R Programming

A LANGUAGE AND ENVIRONMENT FOR STATISTICAL COMPUTING AND GRAPHICS.

Agenda

- Importing data
  - Reading Tabular Data files
  - Reading CSV files
  - Importing data from excel
- Loading and storing data with clipboard
- Loading R data objects
- Writing data to file
- Writing text and output from analyses to file

Agenda

- Manipulating Data
- Selecting rows/observations
- Rounding Number
- Creating string from variable
- R Programming
  - While loop
  - If Statement
  - For loop
  - Arithmetic operations

Reading Tabular Data files

- If you have a .txt or a tab-delimited text file, you can easily import it with the basic R function read.table(). In other words, the contents of your file will look similar to this
  - 1 6 a
  - 2 7 b
  - 3 8 c
  - 4 9 d
  - 5 10 e
- and can be imported as follows:
  - # To Locate the Current Working Directory
  - getwd()
  - Company.employees <- read.table("EmployeeSales.txt", TRUE, sep = " ", quote="\"")
  - Company.employees
  - write.table(Company.employees,"EmployeeSales2.txt")

Reading Tabular Data files

- R Read table Function – testing arguments
  - employees <- read.table("EmployeeSales.txt", TRUE, quote="\"",
  - na.strings = c("NA",""), strip.white = TRUE,
  - blank.lines.skip = TRUE)
  - print(employees)
  - # Below is the example for using strip.white
  - employees <- read.table("empdata.csv", TRUE, quote="\"",
  - na.strings = c("NA",""), sep=",", strip.white = T,
  - blank.lines.skip = TRUE)

  - strip.white: If the sep argument is not equal to "" then you may use this Boolean value to trim the extra leading & tailing white spaces from the character field.
  - blank.lines.skip: A Boolean value that specifies whether you want to skip/ignore the blank lines or not.
  - na.strings: A character vector specifying values that should be read as NA

Reading Tabular Data files

- Testing R Read table arguments
  - In this example we will show you, How to rename the column names, skip the number of rows, changing the default factors.

  - employeeNames <- c("Employee_ID", "First Name", "Last Name", "Education", "Profession","Salary","Sales")
  - employees <- read.table("EmployeeSales.txt", TRUE, quote="\"",
  - strip.white = TRUE, skip = 3,
  - as.is = c(TRUE, TRUE, FALSE, FALSE, TRUE),
  - col.names = employeeNames,
  - blank.lines.skip = TRUE)
  - print(employees)
  - print(str(employees))

Reading Tabular Data files

- col.names: A Character vector that contains the column names for the returned data frame

- as.is: Please specify the Boolean vector of same length as the number of column. This argument will convert the character values to factors based on the Boolean value. For example, we have two columns (FirstName, Occupation) and we uses as.is = c(TRUE, FALSE). This will keep the FirstName as character (not an implicit factor), and Occupation as Factors

- skip: Please specify the number of rows you want to skip from text file before beginning the data read. For example, if you want to skip top 3 records, use skip = 3

Reading and writing CSV files

- Example of importing CSV data are provided below.
- Read from a Comma Delimited Text File
  - # first row contains variable names
  - # “row.names” assigns the variable id to row names
  - # If we do not specify row.names then it would create another running serial number to it

  - mydata <- read.csv("empdata.csv", header=TRUE, row.names="Employee_ID")
  - mydata

- Write to Comma Delimited Text File
  - write.csv(mydata, "MyEmpData.csv")

Importing data from excel

- From Excel
  - You can use the xlsx package to access Excel files. The first row should contain variable/column names.

