# R Statistical Programming Language - Notes

## Introduction to R :

**What Is R?**

R is open source computer software for statistical analysis. It is made available through Internet under the General public License (GPL) which means that the software is freely available to everyone for use.

It is mainly used for statistical computing. It offers a range of algorithms which are heavily used in machine learning domain such as time series analysis, classification, clustering, linear modeling, etc.

R is also an environment that includes a suite of software packages that can be used for performing a numerical calculation to chart plotting to data manipulation. R is heavily used in statistical research projects. R is very similar to the S programming language.

R is compiled and runs on UNIX, Windows and Linux platforms. R has a large number of data structures, operators and features. It offers from arrays to matrices to loops to recursion along with integration with other programming languages such as C, C++, and Fortran. C programming language can be used to update R objects directly.

**Benefits of R:**

Along with the benefits listed below, R is:

- Straightforward to learn.
- A large number of packages are available for statistical, analytics and graphics which are open-source and free.
- A wealth of academic papers with their implementation in R are available.
- World's top universities teach their students the R programming language, therefore, it has now become an accepted standard and thus, R will continue to grow.
- Integration capabilities with other languages.
- Plus there is a large community support.

**Applications of R Programming in Real World:**

1. Data Science Harvard Business Review named data scientist the "sexiest job of the 21st century". Glass-door named it the "best job of the year" for 2016. With the advent of IoT(Internet of Things) devices creating terabytes and terabytes of data that can be used to make better decisions. Simply explained, a data scientist is a statistician with an extra asset: computer programming skills. Programming languages like R give a data scientist superpowers that allow them to collect data in real-time, perform statistical and predictive analysis, create visualizations and communicate actionable results to stakeholders. Most courses on data science include R in their curriculum because it is the data scientist's favourite tool.
2. Statistical computing R is the most popular programming language among statisticians. In fact, it was initially built by statisticians for statisticians. It has a rich package repository with more than 9100 packages with every statistical function you can imagine. R's expressive syntax allows researchers - even those from non computer science backgrounds to quickly import, clean and analyze data from various data sources. R also has charting capabilities, which means you can plot your data and create interesting visualizations from any data set.
3. Machine Learning R has found a lot of use in predictive analytics and machine learning. It has various package for common ML tasks like linear and non-linear regression, decision trees, linear and non-linear classification and many more. Everyone from machine learning enthusiasts to researchers use R to implement machine learning algorithms in fields like finance, genetics research, retail, marketing and health care.

**Limitations of R:**

There are a handful of limitations too:

- R isn't as fast as C++, plus security and memory management is an issue too.
- R has a large number of name spaces, sometimes that could appear to be too many. However, it is getting better.
- R is a statistician language thus it is not as natural as Python. It's not as straightforward to create OOP as it is with Python.

## How To Install R?

You can install R on:

- Ubunto
- Mac
- Windows
- Fedora
- Debian
- SLES
- OpenSUSE

**The first step is to download R:**

- Open an internet browser
- Go to www.r-project.org.
- The latest R version at the point of writing this article is 3.6.3 (Holding the Windsock). It was released on 2020–02–29.

These are the links:

- Download R for Linux - https://cran.r-project.org/bin/linux/
- Download R for (Mac) OS X - https://cran.r-project.org/bin/macosx/
- Download R for Windows - https://cran.r-project.org/bin/windows/

**Where To Code R?**

There are multiple graphical interfaces available. I highly recommend R-Studio

Download RStudio Desktop:

- Download RStudio from https://rstudio.com/products/rstudio/download/
- RStudio Desktop Open Source License is Free
- You can learn more here: https://rstudio.com/products/rstudio/#rstudio-desktop
- RStudio requires R 3.0.1+.

It usually installs R Studio in the following location if you are using Windows:

C:\Program Files\RStudio

**R Script:**

R script is where a data scientist can write the statistical code. It is a text file with an extension .R e.g. we can call the script as tutorial.R

We can create multiple R scripts in a package.

