[]`, where the indices of elements are put inside the bracket. To call multiple elements, we need to write down the list of indices using `c()` again. Furthermore, we can call multiple elements but certain elements by adding `-`. See the code below and do not forget to practice:

```r
vec_num <- c(4,8,13,200) # A vector consisting of four elements

vec_num[1] # Call the first element
```

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

```r
vec_num[c(1,4)] # Call the first and the fourth element
```

```
## [1]   4 200
```

```r
vec_num[-2] # Call all elements excluding the second element
```

```
## [1]   4  13 200
```

```r
vec_num[-c(1,3)] # Call all elements excluding the first and the third element
```

```
## [1]   8 200
```

The next example is more complicated, but it is worth being told. Combining relational operators and Boolean logic, certain elements that satisfy the given condition can be called. The key here is to call the elements according to a vector of `logical`. That is, an element is called given `TRUE`, and `FALSE` for otherwise.

```
idx <- c(F,F,T,T) # TRUE are given only for the third and fourth element
vec_num[idx]
```

```
## [1]  13 200
```

The example above is not really useful in practice. Let us try again but only elements that are larger than a certain threshold are called:

```
vec_num[vec_num>12] # Call elements that are larger than 12
```

```
## [1]  13 200
```

The key to this trick is to understand that `vec_num>12` outputs a vector of `logical`. To be clear,

```
vec_num>12
```

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

To check the length of a vector, try to type `length(vec_num)` on your R console.

7.1.3 Modifying vector elements

Not just accessing elements, it is of course also possible to modify existing vector elements. The first step is to point out the indices of the target elements and then modify the values using another assignment. Let us use the following vector of characters:

```
vec_ex <- c('A','B','C','D')
print(vec_ex)
```

```
## [1] "A" "B" "C" "D"
```

```
vec_ex[1] <- 'Z' # Change the first element to Z
print(vec_ex)
```

```
## [1] "Z" "B" "C" "D"
```

```
vec_ex[c(2,3)] <- c('Waw','Wow') # Change the second and third elements
print(vec_ex)
```

```
## [1] "Z"   "Waw" "Wow" "D"
```

It is worth noting that the length of the elements to be modified should be the same with that of the new assignment.

7.2 Matrix

7.2.1 Creating a matrix from scratch

Matrix is another useful data structure frequently used in both scientific and statistical computing. Every time a matrix operation is needed, the operation is performed on a matrix data structure. The method to create a matrix in R is quite different compared to either Python or MATLAB, so it is important to try it by yourself and do a lot of practice.

The necessary function is `matrix()`, which creates a matrix from the given set of values. The complete syntax for `matrix()` is

`matrix(data,nrow,ncol,byrow,dimnames)`

The function `matrix()` takes minimal one argument (i.e., the `data`), but actually it is not a proper way to create a matrix. That is, a matrix is defined by the number of rows and the number of columns. Thus, the extra arguments `nrow` (number of rows) and `ncol` (number of columns) complete the definition of a matrix.

Let us begin with a simple example, the code

```
A <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

create the following matrix:

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

On the other hand, if an extra argument `byrow=TRUE` is added:

```
A <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3,byrow=TRUE)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

The output is the following matrix

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

It is not necessary to define both `nrow` and `ncol`. For example, if only the `nrow` is given then R automatically calculates the `ncol`.

7.2.2 Create a matrix from vectors

Another method of creating a matrix in R is to combine multiple vectors using `rbind()` (row bind) or `cbind()` (column bind) functions. For example, let us define two vectors `vec1` and `vec2`:

```
vec1 <- c(1,2,3)
vec2 <- c(4,5,6)
```

Now combine them into a matrix using `rbind()`

```
mat1 <- rbind(vec1, vec2)
print(mat1)
```

```
##      [,1] [,2] [,3]
## vec1    1    2    3
## vec2    4    5    6
```

and `cbind()`

```
mat2 <- cbind(vec1, vec2)
print(mat2)
```

```
##      vec1 vec2
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Notice that it is possible to combine more than two vectors. To check the dimension of a matrix, use `dim()`:

```
dim(mat2)
```

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

where the first and the second element indicate the number of rows and columns, respectively.

Not just vectors, two or more matrices can be combined using `rbind()` or `cbind()`. See one example below:

```
mat_a <- matrix(c(1,2,3,4),nrow=2) # Notice that ncol can be skipped here
mat_b <- matrix(c(5,6,7,8),ncol=2) # While for this, nrow is skipped

mat_c <- cbind(mat_a,mat_b)
print(mat_c)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

```
mat_d <- rbind(mat_a,mat_b)
print(mat_d)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    5    7
## [4,]    6    8
```