  - write.xlsx(mydata, "EmployeeSales.xlsx", row.names= F)
  - setwd("C:/Users/Atin/Desktop/RProgramming/WorkingD")
  - install.packages("xlsx")
  - library(xlsx)
  - mydata <- read.xlsx("EmployeeSales.xlsx", 1)
  - mydata
  - # read in the worksheet named mysheet
  - mydata <- read.xlsx("EmployeeSales.xlsx", sheetName = "Sheet 1")
  - mydata

Loading and storing data with clipboard

- You can actually copy the contents to your clipboard and import them quickly into R.
- To do this, you can either use the readClipboard() or read.table() functions:
  - readClipboard() #Only on Windows
  - read.table(file="clipboard“, sep=“\t”)

- writeClipboard
  - R has a function writeClipboard that does what the name implies. However, the argument to writeClipboard may need to be cast to a character type. For example the code
  - > x <- "hello world"
  - > writeClipboard(x)
  - copies the string “hello world” to the clipboard as expected. However the code
  - > x <- 3.14
  - > writeClipboard(x)
  - produces the error message

Loading and storing data with clipboard

- Here we will use a package for reading data from Clipboard

- install.packages("psych")
- library(psych)

- My_data= read.clipboard.csv(header=TRUE,sep=',')
- My_data

- print(read.clipboard.tab(header=TRUE,sep='\t'))

Loading R data objects

- Writing data, in csv or Excel file formats, is the best solution if you want to open these files with other analysis software, such as Excel.
- However this solution doesn't preserve data structures, such as column data types (numeric, character or factor).
- In order to do that, the data should be written out in R data format.
- Save one object to a file
  - It's possible to use the function saveRDS() to write a single R object to a specified file (in rds file format).
  - The object can be restored back using the function readRDS().
- The simplified syntax for saving and restoring is as follow:
  - my_data= read.clipboard.csv(header=TRUE,sep=',')
  - saveRDS(my_data, file = "my_data.rds")
  - rm(my_data)
  - my_data
  - my_data = readRDS(file = "my_data.rds")
  - my_data

Loading R data objects

- In the R code below, we'll save the mtcars data set and restore it under different name:
  - # Save a single object to a file
  - saveRDS(mtcars, "mtcars.rds")
  - # Restore it under a different name
  - my_data <- readRDS("mtcars.rds")
- Save multiple objects to a file
  - The function save() can be used to save one or more R objects to a specified file.
  - The function can be read back from the file using the function load().

  - a=1;b=2;c=3;d="Sumit";e=T
  - a;b;c;d;e
  - save(a, file = "data.RData") # Saving on object in RData format
  - # Save multiple objects
  - save(b,c,d,e, file = "data.RData")
  - load("data.RData") # To load the data again
  - a;b;c;d;e

Writing text and output from analyses to file

- Sometime we require to redirect our output to some physical file for later analysis.
- The sink() function will redirect output to a file instead of to the R terminal.
- To return output to the terminal. , call sink() without any arguments

```
# Start writing to an output file
sink('analysis-output.txt')
a=1;b=2;c=3;d="Sumit";e=T
a;b;c;d;e

sink() # return out back to termincal
a;b;c;d;e
```

Writing text and output from analyses to file

- # Append to the file
- sink('analysis-output.txt', append=TRUE)
- cat("Some more stuff here...\n")
- sink()

Manipulating Data

- The dplyr package is one of the most powerful and popular package in R.
- This package was written by the most popular R programmer Hadley Wickham.
- The dplyr is a powerful R-package to
  - manipulate,
  - clean and
  - summarize unstructured data.
- The package "dplyr" comprises many functions that perform mostly used data manipulation operations such as
  - applying filter,
  - selecting specific columns,
  - sorting data,
  - adding or deleting columns and
  - aggregating data.
- Another most important advantage of this package is that it's very easy to learn and use dplyr functions.