As an instance, if you have created two R scripts:

1. blog.R
2. publication.R

And if you want to call the functions of publication.R in blog.R then you can use the source("target R script") command to import publication.R into blog.R*:*source("publication.R")

## What Are The Different R Data Types?

It is vital to understand the different data types and structures in R to be able to use the R programming language efficiently. This section will illustrate the concepts.

**Data Types:**

These are the basic data types in R:

1. **character**: such as "abc" or "a"
2. **integer**: such as 5L
3. **numeric**: such as 10.5
4. **logical**: TRUE or FALSE
5. **complex**: such as 5+4i

We can use the type of(variable) to find the type of a variable.

To find the meta data, such as attributes of a type, use the attributes(variable) command.

**Data Structures:**

A number of data structures exist in R. The most important data structures are:

1. **vector**: the most important data structure that is essentially a collection of elements.
2. **matrix**: A table-like structure with rows and columns
3. **data frame**: A tabular data structure to perform statistical operations
4. **lists**: A collection that can hold a combination of data types.
5. **factors**: to represent categorical data

**How To Declare Variables?**

We can create a variable and assign it a value. A variable could be of any of the data types and data structures that are listed above. There are other data structures available too. Additionally, a developer can create their own custom classes.

A variable is used to store a value that can be changed in your code.

As a matter of understanding, it is vital to remember what an environment in R is. Essentially, an environment is where the variables are stored. It is a collection of pairs where the first item of the pair is the symbol (variable) and the second item is its value.

Environments are hierarchical (tree structure), hence an environment can have a parent and multiple children. The root environment is the one with an empty parent.

We have to declare a variable and assign it a value using →

```
x <- "my variable"
print(x)
```

## Functions:

Sometimes we want the code to perform a set of tasks. These tasks can be grouped as functions. The functions are essentially objects in R. Arguments can be passed to a function in R and a function can return an object too. R is shipped with a number of in-built functions such as length(), mean(), etc.

Every time we declare a function (or variable) and call it, it is searched in the current environment and is recursively searched in the parent environments until the symbol is found. A function has a name. This is stored in the R environment. The body of the function contains the statements of a function.

A function can return value and can optionally accept a number of arguments.

To create a function, we need the following syntax:

```
name_of_function <- function(argument1...argumentN) {Body of the function}
```

For example, we can create a function that takes in two integers and returns a sum:

```
my_function <- function(x, y) {
z <- x + y
return(z)
}
```

To call the function, we can pass in the arguments:

```
z <- my_function(1,2)print(z)
```

This will print 3.

We can also set default values to an argument so that its value is taken if a value for an argument is not provided:

```
my_function <- function(x, y=2) {
z <- x + y
return(z)
}
output <- my_function(1)
print(output)
```

The default value of y is 2, therefore, we can call the function without passing in a value for y.

```
my_function <- function(x) {
z <- sin(x)^2 + cos(x)^2+x
return(z)
}
output <- my_function(2)
print(output)
```

The key to note is the use of the curly brackets {...}

**Let's look at a complex case whereby we will use a logical operator**

Let's consider that we want to create a function that accepts the following arguments: Mode, x and y.

- If the Mode is True then we want to add x and y.
- If the Mode is False then we want to subtract x and y.

```
my_function <- function(mode, x, y) {
if (mode == TRUE)
{
z <- x + y
```

```
return(z)
}
else
{
z <- x - y
return(z)
}
}
```

To call the function to add the values of x and y, we can do:

```
output <- my_function(TRUE, 1, 5)
print(output)
```

This will print 6

## Control flow and Loops:

These are the basic control-flow constructs of the R language. They function in much the same way as control statements in any Algol-like language. They are all reserved words.