7.2.3 Row and column names

When defining a matrix, it is sometimes handful to put names to the rows and the columns. The function `matrix()` also takes `dimnames` as one of its argument. The input for `dimnames`should be a `list` (see `list` below) of length giving the row and column names, respectively. Let us put this into practice

```
row_names <- c("row1","row2","row3")
col_names <- c("col1","col2","col3")
A_names <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3,byrow=TRUE,dimnames=list(row_names,col_names))
print(A_names)
```

```
##      col1 col2 col3
## row1    1    2    3
## row2    4    5    6
## row3    7    8    9
```

7.2.4 Special matrix

Besides `matrix()`, there are special functions for creating special matrices such as an identity or diagonal matrix. For example, the function `diag()` is for creating an identity or diagonal matrix:

```
mat_diag <- diag(c(1,2,3,4)) # Fill this with the vector of elements in the diagonal
print(mat_diag)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    3    0
## [4,]    0    0    0    4
```

```
mat_id <- diag(4) # Use integer as the input to create an identity matrix
print(mat_id)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

How about all zeros matrix? The trick is to use `matrix()` but specify only a single element. The following example creates all-zeros matrix of size 4×4:

```
mat_z <- matrix(0,nrow=4,ncol=4)
print(mat_z)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

As guessed, the zero can be replaced with any other real numbers. Try replacing `0` with `10` and a all-ten matrix will be created.

7.2.5 Matrix indexing

As a side note, the function `matrix()` is also applicable for any data type and not just `numeric`. However, there is actually not so much use in creating a matrix for, say, `character`. A matrix is mostly used for saving numerical values, and this tutorial sticks to that.

To call a specific element, it is necessary to point out the row and the column where that element is located. Consider the matrix `A` again, a specific element is called by using `[r,c]`, where `r` and `c` are the index for the row and the column, respectively. For example,

```
A <- matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,ncol=3,byrow=TRUE)
A[1,2] # Call the element in the first row and second column
```

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

```
A[3,3] # Call the element in the third row and third column
```

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

Obviously, calling an element that is not within the matrix is not possible (for example, if the size of the matrix is 3×3, there will no be element in the fourth row).

Calling multiple elements is also possible using vectors (i.e., `c`). See an example below

```
A[c(1,2),1] # Call the elements on the first and second row, first column
```

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

```
A[3,c(1,3)] # Call the elements on the first and third column, third row
```

```
## [1] 7 9
```

```
A[c(1,3),c(1,3)] # Call the elements on the first and third row, and the first and third column
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    7    9
```

Another useful trick is to call all elements in certain rows or columns. MATLAB uses : to do the trick. In R, no special symbol is needed, as in the following example:

```
A[,2] # Access all elements at the second column
```

```
## [1] 2 5 8
```

```
A[3,] # Access all elements at the third row
```

```
## [1] 7 8 9
```

```r
A[,c(1,2)] # Access all elements at the first and second column
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    4    5
## [3,]    7    8
```

As a side note, when a single or column is accessed (e.g., `A[,2]` and `A[3,]`), the output is a vector (use `is.vector()` to check this). On the other hand, the output is a matrix if multiple columns or rows are accessed (e.g., `A[,c(1,2)]`)

7.3 List

List is a more general version of data frame. Indeed, there are many questions about the main differences between a data frame and a list. A list is the most flexible data structure. Compared to a data frame, there are more flexibilities in list including:

- The elements in a list can be anything, not restricted to vectors.
- The length of the elements in a list can be of an arbitrary length.

It is even possible for a list to save data frames as elements.

```r
list_ex <- list(numbers=c(1,2,3,4,5,6),char=c('A','B'),string=c("Dono","Kasino","Indro"))
list_ex
```

```
## $numbers
## [1] 1 2 3 4 5 6
##
## $char
## [1] "A" "B"
##
## $string
## [1] "Dono"   "Kasino" "Indro"
```

Notice that the elements in `list_ex` have different lengths, which is not possible in a data frame. Accessing elements in a list is similar to data frame, that is, by using the dollar sign (`$`), single bracket `[]`, or double bracket `[[]]`. For single and double bracket, the index of the element needs to be called instead of the name of the element,

The difference between single and double brackets is that the former returns a `list` while the latter returns the element on its original data type (e.g., a vector, a matrix, or a data frame). Notice that `$` will also return the original data type. For example,

```r
list_ex$numbers
```

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

```r
list_ex[1]
```

```
## $numbers
## [1] 1 2 3 4 5 6
```

```
list_ex[[1]]
```

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

For practice, use `class()` to check the data type.