Manipulating Data

- To install the dplyr package, type the following command.
  - install.packages("dplyr")
- To load dplyr package, type the command below
  - library(dplyr)
- Important dplyr Functions to remember

dplyr Function	Description	Equivalent SQL
select()	Selecting columns (variables)	SELECT
filter()	Filter (subset) rows.	WHERE
group_by()	Group the data	GROUP BY
summarise()	Summarise (or aggregate) data	-
arrange()	Sort the data	ORDER BY
join()	Joining data frames (tables)	JOIN
mutate()	Creating New Variables	COLUMN ALIAS

Manipulating Data

- Sample Data:

- How to load data:
  - *mydata = read.csv("sampledata.csv")*

- The sample_n function selects random rows from a data frame. The second parameter of the function tells R the number of rows to select.
  - sample_n(mydata,3)
- The sample_frac function returns randomly N% of rows. In the example below, it returns randomly 10% of rows.
  - sample_frac(mydata,0.1)

Manipulating Data

- The distinct function is used to eliminate duplicates.
- Remove Duplicate Rows based on all the variables (Complete Row)
  - x1 = distinct(mydata)

- Remove Duplicate Rows based on a variable
- The .keep_all function is used to retain all other variables in the output data frame.
  - distinct(mydata, Index, .keep_all= TRUE)

- Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".
  - select(mydata, Index, State:Y2008)

Manipulating Data

- The minus sign before a variable tells R to drop the variable.
  - select(mydata, -Index, -State)
- The starts_with() function is used to select variables starts with an alphabet.
  - select(mydata, starts_with("Y"))
- Selecting Variables contain 'I' in their names
  - select(mydata, contains("I"))
- The code below keeps variable 'State' in the front and the remaining variables follow that.
  - select(mydata, State, everything())

Manipulating Data

- In the following code, we are renaming 'Index' variable to 'Index1'.
  - rename(mydata, Index1=Index)
- Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.
  - filter(mydata, Index == "A")
- Suppose you need to apply 'AND' condition. In this case, we are picking data for 'A' and 'C' in the column 'Index' and income greater than 1.3 million in Year 2002.
  - filter(mydata, Index %in% c("A", "C") & Y2002 >= 1300000)
- The 'I' denotes OR in the logical condition. It means any of the two conditions.
  - filter(mydata, Index %in% c("A", "C") | Y2002 >= 1300000)
- The "!" sign is used to reverse the logical condition.
  - filter(mydata, !Index %in% c("A", "C"))

Manipulating Data

- Calculating mean and median for the variable Y2015.
  - summarise(mydata, Y2015_mean = mean(Y2015), Y2015_med=median(Y2015))
- sorting data by multiple variables.
  - arrange(mydata, Index, Y2011)

Selecting rows/observations

- Suppose you have a data frame, df - consisting of three vectors that consist of information such as height, weight, age.
  - df <- data.frame(c(183, 85, 40), c(175, 76, 35), c(178, 79, 38))
  - names(df) <- c("Height", "Weight", "Age")

- # All Rows and All Columns
- df[,]
- # First row and all columns
- df[1,]
- # First two rows and all columns
- df[1:2,]

Selecting rows/observations

- # First and third row and all columns
- df[c(1,3),]
- # First Row and 2nd and third column
- df[1, 2:3]
- # First, Second Row and Second and Third Column
- df[1:2, 2:3]

Rounding Number

- Although R can calculate accurately to up to 20 digits, you don't always want to use that many digits.
  - format(22/7,nsmall=20)
- In this case, you can use a couple functions in R to round numbers.
- To round a number to two digits after the decimal point, use the round() function as follows:
  - > round(22/7,digits=2)
  - [1] 123.46

Rounding Number

- You also can use the round() function to round numbers to multiples of 10, 100, and so on. For that, you just add a negative number as the digits argument:
  - > round(-123.456,digits=-2)
  - [1] -100
- If the first digit that is dropped is exactly 5, it rounds to the nearest even number. round(1.5) and round(2.5) both return 2, for example, and round(-4.5) returns -4.

Creating string from variable

- The two common ways of creating strings from variables are
  - the paste function and
  - the sprintf function.

- paste is more useful for vectors, and
- sprintf is more useful for precise control of the output.