R supports control structures too. Data scientists can add logic to the R code. This section highlights the most important control structures:

**For loop:**

**If the number of repetition is known in advance then a for( ) loop can be used.**

Syntax

For (name in vector) {commands to be executed}

All operations/commands are executed for all these values.

```
for(i in 1:5) print(1:i)
for(n in c(2,5,10,20,50)) {
    x <- stats::rnorm(n)
    cat(n, ": ", sum(x^2), "\n", sep = "")
}
f <- factor(sample(letters[1:5], 10, replace = TRUE))
for(i in unique(f)) print(i)
```

**While loop:**

**If the number of loops are not known in before, for example when an iterative algorithm to maximize a likelihood function is used, one can use a while ( ) loop.**

Syntax

While (condition) {commands to be executed as long as condition is TRUE}

If the condition is not TRUE before entering the loop, no commands within the the loop are executed.

**Help ("while")**

**Example:**

```
i<-1
while(i<5){
print  (i^2)
i<-i+2
}
```

[1] 1

[1] 9

**Note : The programmer itself has to be careful that the counting variable 'i' within the loop is incremented. Otherwise an infinite loop is occurs.**

Occasionally, we have to loop until a condition is met. Once the condition is false then we can exit the loop.

We can use the while loops to achieve the desired features.

In the code below, we are setting the value of x to 3 and the value of z to 0. Subsequently, we are incrementing the value of z by 1 each time until the value of z is equal to or greater than x.

```
x <- 3
z <- 0
while(z < x) {
  z <- z + 1
}
```

**If Else (optional)**

If Then Else are heavily used in programming.

In a nutshell, a condition is evaluated in the if control block. If it is true then its code is executed otherwise the next block is executed. The next block can either by Else If or Else.

if(condition is true) {# execute statements}
We can also have an optional else:
if(condition is true) {# execute statements}
else if (another condition is true) {# execute statements}

else {# execute statement}

Ifelse (Test, Yes, No)

x <- 1:10
ifelse (x<6, x^2, x+2)

**Repeat**

The Repeat loop does not test any condition - in contrast to the while () loop - before entering the loop and also not during the execution of the loop.

Again the programmer is responsible that the loop terminates after the appropriate number of iterations. For this the **break** command can be used.

Syntax

repeat { commands to be executed}

i<-1
repeat{
print(i^2)
i<-i+2
if(i>10){
break
}
}

```
x <- 1
repeat {
print(x)
x = x+1
if (x == 6){
break
```

```
}
}
```

If we want to repeat a set of statements for an unknown number of times (maybe until a condition is met or a user enters a value etc.) then we can the repeat/break statements. A break can end the iteration.

```
repeat {
print("hello")
x <- random()
if (x > 0.5)
{
next #iterate again
}
}
```

If we want to skip an iteration, we can use the next statement:

## Working with Vectors and Matrices:

**What Are Vectors?**

Vector is one of the most important data structures in R. Essentially, a vector is a collection of elements where each element is required to have the same data type e.g. logical (TRUE/FALSE), Numeric, character.

We can also create an empty vector:

```
x <- vector()
```

By default, the type of vector is logical, such as True/False. The line below will print logical as the type of vector:

```
Type of(x)
```

To create a vector with your elements, you can use the concatenate function (c):

```
x <- c("Pranav", "Balaji", "Naimisha")
print(x)
```

This will print:

[1] “Pranav”
[2] “Balaji”
[3] “Naimisha”

If we want to find the length of a vector, we can use the length() function:

```
length(x)
```

This will print 3 as there are three elements in the vector.

To add elements into a vector, we can combine an element with a vector.

For example, to add “world” at the start of a vector with one element “hello”:

```
x <- c("hello")
x <- c("world", x)
print(x)
```

This will print “world” “hello”

```
a <- "Hello"

b <- 'How'

c <- "are you? "

print(paste(a,b,c))

print(paste(a,b,c, sep = "-"))

print(paste(a,b,c, sep = "", collapse = ""))
```

**sep** represents any separator between the arguments. It is optional.

**collapse** is used to eliminate the space in between two strings. But not the space

within two words of one string.

