1. Introduction to R and R studio

- R is an open-source programming language that is widely used as a statistical software and data analysis tool. R is an important tool for Data Science.
- R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.
- The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.
- R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.
- R is free software distributed under a GNU-style copy left, and an official part of the GNU project called GNU S.

Features of R – Data Science

- R provides extensive support for statistical modelling.
- R is a suitable tool for various data science applications because it provides aesthetic visualization tools.
- R is heavily utilized in data science applications for ETL (Extract, Transform, Load). It provides an interface for many databases like SQL and even spreadsheets.
- R also provides various important packages for data wrangling.
- With R, data scientists can apply machine learning algorithms to gain insights about future events.
- One of the important feature of R is to interface with NoSQL databases and analyze unstructured data.

R Libraries

Dplyr: For performing data wrangling and data analysis, we use the dplyr package. We use this package for facilitating various functions for the Data frame in R. Dplyr is actually built around

these 5 functions. You can work with local data frames as well as with remote database tables. You might need to:

Select certain columns of data.

Filter your data to select specific rows.

Arrange the rows of your data into order.

Mutate your data frame to contain new columns.

Summarize chunks of your data in some way.

Ggplot2: R is most famous for its visualization library ggplot2. It provides an aesthetic set of graphics that are also interactive. The ggplot2 library implements a “grammar of graphics”.

This approach gives us a coherent way to produce visualizations by expressing relationships between the attributes of data and their graphical representation.

Esquisse: This package has brought the most important feature of Tableau to R. Just drag and drop, and get your visualization done in minutes. This is actually an enhancement to ggplot2.It allows us to draw bar graphs, curves, scatter plots, histograms, then export the graph or retrieve the code generating the graph.

Tidyr: Tidyr is a package that we use for tidying or cleaning the data. We consider this data to be tidy when each variable represents a column and each row represents an observation.

Shiny: This is a very well known package in R. Shiny is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or embed them in R Markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions. It’s a Data Scientist’s best friend.

Caret: Caret stands for classification and regression training. Using this function, you can model complex regression and classification problems.

E1071: This package has wide use for implementing clustering, Fourier Transform, Naive Bayes, SVM and other types of miscellaneous functions.

Mlr: This package is absolutely incredible in performing machine learning tasks. It almost has all the important and useful algorithms for performing machine learning tasks. It can also be termed as the extensible framework for classification, regression, clustering, multi-classification and survival analysis.

Applications of R for Data Science

Top Companies that use R for Data Science:

Google: At Google, R is a popular choice for performing many analytical operations. The Google Flu Trends project makes use of R to analyze trends and patterns in searches associated with flu.

Facebook: Facebook makes heavy use of R for social network analytics. It uses R for gaining insights about the behavior of the users and establishes relationships between them.

IBM: IBM is one of the major investors in R. It recently joined the R consortium. IBM also utilizes R for developing various analytical solutions. It has used R in IBM Watson – an open computing platform.

Uber: Uber makes use of the R package shiny for accessing its charting components. Shiny is an interactive web application that's built with R for embedding interactive visual graphics.

Installation of R and R studio

Windows Installation

You can download the Windows installer version of R from R-3.2.2 for Windows (32/64 bit) and save it in a local directory.

As it is a Windows installer (.exe) with a name "R-version-win.exe". You can just double click and run the installer accepting the default settings. If your Windows is 32-bit version, it installs the 32-bit version. But if your windows is 64-bit, then it installs both the 32-bit and 64-bit versions.

After installation you can locate the icon to run the Program in a directory structure "R\R-3.2.2\bin\i386\Rgui.exe" under the Windows Program Files. Clicking this icon brings up the R-GUI which is the R console to do R Programming.

First install R

https://cran.r-project.org/

R studio

https://posit.co/download/rstudio-desktop/

Basics of R

R Command prompt

Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt:

```
$ R
```

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows:

```
> myString <- "Hello, World!"
> print ( myString)
[1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

RScriptFile

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called Rscript. So let's start with writing following code in a text file called test.R as under:

```
# My first program in R Programming
myString <- "Hello, World!"


print ( myString)
```

Save the above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

```
$ Rscript test.R
```

When we run the above program, it produces the following result.

```
[1] "Hello, World!"
```

Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows:

```
# My first program in R Programming
```

R does not support multi-line comments

R- Data Types

➢ Generally, while doing programming in any programming language, you need to use various variables to store various information.

- Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.
- You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.
- In contrast to other programming languages like C and java in R, the variables are not declared as some data type.
- The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are:
    - ➢ Vectors
    - ➢ Lists
    - ➢ Matrices
    - ➢ Arrays
    - ➢ Factors
    - ➢ Data Frames

The simplest of these objects is the vector object and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type	Example	Verify

Logical	TRUE , FALSE	`v <- TRUE`
		`print(class(v))`
		it produces the following result:
		`[1] "logical"`

Numeric	12.3, 5, 999	`v <- 23.5`
		`print(class(v))`
		it produces the following result:

Integer	2L, 34L, 0L	`v <- 2L`
		`print(class(v))`
		it produces the following result:
		`[1] "integer"`

Complex	3 + 2i	`v <- 2+5i`
		`print(class(v))`
		it produces the following result:
		`[1] "complex"`

Character	'a' , '"good", "TRUE", '23.4'	`v <- "TRUE"`
		`print(class(v))`
		it produces the following result:
		`[1] "character"`

Raw	"Hello" is stored as 48 65 6c 6c 6f	`v <- charToRaw("Hello")`
		`print(class(v))`
		it produces the following result:
		`[1] "raw"`


```
> x = 11
>
> print(x)
[1] 11
>
> x
[1] 11
>
> X
Error: object 'X' not found
>
> y <- 7
> y
[1] 7
>
> y <- 9
> y
[1] 9
>
> ls()
[1] "x" "y"
>
> rm(y)
>
> y
Error: object 'y' not found
>
> y <- 9
```

```
> x.1 <- 14
> x.1
[1] 14
>
> 1x <- 22
Error: unexpected symbol in "1x"
>
> xx <- "marin"
> xx
[1] "marin"
>
> yy <- "1"
> yy
[1] "1"
>
> 11+14
[1] 25
>
> 7*9
[1] 63
>
> x
[1] 11
> y
[1] 9
>
> x+y
[1] 20
```

```
> z <- x+y
> z
[1] 20
>
> x-y
[1] 2
>
> x*y
[1] 99
>
> x/y
[1] 1.222222
>
> y^2
[1] 81
>
> x^2 + y^2
[1] 202
>
> sqrt(y)
[1] 3
>
> y^(1/2)
[1] 3
>
> log(y)
[1] 2.197225
```