Creating string from variable

- a <- "apple"
- b <- "banana"

- # Put a and b together, with a space in between:
- paste(a, b)
- #> [1] "apple banana"

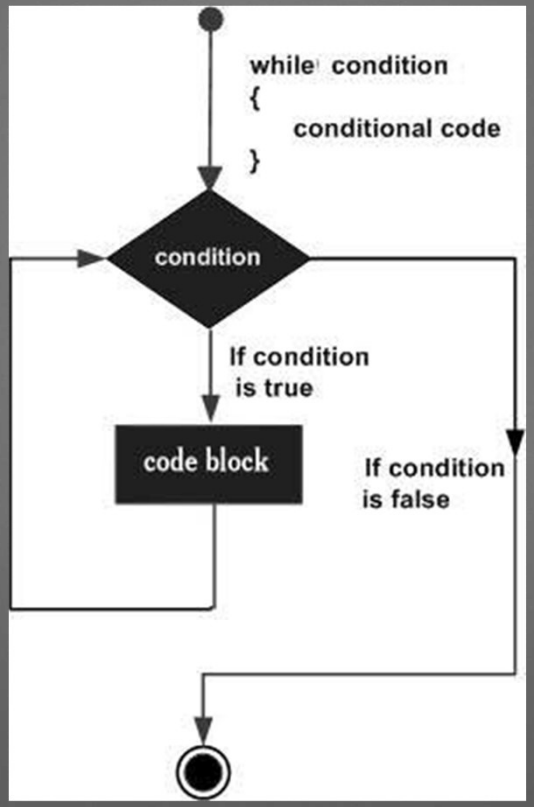
Creating string from variable

- # With no space, use sep="", or use paste0():
- paste(a, b, sep="")
- #> [1] "applebanana"

- # With a comma and space:
- paste(a, b, sep=", ")
- #> [1] "apple, banana"

R Programming - While loop

- The While loop executes the same code again and again until a stop condition is met.
- The basic syntax for creating a while loop in R is −
  - while (test_expression) {
  - statement
  - }



R Programming - While loop

- Example
  - v <- c("Hello","while loop")
  - cnt <- 2

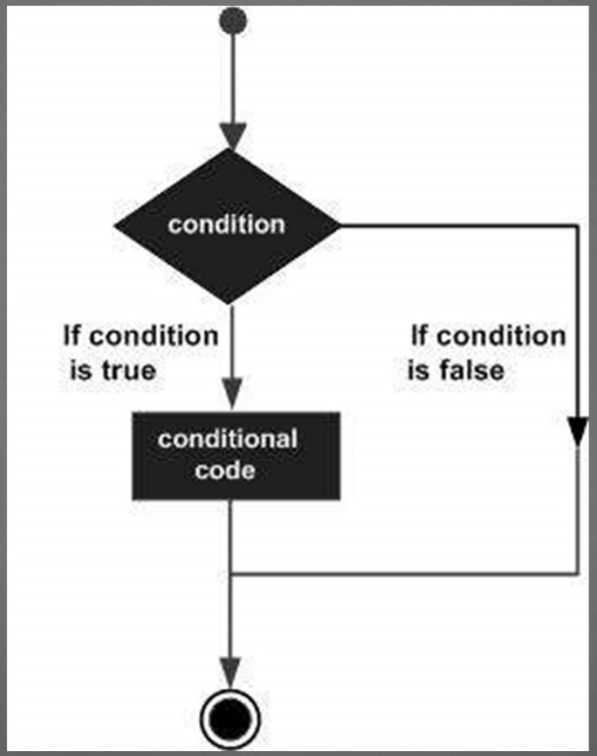
  - while (cnt < 7) {
  - print(v)
  - cnt = cnt + 1
  - }

- When the above code is compiled and executed, it produces the following result −