If we mix the types of elements then R will accommodate the type of the vector too. The type (mode) of the vector will become whatever it considers being the most suitable for the vector:

```
another_vec <- c("test", TRUE)
print(typeof(another_vec))
```

```
another_vec <- c("100", TRUE)
print(typeof(another_vec))
```

Although the second element is a logical value, the type will be printed as “character”.

Operations can also be performed on vectors.

As an instance, to multiply a scalar to a vector:

```
x <- c(1,2,3)
y <- x*2
print(y)
```

This will print 2,4,6

We can also add two vectors together:

```
x <- c(1,2,3)
y <- c(4,5,6)
z <- x+y
print(z)
```

This will print 5 7 9

f the vectors are characters and we want to add them together:

```
x <- c("F","A","C")
y <- c("G","E","D")
z <- x+y
print(z)
```

It will output:

*Error in x + y : non-numeric argument to binary operator*.

**Counting number of characters in a string -ncahr() function**

This function counts the number of characters including spaces in a string.

**Syntax**

The basic syntax for nchar() function is :

```
nchar(x)
```

```
result <- nchar("Count the number of characters")
```

```
print(result)
```

## Matrices:

Matrix is also one of the most common data structures in R. Matrices are important objects in any calculation.

It can be considered as an extension of a vector. A matrix can have multiple rows and columns. All of the elements of a matrix must have the same data type.

To create a matrix, use the matrix() constructor and pass in **nrow** and **ncol** to indicate the columns and rows:

```
x <- matrix(nrow=4, ncol=4)
```

```
x <- matrix(nrow=4, ncol=4, data = c(1,2,3,4,5,6,7,8))
```

This will create a matrix variable, named x, with 4 rows and 4 columns.

A vector can be transformed into a matrix by passing a matrix in the constructor. The resultant matrix will be filled column-wise:

```
vector <- c(1, 2, 3)
x <- matrix(vector)
print(x)
```

This will create a matrix with 1 column and 3 rows (one for each element):

```
[,1]
[1,] 1
```

[2,] 2

[3,] 3

If we want to fill a matrix by row or column then we can explicitly pass in the number of rows and columns along with the *byrow* parameter:

```
vector <- c(1, 2, 3, 4)
x <- matrix(vector, nrow=2, ncol=2, byrow=TRUE)
print(x)
```

The above code created a matrix with 2 columns and rows. The matrix is filled by row.

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

```
vector <- c(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
x <- matrix(vector, nrow=4, ncol=4, byrow=TRUE)
print(x)
```

Construction of a Diagonal Matrix:
Here identity of a diagonal matrix is : 2

```
d<- diag(1, nrow=2, ncol=2)
```

Transpose Matrix:

```
x <- matrix(nrow=4, ncol=2, data=1:8, byrow=T)
```

```
xt<- t(x)
xt
```

Multiplication of a matrix:

```
4*x
t(x)%*%x
```

## Reading in Data:

R offers a range of packages that can allow us to read/write to external data such as excel files or SQL tables. This section illustrates how we can achieve it.

http://home.iitk.ac.in/~shalab/Rcourse/munichdata.asc

head(state.x77)

setwd("F:/GITAM/2019 ODD SEM SESSIONAL MARKS")

read.csv("csbs.csv")

mydata<-read.csv("csbs.csv")

**datamunich<-**

**read.table(file="http://home.iitk.ac.in/~shalab/Rcourse/munichdata.asc", header=TRUE)**

require(XLConnect)

tmp = tempfile(fileext = ".xls")

download.file(url = "https://www.misoenergy.org/Library/Repository/Market%20Reports/20140610_sr_nd_is.xls", destfile = tmp, method = "curl")

readWorksheetFromFile(file = tmp, sheet = "Sheet1", header = TRUE, startRow = 11, startCol = 2, endCol = 13)