```
> exp(y)
[1] 8103.084
>
> log2(y)
[1] 3.169925
>
> abs(-14)
[1] 14
>
> sqrt(y
+
+ )
[1] 3
>
> the code below is for...
Error: unexpected symbol in "the code"
>
```

R-Variables

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R-objects.

A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	Valid	Has letters, numbers, dot and underscore
var_name%		

Invalid	Has the character '%'. Only dot(.) and underscore allowed.	
2var_name	Invalid	Starts with a number
.var_name ,var.name	valid	Can start with a dot(.) but the dot(.)should not befollowed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid
_var_name	invalid	Starts with _ which is not valid

Variable Assignment

- The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using print() or cat()function. The cat() function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)

# Assignment using leftward operator.
var.2 <- c("learn","R")
```

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

We have the following types of operators in R programming:

☐ Arithmetic Operators

☐ Relational Operators

☐ Logical Operators

☐ Assignment Operators

☐ Miscellaneous Operators

Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

Operator	Description	Example

+	Adds two vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v+t)`
		it produces the following result:
		`[1] 10.0 8.5 10.0`

−	Subtracts second vector from the first	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v-t)`
		it produces the following result:
		`[1] -6.0 2.5 2.0`

*	Multiplies both vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v*t)`
		it produces the following result:
		`[1] 16.0 16.5 24.0`

/	Divide the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v/t)`
		When we execute the above code, it produces the following result:
		`[1] 0.250000 1.833333 1.500000`

%%	Give the remainder of the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%%t)`
		it produces the following result:
		`[1] 2.0 2.5 2.0`

%/%	The result of division of first vector with second (quotient)	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%/%t)`
		it produces the following result:
		[1] 0 1 1

^	The first vector raised to the exponent of second vector	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v^t)`
		it produces the following result:
		`[1] 256.000 166.375 1296.000`

Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>t)`
		it produces the following result:
		`[1] FALSE  TRUE FALSE FALSE`

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v < t)`
		it produces the following result:
		`[1]  TRUE FALSE  TRUE FALSE`

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v==t)`
		it produces the following result:
		`[1] FALSE FALSE FALSE  TRUE`

<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v<=t)`
		it produces the following result:
		`[1]  TRUE FALSE  TRUE  TRUE`

>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>=t)`
		it produces the following result:
		`[1] FALSE  TRUE FALSE  TRUE`

Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	`v <- c(3,1,TRUE,2+3i)`
		`t <- c(4,1,FALSE,2+3i)`
		`print(v&t)`
		it produces the following result:
		`[1]  TRUE  TRUE FALSE  TRUE`

|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(4,0,FALSE,2+3i)`
		`print(v|t)`
		it produces the following result:
		`[1]  TRUE FALSE  TRUE  TRUE`

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	`v <- c(3,0,TRUE,2+2i)`
		`print(!v)`
		it produces the following result:
		`[1] FALSE  TRUE FALSE FALSE`

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(1,3,TRUE,2+3i)`
		`print(v&&t)`
		it produces the following result:
		`[1] TRUE`

||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(0,0,TRUE,2+2i)`
		`t <- c(0,3,TRUE,2+3i)`
		`print(v||t)`
		it produces the following result:
		`[1] FALSE`

Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example

<-
or
=
or
<<-
	Called Left Assignment	`v1 <- c(3,1,TRUE,2+3i)`
		`v2 <<- c(3,1,TRUE,2+3i)`
		`v3 = c(3,1,TRUE,2+3i)`
		`print(v1)`
		`print(v2)`
		`print(v3)`
		`it produces the following result:`
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

-> or ->>	Called Right Assignment	c(3,1,TRUE,2+3i) -> v1 c(3,1,TRUE,2+3i) ->> v2 print(v1) print(v2) it produces the following result: [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logic computation.

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	v <- 2:8 print(v) it produces the following result: [1] 2 3 4 5 6 7 8
%in%	This operator is used to identify if an element belongs to a vector.	v1 <- 8 v2 <- 12 t <- 1:10 print(v1 %in% t) print(v2 %in% t) it produces the following result: [1] TRUE [1] FALSE

%*%	This operator is used to multiply a matrix with its transpose.	`M = matrix( c(2,6,5,1,10,4), nrow=2,ncol=3,byrow = TRUE)`
		`t = M %*% t(M)`
		`print(t)`
		it produces the following result:
		`[,1] [,2]`
		`[1,] 65 82`
		`[2,] 82 117`

2. Create and Work with Vectors

Vectors

The key feature which makes R very useful for statistics is that it is vectorized. This means that many operations can be performed point-wise on a

vector. The function c() is used to create vectors:

6

> x <- c(1, -1, 3.5, 2)

> x

[1] 1.0 -1.0 3.5 2.0

Then if we want to add 2 to everything in this vector, or to square each

entry:

> x + 2

[1] 3.0 1.0 5.5 4.0

> x^2

[1] 1.00 1.00 12.25 4.00

This is very useful in statistics:

> sum((x - mean(x))^2)