The example below shows a more complicated list where a matrix is also included as one element:

```
mat_ex <- matrix(c(1,2,3,4),nrow=2,ncol=2)
list_mat <- list(char=c('A','B'),string=c("Dono","Kasino","Indro"),matrix=mat_ex)
```

7.4 Data Frame

7.4.1 Creating a Data Frame

In addition to vector, another highly useful fundamental data structure in R is *Data frame*. Imagine it as a spreadsheet table, that is what a data frame is. A data frame can hold any data type and not limited to a single data type. It is even possible to include a vector inside a data frame, which is essentially how it should be. The function `data.frame()` creates data frames as in the following example:‘

```
var_df <- data.frame(name=c("Dono","Kasino","Indro"),IPK=c(2.3,4,1.5),sex=c('M','M','M'))
```

Here are some useful functions applicable to data frames:

- `names()`, returns the names of columns.
- `nrow()`, returns the number of rows
- `ncol()`, returns the number of columns
- `head()`, returns the first parts of a data frame
- `tail()`, returns the last parts of a data frame

Notice that both `head()` and `tail()` are also applicable to vectors and matrix.

7.4.2 Accessing a data frame

There are several methods for accessing a data frame. The first method is to treat is like a list, that is, use either `[]`,`[[]]`, or `$` to access columns/elements. Here are the differences between the three, let us try calling `name` from `var_df`:

```
var_df["name"] # Return elements in "name" as a data frame
```

```
##     name
## 1   Dono
## 2 Kasino
## 3  Indro
```

```
var_df$name # Return elements in "name" as vector
```

```
## [1] "Dono"   "Kasino" "Indro"
```

```
var_df[["name"]] # Equivalent to `$`
```

```
## [1] "Dono"   "Kasino" "Indro"
```

8 Downloading extra packages

R packages are collections of functions developed by the community to be used by the community. Some features are not provided by the R built-in functions, which is why external packages are needed. For example, the function to calculate the skewness data is not directly provided, which is why specialized packages such as `moments` exist.

Most packages are available in the R official repository (CRAN) or other sources such as GitHub. The most straightforward way to download R packages from CRAN is to use `install.packages()` function. The following code install the package `moments` from CRAN:

```
install.packages("moments")
```

Once you download the package, use `packageDescription()` and `help(package="packages")` to find the description and all available functions.

```
packageDescription("moments")
```

```
## Package: moments
## Type: Package
## Title: Moments, Cumulants, Skewness, Kurtosis and Related Tests
## Version: 0.14.1
## Date: 2015-01-05
## Author: Lukasz Komsta <lukasz.komsta@umlub.pl>, Frederick Novomestky
##         <fnovomes@poly.edu>
## Maintainer: Lukasz Komsta <lukasz.komsta@umlub.pl>
## Description: Functions to calculate: moments, Pearson's kurtosis,
##         Geary's kurtosis and skewness; tests related to them
##         (Anscombe-Glynn, D'Agostino, Bonett-Seier).
## License: GPL (>= 2)
## URL: https://www.r-project.org, http://www.komsta.net/
## Packaged: 2022-05-02 10:08:08 UTC; hornik
## NeedsCompilation: no
## Repository: CRAN
## Date/Publication: 2022-05-02 13:01:55 UTC
## Built: R 4.2.0; ; 2022-05-04 00:14:49 UTC; windows
##
## -- File: C:/Users/pramsatriapalar/AppData/Local/R/win-library/4.2/moments/Meta/package.rds
```

```
help(package="moments")
```

To use packages in the current session, use `library()` as in the following

```
library(moments)
```

9 Final thoughts

This is the beginning of your journey into R. R is probably the best programming language out there to do statistical computing for many reasons. It is free, easy to use, and has a large community of users. Its popularity is growing, and I believe it will still do so in the near future. Let me close this tutorial with some pointers for your future journey:

- Try my other learning modules. If you take my Research Methodology course, you should read and try all of them.
- There are so many free reading materials out there for learning R. Make sure to explore them.
- Explore packages that suit your need. For example, if you want to use R's advanced plotting, try `ggplot2`. There are many packages out there in CRAN repository for many purposes, spanning from geostatistics, social sciences, general regression models, to Bayesian statistics.
- Try finishing your own project using R. It might be a simple weekend project or even your graduate thesis. It will greatly enhance your skills and experiences in using R.
- Later on, you can try learning R Markdown, which is an excellent tool for writing beautiful and reproducible scientific reports.

Furthermore, I need to make some notes on the R language itself here:

- R is not designed for general scientific computing. MATLAB and R are usually better for this purpose. I do not even think of using R to solve partial differential equations.
- Use R with other languages, such as MATLAB, Python, or Julia, because all languages have their strengths and weaknesses. Do not stick to just one language, but surely R will be a great addition to your arsenal.

Hopefully, this short tutorial serves as a good introductory document to new R users: including **you**. Thanks for reading this document.