**Read an Excel file**

## Writing Data:

**Write To an Excel File**

columnA <- runif(100,0.1,100)

columnB <- runif(100,5,1000)

df <- data.frame(columnA, columnB)

write.xlsx(df, file = path, sheetName="NewSheet", append=TRUE)

It created a new excel file with sheet named as NewSheet:

library(MASS)

attach(bacteria)

fix(bacteria)

## Manipulating Data:

Now we will use **dplyr** to manipulate the data, using the basic operations we discussed in week 1:

**Sort:** Largest to smallest, oldest to newest, alphabetical etc.

**Filter:** Select a defined subset of the data.

**Summarize/Aggregate:** Deriving one value from a series of other values to produce a summary statistic. Examples include: count, sum, mean, median, maximum, minimum etc. Often you'll **group** data into categories first, and then aggregate by group.

**Join:** Merging entries from two or more datasets based on common field(s), e.g. unique ID number, last name and first name.

Here are some of the most useful functions in **dplyr**:

- select Choose which columns to include.
- filter **Filter** the data.
- arrange **Sort** the data, by size for continuous variables, by date, or alphabetically.
- group_by **Group** the data by a categorical variable.
- summarize **Summarize**, or aggregate (for each group if following group_by). Often used in conjunction with functions including:
  - mean Calculate the mean, or average.
  - median Calculate the median.
  - max Find the maximum value.
  - min Find the minimum value
  - sum Add all the values together.
  - n Count the number of records.
- mutate Create new column(s) in the data, or change existing column(s).
- rename Rename column(s).
- bind_rows Merge two data frames into one, combining data from columns with the same name.

Now we will **filter** and **sort** the data in specific ways. For each of the following examples, copy the code that follows into your script, and view the results. Notice how we create a new objects to hold the processed data.

**Find doctors in California paid $10,000 or more by Pfizer to run "Expert-Led Forums."**

```
# doctors in California who were paid $10,000 or more by Pfizer to run "Expert-Led Forums."
ca_expert_10000 <- pfizer %>%
  filter(state == "CA" & total >= 10000 & category == "Expert-Led Forums")
```

Notice the use of == to find values that match the specified text, >= for greater than or equal to, and the Boolean operator &.

Now add a **sort** to the end of the code to list the doctors in descending order by the payments received:

```
# doctors in California who were paid $10,000 or more by Pfizer to run "Expert-Led Forums."
ca_expert_10000 <- pfizer %>%
  filter(state == "CA" & total >= 10000 & category == "Expert-Led Forums") %>%
  arrange(desc(total))
```

## Simulation:

In a **simulation**, you set the ground rules of a random process and then the computer uses random numbers to generate an outcome that adheres to those rules. As a simple example, you can simulate flipping a fair coin with the following commands.

```
outcomes <- c("heads", "tails")
sample(outcomes, size = 1, replace = TRUE)
```

The vector outcomes can be thought of as a hat with two slips of paper in it: one slip says "heads" and the other says "tails". The function sample() draws one slip from the hat and tells us if it was a head or a tail.

## Data Frame:

Most, if not all of the data science projects require input data in tabular format. Data frames data structure can be used to represent tabular data in R. Each column can contain a list of elements and each column can be of a different type than each other.

To create a data frame of 2 columns and 5 rows each, simply do:

```
my_data_frame <- data.frame(column_1 = 1:5, column_2 = c("A", "B", "C", "D", E"))
print(my_data_frame)

library(MASS, lib.loc = "C:/Program Files/R/R-3.6.3/library")
Painters
rownames(painters)

colnames(painters)

summary(painters)

is.factor(painters$School)

is.factor(painters$Drawing)

is.data.frame(painters)
```

## Graphics in R:

Everyone comes across some form of graphical representation in both professional and personal endeavors. We see graphical representations of data in newspaper, professional and research journals. We need to have a basic knowledge of graphical representations on one or both of the following two counts. First, we should be able to understand graphical representations. Second, if need be, we should be able to make graphical representations ourselves. We shall demonstrate how to make different kinds of graphical representation of data in this section.