[1] 10.69

```
> x1 <- c(1,3,5,7,9)
> x1
[1] 1 3 5 7 9
>
> gender <- c("male", "female")
> gender
[1] "male"   "female"
>
> 2:7
[1] 2 3 4 5 6 7
>
> seq(from=1, to=7, by=1)
[1] 1 2 3 4 5 6 7
>
> seq(from=1, to=7, by=1/3)
 [1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667 3.000000 3.333333
 [9] 3.666667 4.000000 4.333333 4.666667 5.000000 5.333333 5.666667 6.000000
[17] 6.333333 6.666667 7.000000
>
> seq(from=1, to=7, by=0.25)
 [1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25
[15] 4.50 4.75 5.00 5.25 5.50 5.75 6.00 6.25 6.50 6.75 7.00
>
> rep(1, times=10)
 [1] 1 1 1 1 1 1 1 1 1 1
>
> rep("marin", times=5)
[1] "marin" "marin" "marin" "marin" "marin"
>
```

```
Console ~/
> rep(1:3, times=5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
>
> rep(seq(from=2, to=5, by=0.25), times=5)
 [1] 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00
[15] 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25
[29] 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25 2.50
[43] 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25 2.50 2.75
[57] 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00
>
> rep(c("m", "f"), times=5)
 [1] "m" "f" "m" "f" "m" "f" "m" "f" "m" "f"
>
> x <- 1:5
> x
[1] 1 2 3 4 5
>
> y <- c(1,3,5,7,9)
> y
[1] 1 3 5 7 9
>
> x + 10
[1] 11 12 13 14 15
>
> x - 10
[1] -9 -8 -7 -6 -5
>
> x*10
[1] 10 20 30 40 50
>
```

```
Console ~/
> x/2
[1] 0.5 1.0 1.5 2.0 2.5
>
> # if two vectors are of the same length, we may add/subtract/mult/div
> # corresponding elements
>
> x
[1] 1 2 3 4 5
>  y
[1] 1 3 5 7 9
>
> x+y
[1]  2  5  8 11 14
>
> x-y
[1]  0 -1 -2 -3 -4
>
> x*y
[1]  1  6 15 28 45
>
> x/y
[1] 1.0000000 0.6666667 0.6000000 0.5714286 0.5555556
>
> x
[1] 1 2 3 4 5
>
> y
[1] 1 3 5 7 9
>
```

```
Console ~/
> y[3]
[1] 5
>
> y[-3]
[1] 1 3 7 9
>
> y[1:3]
[1] 1 3 5
>
> y[c(1, 5)]
[1] 1 9
>
> y[-c(1, 5)]
[1] 3 5 7
>
> y[y<6]
[1] 1 3 5
>
> matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
>
> matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
```

```
>
> mat <- matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=TRUE)
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
>
> mat[1, 2]
[1] 2
>
> mat[c(1, 3), 2]
[1] 2 8
>
> mat[2,]
[1] 4 5 6
>
> mat[,1]
[1] 1 4 7
>
> mat*10
     [,1] [,2] [,3]
[1,]   10   20   30
[2,]   40   50   60
[3,]   70   80   90
>
```

Exercise 2.1. The weights of five people before and after a diet are given in the table.

Before 78 72 78 79 105

After 67 65 79 70 93

Read the 'before' and 'after' values into two different vectors called before and after.

Use R to evaluate the amount of weight lost for each participant. What is the average amount of weight lost?

We can also use R's vectorization to create more interesting sequences:

> 2^(0:10)

[1] 1 2 4 8 16 32 64 128 256 512 1024

> 1:3 + rep(seq(from=0,by=10,to=30), each=3)

[1] 1 2 3 11 12 13 21 22 23 31 32 33

The last example demonstrates recycling, which is also an important part of vectorization. If we perform a binary operation (such as +) on two vectors of different lengths, the shorter one is used over and over again until the operation has been applied to every entry in the longer one. If the longer length is not a multiple of the shorter length, a warning is given.

> 1:10 * c(-1,1)

[1] -1 2 -3 4 -5 6 -7 8 -9 10

> 1:7 * 1:2

Warning: longer object length is not a multiple of shorter object length

[1] 1 4 3 8 5 12 7

Exercise 2.2. Create the following vectors in R using seq() and rep().

(i) 1, 1.5, 2, 2.5, . . . , 12

(ii) 1, 8, 27, 64, . . . , 1000.

(iii) 1, 0, 3, 0, 5, 0, 7, . . . , 0, 49.

Subsetting

It's frequently necessary to extract some of the elements of a larger vector. In R you can use square brackets to select an individual element or group of elements:

> x <- c(5,9,2,14,-4)

> x[3]

[1] 2

> # note indexing starts from 1

> x[c(2,3,5)]

[1] 9 2 -4

> x[1:3]

[1] 5 9 2

> x[3:length(x)]

[1] 2 14 -4

There are two other methods for getting subvectors. The first is using a logical vector (i.e. containing TRUE and FALSE) of the same length:

```
> x > 4

[1] TRUE TRUE FALSE TRUE FALSE

> x[x > 4]

[1] 5 9 14
```

or using negative indices to specify which elements should not be selected:

```
> x[-1]

[1] 9 2 14 -4

> x[-c(1,4)]

[1] 9 2 -4
```

(Note that this is rather different to what other languages such as C or Python would interpret negative indices to mean.)

Exercise 2.3. The built-in vector LETTERS contains the uppercase letters of the alphabet. Produce a vector of (i) the first 12 letters; (ii) the odd 'numbered' letters; (iii) the (English) consonants.

Character Vectors

As you might have noticed in the exercise above, vectors don't have to contain numbers. We can equally create a character vector, in which each entry is a string of text. Strings in R are contained within double quotes ":

```
> x <- c("Hello", "how do you do", "lovely to meet you", 42)

> x

[1] "Hello" "how do you do" "lovely to meet you"

[4] "42"
```

Notice that you cannot mix numbers with strings: if you try to do so the number will be converted into a string. Otherwise character vectors are much like their numerical counterparts.

```
> x[2:3]

[1] "how do you do" "lovely to meet you"

> x[-4]
```

[1] "Hello" "how do you do" "lovely to meet you"

> c(x[1:2], "goodbye")

[1] "Hello" "how do you do" "goodbye"

Matrices

Matrices are much used in statistics, and so play an important role in R. To create a matrix use the function matrix(), specifying elements by column

first:

> matrix(1:12, nrow=3, ncol=4)

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

[1,] 1 4 7 10

[2,] 2 5 8 11

[3,] 3 6 9 12

This is called column-major order. Of course, we need only give one of the dimensions:

> matrix(1:12, nrow=3)

unless we want vector recycling to help us:

> matrix(1:3, nrow=3, ncol=4)

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

[1,] 1 1 1 1

[2,] 2 2 2 2

[3,] 3 3 3 3

Sometimes it’s useful to specify the elements by row first

> matrix(1:12, nrow=3, byrow=TRUE)

There are special functions for constructing certain matrices:

> diag(3)

[,1] [,2] [,3]

[1,] 1 0 0

[2,] 0 1 0

[3,] 0 0 1

```
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
> 1:5 %o% 1:5
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

The last operator performs an outer product, so it creates a matrix with (i, j)-th entry xiyj . The function outer() generalizes this to any function f on two arguments, to create a matrix with entries f(xi , yj).

```
> outer(1:3, 1:4, "+")
     [,1] [,2] [,3] [,4]
[1,]    2    3    4    5
[2,]    3    4    5    6
[3,]    4    5    6    7
```

Matrix multiplication is performed using the operator %*%, which is quite distinct from scalar multiplication *.