  - [1] "Hello" "while loop"
  - [1] "Hello" "while loop"
  - [1] "Hello" "while loop"
  - [1] "Hello" "while loop"
  - [1] "Hello" "while loop"

R Programming - If Statement

- Following is the general form of a typical decision making structure found in most of the programming languages



R Programming - If Statement

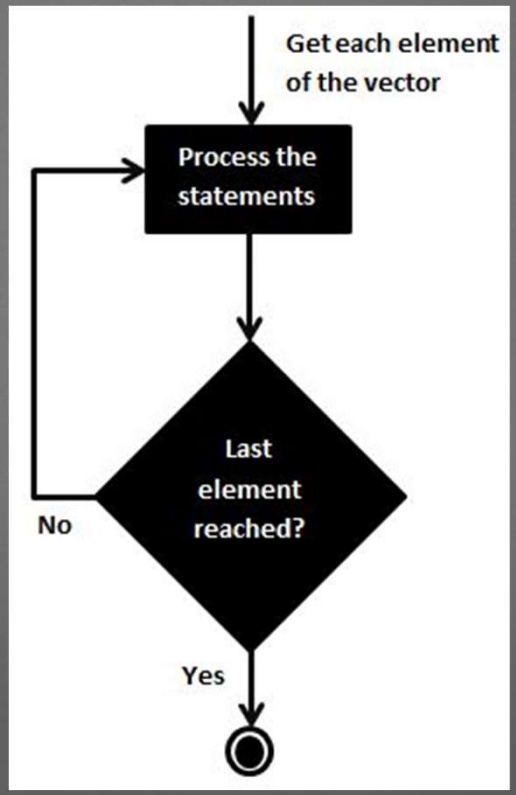
- The basic syntax for creating an if statement in R is −
  - if(boolean_expression) {
  - // statement(s) will execute if the boolean expression is true.
  - }
- Example
  - x <- 30L
  - if(is.integer(x)) {
  - print("X is an Integer")
  - }

R Programming - If Statement

- The basic syntax for creating an if...else statement in R is −
  - if(boolean_expression) {
  - // statement(s) will execute if the boolean expression is true.
  - } else {
  - // statement(s) will execute if the boolean expression is false.
  - }
- Example
  - x <- c("what","is","truth")
  - if("Truth" %in% x) {
  - print("Truth is found")
  - } else {
  - print("Truth is not found")
  - }

R Programming - For loop

- A For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- The basic syntax for creating a for loop statement in R is −
  - for (value in vector) {
  - statements
  - }



R Programming - For loop

- R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

- Example
  - v <- LETTERS[1:4]
  - for (i in v) {
  - print(i)
  - }
- When the above code is compiled and executed, it produces the following result –
  - [1] "A"
  - [1] "B"
  - [1] "C"
  - [1] "D"

Arithmetic operations

- The R Arithmetic operators includes operators like
  - Arithmetic Addition,
  - Subtraction,
  - Division,
  - Multiplication,
  - Exponent,
  - Integer Division and
  - Modulus.
- All these operators are binary operators, which means they operate on two operands.

R Programming - Arithmetic operations

- Below table shows all the Arithmetic Operators in R Programming language with examples.

R ARITHMETIC OPERATORS	OPERATION	EXAMPLE
+	Addition	15 + 5 = 20
–	Subtraction	15 – 5 = 10
*	Multiplication	15 * 5 = 75
/	Division	15 / 5 = 3
%/%	Integer Division – Same as Division but it will return the integer value by flooring the extra decimals	16 %/% 3 = 5. If you divide 16 with 3 you get 5.333 but the Integer division operator will trim the decimal values and outputs the integer
^	Exponent – It returns the Power of One variable against the other	15 ^ 3 = 3375 (It means 15 Power 3 or 10^3).
%%	Modulus – It returns the remainder after the division	15 %% 5 = 0 (Here remainder is zero). If it is 17 %% 4 then it will be 1.

Thanks