R offers a remarkable variety of graphics. To get an idea, one can type demo(graphics) or demo(persp). It is not possible to detail here the possibilities of R in terms of graphics, particularly since each graphical function has a large number of options making the production of graphics very flexible.

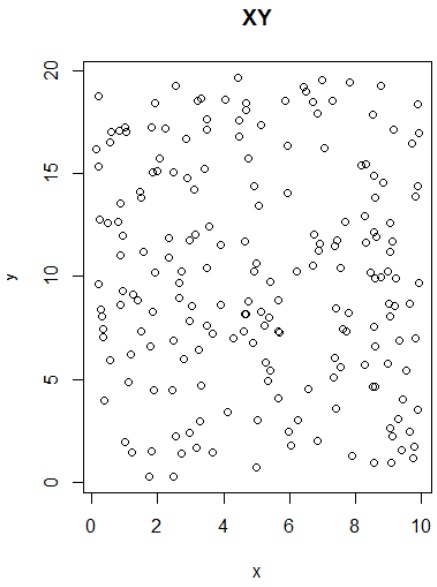
**X-Y Scatter Plot**

I have generated the following data:

```
x<-runif(200, 0.1, 10.0)
y<-runif(200, 0.15, 20.0)print(x)
print(y)
```

The code snippet will plot the chart

```
plot(x, y, main='Line Chart')
```




XY Scatter Plot

**Histogram:** The histogram plots the numeric vector along the x-axis (with equal width boxes for the histogram) and height of the histogram is the frequency of values within each of the value intervals used for plotting the histogram.

Example 1:

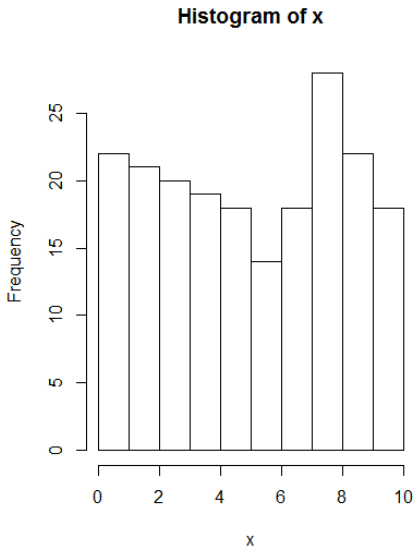
Histogram of values 1, 2, 1, 1, 2, 3, 1, 2, 3, 1, 2, 2, 3

```
x=c(1, 2, 1, 1, 2, 3, 1, 2, 3, 1, 2, 2, 3)

hist(x)
```

Example 2:

```
x<-runif(200, 0.1, 10.0)
hist(x)
```




**Pie Chart**: Different pie charts have been drawn for demonstration. The reader may understand and

configure the various pie chart options used. They are simple to understand!

```
school<-{col=c(2,3,4,5,6)}
pie(school)
pie(table(school))
```

The above three pie charts have been configured in a single diagram with layout matrix option and the relevant R-code has been given below. The following R-code will gives all the three pie charts in one graph. The layout matrix option of graphical plotting gives a higher flexibility on plotting and could include different sized graphical diagrams on the same plot.

```
layout(matrix(c(1,2,3,3), nrow=2, ncol=2, byrow=T))

pie(school, col=c(2,3,4,5,6))

pie(school, col=c(2,3,4,5,6), label=c("pre
primary(59)","primary(202)","middle(61)","high

school(38)","higher secondary(35)"))

pie(school, density=10, angle=15+10*1:5, label=c("pre

primary(59)","primary(202)","middle(61)","high(38)","higher secondary(35)"),
main="Number of schools, Perambalur District")
```