```
> A <- matrix(c(1:8,10), 3, 3)
> x <- c(1,2,3)
> A %*% x # matrix multiplication
     [,1]
[1,]   30
[2,]   36
[3,]   45
```

> A*x # NOT matrix multiplication

[,1] [,2] [,3]

[1,] 1 4 7

[2,] 4 10 16

[3,] 9 18 30

Standard functions exist for common mathematical operations on matrices.

> t(A) # transpose

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

[3,] 7 8 10

> det(A) # determinant

[1] -3

> diag(A) # diagonal

[1] 1 5 10

> solve(A) # inverse

[,1] [,2] [,3]

[1,] -0.6667 -0.6667 1

[2,] -1.3333 3.6667 -2

[3,] 1.0000 -2.0000 1

Exercise 2.4. Construct the matrix

B= [1 2 3

4 2 6

-3 1 -3]

Subsetting of Matrices

Matrices can be subsetted much the same way as vectors, although of course they have two indices. Row number comes first:

> A[2,1]

[1] 2

```
> A[2,2:ncol(A)]
[1] 5 8
> A[,1:2] # blank indices give everything
     [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3   6
> A[c(),1:2] # empty indices give nothing!
[,1] [,2]
```

Notice that, where appropriate, R automatically reduces a matrix to a vector or scalar when you subset it. You can override this using the optional drop argument.

```
> A[2,2:ncol(A),drop=FALSE] # returns a matrix
     [,1] [,2]
[1,]  5     8
```

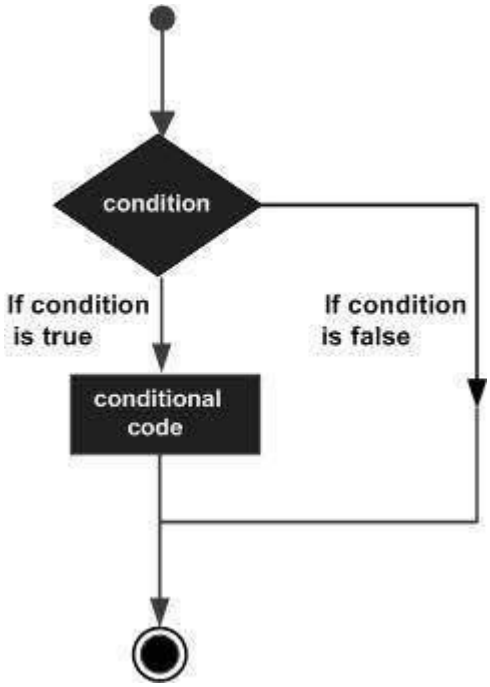
You can stitch matrices together using the rbind() and cbind() functions. These employ vector recycling:

```
> cbind(A, t(A))
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,] 1 4 7 1 2 3
[2,] 2 5 8 4 5 6
[3,] 3 6 10 7 8 10
> rbind(A, 1, 0)
     [,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 10
[4,] 1 1 1
[5,] 0 0 0
```

3. R – Decision making

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

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



R provides the following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An **if** statement consists of a Boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

switch statement	A switch statement allows a variable to be tested for equality against a list of values.

R - If Statement

An **if** statement consists of a Boolean expression followed by one or more statements.

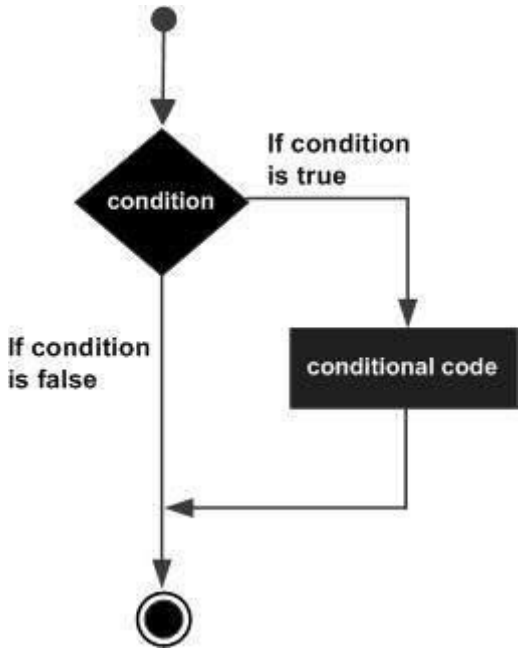
Syntax

The basic syntax for creating an **if** statement in R is:

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
}
```

If the Boolean expression evaluates to be **true**, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
x <- 30L
```

```
if(is.integer(x)){
   print("X is an Integer")
}
```

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

```
[1] "X is an Integer"
```

R – If...Else Statement

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

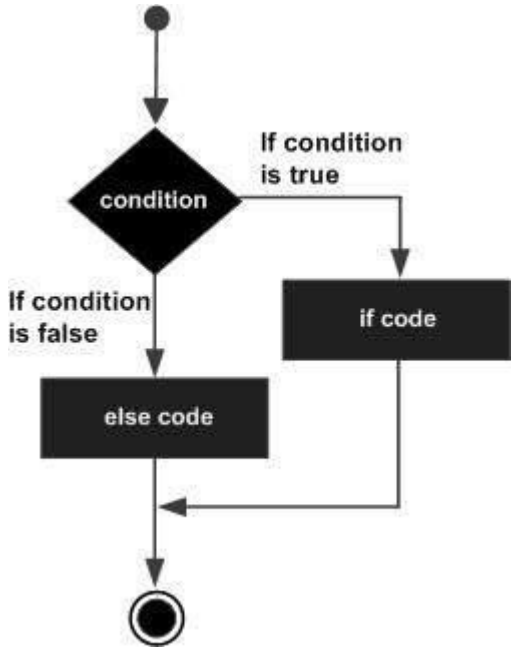
Syntax

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.
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



Example

```
x <- c("what","is","truth")
if("Truth" %in% x){
   print("Truth is found")
} else {
   print("Truth is not found")
}
```

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

```
[1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using **if**, **else if**, **else** statements there are few points to keep in mind.

- An **if** can have zero or one **else** and it must come after any **else if**'s.
- An **if** can have zero to many **else if**'s and they must come before the else.

- Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

The basic syntax for creating an **if...else if...else** statement in R is:

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true.
}else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true.
}else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true.
}else {
   // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")
if("Truth" %in% x){
   print("Truth is found the first time")
} else if ("truth" %in% x) {
   print("truth is found the second time")
} else {
   print("No truth found")
}
```

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

```
[1] "truth is found the second time"
```

R – Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

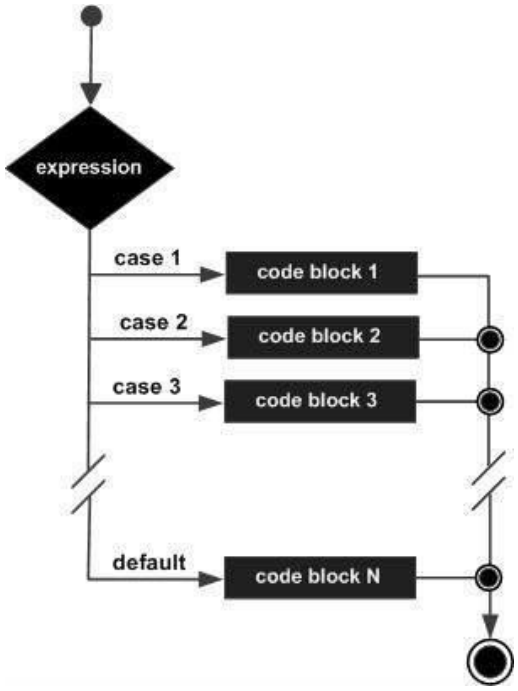
The basic syntax for creating a switch statement in R is :

```
switch(expression, case1, case2, case3 ...)
```

The following rules apply to a switch statement:

- If the value of expression is not a character string it is coerced to integer.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- If the value of the integer is between 1 and nargs()-1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.
- If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- No Default argument is available.
- In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

Flow Diagram



Example

```
x <- switch(
  3,
  "first",
  "second",
  "third",
  "fourth"
)
print(x)
```

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

```
[1] "third"
```

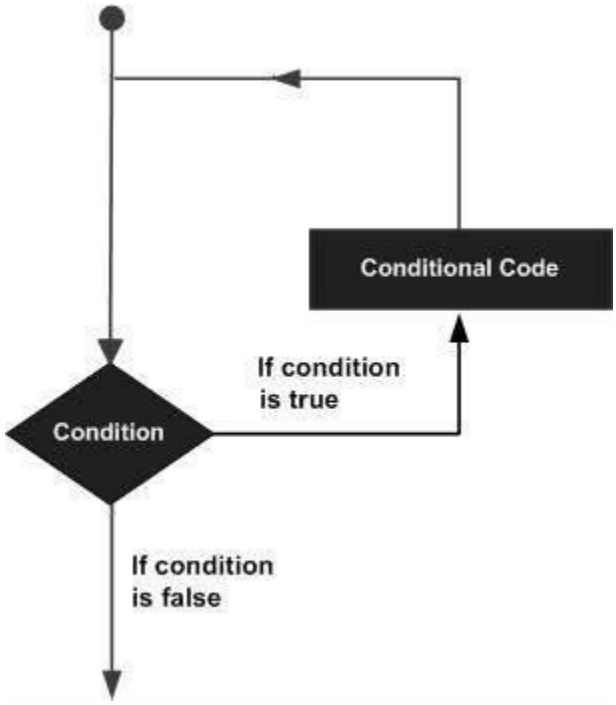
```
Following is val1 simple R program
# to demonstrate syntax of switch.

# Mathematical calculation

val1 = 6
val2 = 7
val3 = "s"
result = switch(
    val3,
    "a"= cat("Addition =", val1 + val2),
    "d"= cat("Subtraction =", val1 - val2),
    "r"= cat("Division = ", val1 / val2),
    "s"= cat("Multiplication =", val1 * val2),
    "m"= cat("Modulus =", val1 %% val2),
    "p"= cat("Power =", val1 ^ val2)
)

print(result)
```

R – Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages:



R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
repeat loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

for loop	Like a while statement, except that it tests the condition at the end of the loop body.

R - Repeat Loop

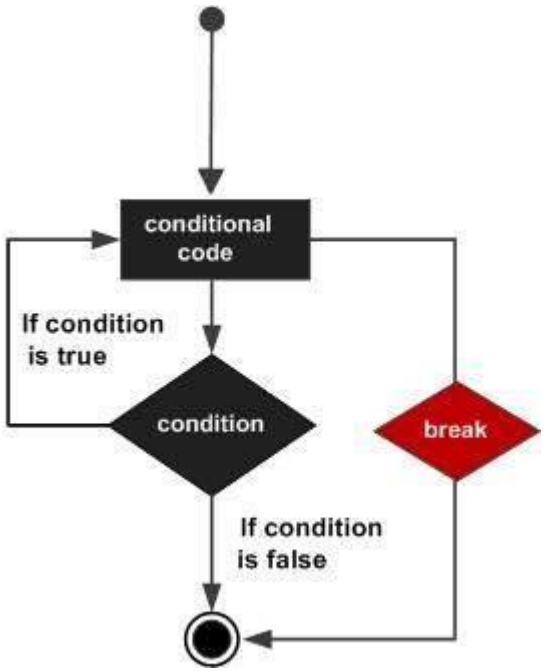
The Repeat loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a repeat loop in R is:

```
repeat {
   commands
   if(condition){
        break
   }
}
```

Flow Diagram



Example

```
v <- c("Hello","loop")
```

```
cnt <- 2
repeat{
   print(v)
   cnt <- cnt+1
   if(cnt > 5){
      break
   }
}
```

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

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

R - While Loop

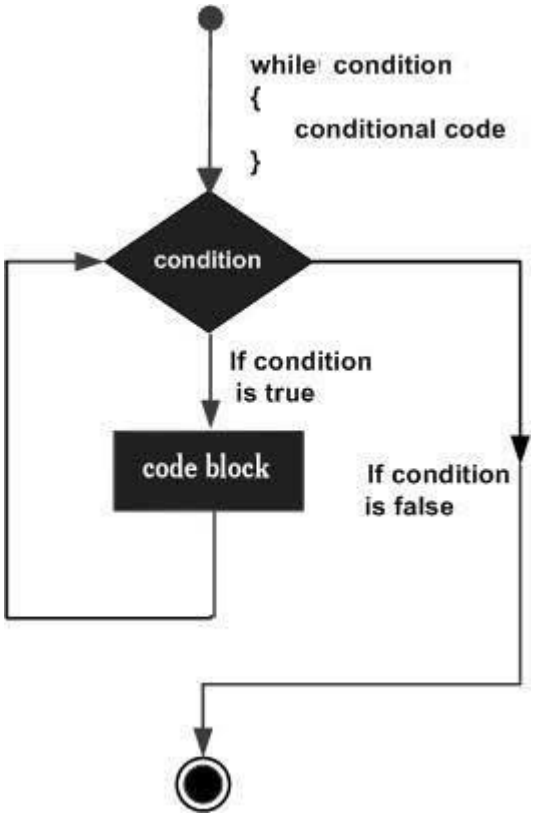
The While loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a while loop in R is :

```
while (test_expression) {
   statement
}
```

Flow Diagram



Here key point of the **while** loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

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 – For Loop

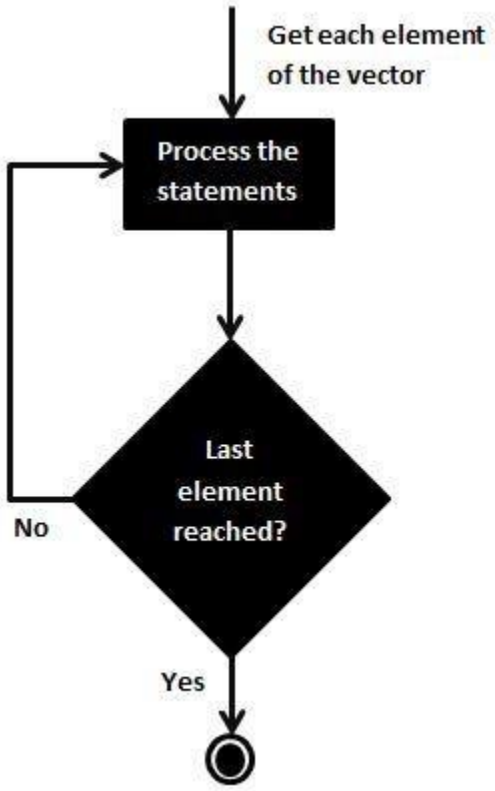
A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The basic syntax for creating a **for** loop statement in R is:

```
for (value in vector) {
    statements
}
```

Flow Diagram



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"
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
Next statement	The next statement simulates the behavior of R switch.

R – Break Statement

The break statement in R programming language has the following two usages:

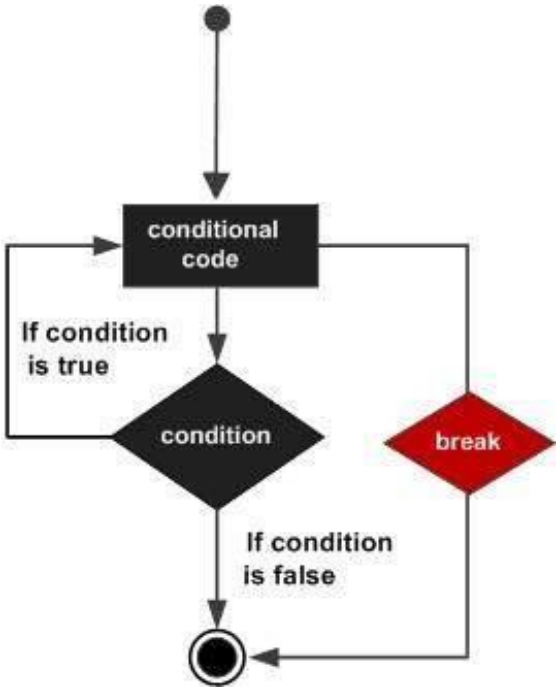
- When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
-
- It can be used to terminate a case in the switch statement (covered in the next chapter).

Syntax

The basic syntax for creating a break statement in R is:

```
break
```

Flow Diagram



Example

```
v <- c("Hello","loop")
cnt <- 2
repeat{
   print(v)
   cnt <- cnt+1
   if(cnt > 5){
      break
   }
}
```

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

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

R – Next Statement

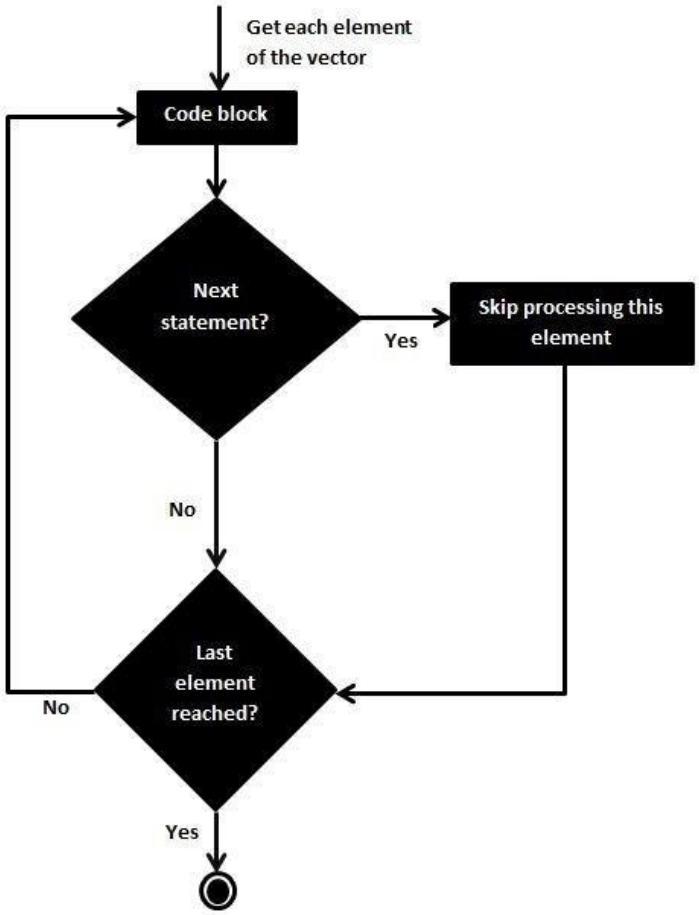
The **next** statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax

The basic syntax for creating a next statement in R is:

```
next
```

Flow Diagram



Example

```
v <- LETTERS[1:6]
for ( i in v){
   if (i == "D"){
      next
   }
   print(i)
}
```

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

```
[1] "A"
[1] "B"
```

R – Function

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows:

```
function_name <- function(arg_1, arg_2, ...) {
    Function body
}
```

Function Components

The different parts of a function are:

- **Function Name:** This is the actual name of the function. It is stored in R environment as an object with this name.
-
- **Arguments:** 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:** The function body contains a collection of statements that defines what the function does.
-
- **Return Value:** The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

Built-in Function

Simple examples of in-built functions are seq(), mean(), max(), sum(x)and paste(...) etc. They are directly called by user written programs. You can refer most widely used R functions.

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```

When we execute the above code, it produces the following result:

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
            b <- i^2
            print(b)
            }
                     }
```

Calling a Function

```
 # Create a function to print squares of numbers in sequence.
```

```
new.function <- function(a) {
  for(i in 1:a) {
            b <- i^2
            print(b)
            }
                     }

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute the above code, it produces the following result:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function <- function() {
            for(i in 1:5) {
            print(i^2)
            }
                     }

# Call the function without supplying an argument.
new.function()
```

When we execute the above code, it produces the following result:

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

```
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function <- function(a,b,c) {
            result <- a*b+c
            print(result)
                        }

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments.
new.function(a=11,b=5,c=3)
```

When we execute the above code, it produces the following result:

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function <- function(a = 3,b =6) {
            result <- a*b
            print(result)
                              }
```

```
# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9,5)
```

When we execute the above code, it produces the following result:

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function <- function(a, b) {
        print(a^2)
        print(a)
        print(b)
                        }

# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute the above code, it produces the following result:

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

R – Strings

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes can not be inserted into a string starting and ending with double quotes.
- Single quote can not be inserted into a string starting and ending with single quote.

Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

```
a <- 'Start and end with single quote'
print(a)

b <- "Start and end with double quotes"
print(b)

c <- "single quote ' in between double quotes"
print(c)

d <- 'Double quotes " in between single quote'
print(d)
```

When the above code is run we get the following output:

```
[1] "Start and end with single quote"
[1] "Start and end with double quotes"
```

```
[1] "single quote ' in between double quote"
[1] "Double quote \" in between single quote"
```

Examples of Invalid Strings

```
e <- 'Mixed quotes"
print(e)

f <- 'Single quote ' inside single quote'
print(f)

g <- "Double quotes " inside double quotes"
print(g)
```

When we run the script it fails giving below results.

```
...: unexpected INCOMPLETE_STRING

.... unexpected symbol
1: f <- 'Single quote ' inside

unexpected symbol
1: g <- "Double quotes " inside
```

String Manipulation

Concatenating Strings - paste() function

Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

Syntax

The basic syntax for paste function is :

```
paste(..., sep = " ", collapse = NULL)
```

Following is the description of the parameters used:

- **...** represents any number of arguments to be combined.
-
- **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.

-

Example

```
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 = ""))
```

When we execute the above code, it produces the following result:

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using **format()**function.

Syntax

The basic syntax for format function is :

```
format(x, digits, nsmall,scientific,width,justify = c("left", "right", "centre",
"none"))
```

Following is the description of the parameters used:

- **x** is the vector input.

-
- **digits** is the total number of digits displayed.
-
- **nsmall** is the minimum number of digits to the right of the decimal point.
-
- **scientific** is set to TRUE to display scientific notation.
-
- **width** indicates the minimum width to be displayed by padding blanks in the beginning.
-
- **justify** is the display of the string to left, right or center.

-

Example

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)

# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)

# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)

# Format treats everything as a string.
result <- format(6)
print(result)

# Numbers are padded with blank in the beginning for width.
result <- format(13.7, width = 6)
print(result)

# Left justify strings.
```

```
result <- format("Hello",width = 8, justify = "l")
print(result)

# Justfy string with center.
result <- format("Hello",width = 8, justify = "c")
print(result)
```

When we execute the above code, it produces the following result:

```
[1] "23.1234568"
[1] "6.000000e+00" "1.314521e+01"
[1] "23.47000"
[1] "6"
[1] "  13.7"
[1] "Hello   "
[1] " Hello  "
```

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)
```

Following is the description of the parameters used:

- **x** is the vector input.

Example

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

When we execute the above code, it produces the following result:

```
[1] 30
```

Changing the case - toupper() & tolower() functions

These functions change the case of characters of a string.

Syntax

The basic syntax for toupper() & tolower() function is :

```
toupper(x)
tolower(x)
```

Following is the description of the parameters used:

- **x** is the vector input.

Example

```
# Changing to Upper case.
result <- toupper("Changing To Upper")
print(result)

# Changing to lower case.
result <- tolower("Changing To Lower")
print(result)
```

When we execute the above code, it produces the following result:

```
[1] "CHANGING TO UPPER"
[1] "changing to lower"
```

Extracting parts of a string - substring() function

This function extracts parts of a String.

Syntax

The basic syntax for substring() function is :

```
substring(x,first,last)
```

Following is the description of the parameters used:

- **x** is the character vector input.
- **first** is the position of the first character to be extracted.

- **last** is the position of the last character to be extracted.

Example

```
# Extract characters from 5th to 7th position.
result <- substring("Extract", 5, 7)
print(result)
```

When we execute the above code, it produces the following result:

```
[1] "act"
```

Lab Session 5

Lists, Arrays, Data Frames

Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.
list1 <- list(c(2,5,3),21.3,”hello”,20L)
# Print the list.
print(list1)
list1[[1]][1]
```

When we execute the above code, it produces the following result:

```
[[1]]
[1] 2 5 3
[[2]]
[1] 21.3
```

```
> x <- list(1:3, TRUE, "Hello", list(1:2, 5))
```

Here x has 4 elements: a numeric vector, a logical, a string and another list.

We can select an entry of x with double square brackets:

```
> x[[3]]
[1] "Hello"
```

To get a sub-list, use single brackets:

```
> x[c(1,3)]
[[1]]
[1] 1 2 3
[[2]]
[1] "Hello"
```

Notice the difference between x[[3]] and x[3].

We can also name some or all of the entries in our list, by supplying argument names to list():

```
> x <- list(y=1:3, TRUE, z="Hello")
> x$y
```

[1] 1 2 3

[[2]]

[1] TRUE

x$z

[1]

Notice that the [[1]] has been replaced by $y, which gives us a clue as to how we can recover the entries by their name. We can still use the numeric position if we prefer:

```
> x$y
[1] 1 2 3
> x[[1]]
[1] 1 2 3
```

The function names() can be used to obtain a character vector of all the names of objects in a list.

```
> names(x)
[1] "y" "" "z"
```

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.
a <- array(c('green','yellow'),dim=c(3,3,2))
print(a)
```

When we execute the above code, it produces the following result:

```
, , 1
[,1] [,2] [,3]
[1,] "green" "yellow" "green"
[2,] "yellow" "green" "yellow"
[3,] "green" "yellow" "green"
, , 2
[,1] [,2] [,3]
[1,] "yellow" "green" "yellow"
[2,] "green" "yellow" "green"
[3,]  "yellow" "green" "yellow"
```

Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modeling.

Factors are created using the **factor()** function.The **nlevels** functions gives the count of levels.

```
# Create a vector.
apple_colors <- c('green','green','yellow','red','red','red','green')
# Create a factor object.
factor_apple <- factor(apple_colors)
# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

When we execute the above code, it produces the following result:

```
[1] green green yellow red red red yellow green
Levels: green red yellow
# applying the nlevels function we can know the number of distinct values
[1] 3
```

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame()** function.

```
# Create the data frame.
BMI <- data.frame(
                    gender = c("Male", "Male","Female"),
                    height = c(152, 171.5, 165),
                    weight = c(81,93, 78),
                    Age =c(42,38,26)
                  )

print(BMI)
```

```
gender = c("Male", "Male","Female")
height = c(152, 171.5, 165)
weight = c(81,93, 78)
Age =c(42,38,26)
df=data.frame(gender,height, weight,age)
Print(df)
```

When we execute the above code, it produces the following result:

```
  gender height weight Age
1 Male 152.0 81 42
2 Male 171.5 93 38
3 Female 165.0 78 26
```

Accessing data from Dataframe:

Syntax: Df[row,column]

Extract first two rows:

BMI[1:2,]

Extract 3rd and 4th row with 2nd and 4th column

BMI[c(3,5),c(2,4)]

BMI$height

Access subframe from an existing frame

Syntax: Newdf=subset(df,conditions)

Create a new dataframe with age>30 from the existing dataframe BMI

newbmi=subset(bmi,age>30)

Print(newbmi)

Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

BMI$place <- c("Blore","Mlore","Mysore",”Delhi”)

v <- BMI

print(v)

Add column using cbind

v=cbind(BMI, place= c("Blore","Mlore","Mysore"))

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the rbind() function.

Newdf=rbind(df, new entries)

newbmi=rbind(bmi, data.frame(gender='M', height=156, weight=67,age=23,place='Delhi'))

newbmi=rbind(bmi,c('M',156,67,23,'Delhi'), c('F,156,67,23,'Delhi'))

R comes with many datasets built-in, particularly in the MASS package. A package is a collection (or library) of functions, datasets, and other objects; most packages are not loaded automatically, so you have to do it yourself:

> library(MASS)

You can now access various datasets from this package. Try looking at the dataset called hills.

> head(hills)

To find out what the data represent, use the help function ?hills.

Data Frames

The object hills is something called a data frame. A data frame is a series of records represented by rows (in this case one for each race), each containing values in several fields (in this case dist, climb, time).

You can check that hills is a data frame by inspecting its class.

> class(hills)

[1] "data.frame"

or more reliably by using an is() command

> is(hills, "data.frame")

[1] TRUE

Data frames share many of the characteristics of matrices. We can select rows or columns in the same way:

> **hills[3,]**

dist climb time

Craig Dunain 6 900 33.65

Hills[hills$dist>=12,1:2] #to display first two columns

```
> hills[hills$dist >= 12,]

              dist climb   time
Bens of Jura    16  7500 204.62
Lairig Ghru     28  2100 192.67
Seven Hills     14  2200  98.42
Two Breweries   18  5200 170.25
Moffat Chase    20  5000 159.83
```

However, they also behave like lists indexed by the columns:

```
> hills$time

 [1]  16.08  48.35  33.65  45.60  62.27  73.22 204.62  36.37  29.75  39.75
[11] 192.67  43.05  65.00  44.13  26.93  72.25  98.42  78.65  17.42  32.57
[21]  15.95  27.90  47.63  17.93  18.68  26.22  34.43  28.57  50.50  20.95
[31]  85.58  32.38 170.25  28.10 159.83
```

Manipulating Data using with()

We often want to use functions on the columns of a data frame, and it quickly becomes inconvenient to repeatedly type (for example) hills$ before every such event. For example, the command below will give a scatter plot of the race times against climbs, amongst only those races less than 10 miles long.

```
> plot(hills$climb[hills$dist < 10], hills$time[hills$dist < 10])
```

```
> with(hills, plot(climb[dist < 10], time[dist < 10]))
```

The command data.frame() is used to create a data frame, each argument representing a column.

```
> books <- data.frame(author=c("Ripley", "Cox", "Snijders", "Cox"),
+ year=c(1980, 1979, 1999, 2006),
+ publisher=c("Wiley", "Chapman", "Sage", "CUP"))
> books
   author year publisher
1 Ripley 1980      Wiley
2     Cox 1979   Chapman
3 Snijders 1999      Sage
4     Cox 2006       CUP
```

Exercise (a) Create a data frame representing a database of films. It should contain the fields title, director, year, country, and at least three films.

Create a second data frame of the same format as above, but containing just one new film.

(b) Merge the two data frames using rbind().

(c) Add two more rows to the data frame

(d) Add one more column genre

(d) Try sorting the titles

order() syntax:

#order(dataframe, decreasing=TRUE)

film1=film[order(film$title),] # sort the dataframe based on title in ascending order

film1=film[order(film$title, decresing=TRUE),]   # descending order

film1=film[order(film$title,film$year),] #sort on multiple columns

Lab Session 6

Importing and Exporting Data and Data visualization

Working with CSV files in R Programming

sample.csv

```
id, name, department, salary, projects
1,   A,     IT,         60754,   4
2,   B,     Tech,       59640,   2
3,   C,     Marketing,  69040,   8
4,   D,     Marketing,  65043,   5
5,   E,     Tech,       59943,   2
6,   F,     IT,         65000,   5
7,   G,     HR,         69000,   7
```

Reading a CSV file

```
df <- read.csv(file = 'sample.csv')
print(df)

# print number of columns
print (ncol(df))

# print number of rows
print(nrow(df))
```

Querying with CSV files

```
min_pro <- min(df$projects)
print(min_pro)


newdf <- subset(df, department == "HR" & projects <10)
print (newdf)
```

Writing into a CSV file

```
write.csv(newdf, "new_sample.csv")
```

```
new_data <-read.csv(file ='new_sample.csv')
print(new_data)
```

The column X contains the row numbers of the original CSV file. In order to remove it, we can specify an additional argument in the write.csv() function that set row names to FALSE.

```
csv_data <- read.csv(file ='sample.csv')

new_csv <- subset(csv_data, department == "HR" & projects <10)

write.csv(new_csv, "new_sample.csv", row.names = FALSE)

new_data <-read.csv(file ='new_sample.csv')

print(new_data)
```

- **Data visualization** is the technique used to deliver insights in data using visual cues such as graphs, charts, maps, and many others.
- This is useful as it helps in intuitive and easy understanding of the large quantities of data and thereby make better decisions regarding it.
- R is a language that is designed for statistical computing, graphical data analysis, and scientific research.
- It is usually preferred for data visualization as it offers flexibility and minimum required coding through its packages.

Consider the following *airquality* data set for visualization in R:

Ozone	Solar R.	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8.0	72	5	2
12	149	12.6	74	5	3
18	313	11.5	62	5	4
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

Types of Data Visualizations
Some of the various types of visualizations offered by R are:

1. Bar Plot

There are two types of bar plots- horizontal and vertical which represent data points as horizontal or vertical bars of certain lengths proportional to the value of the data item.

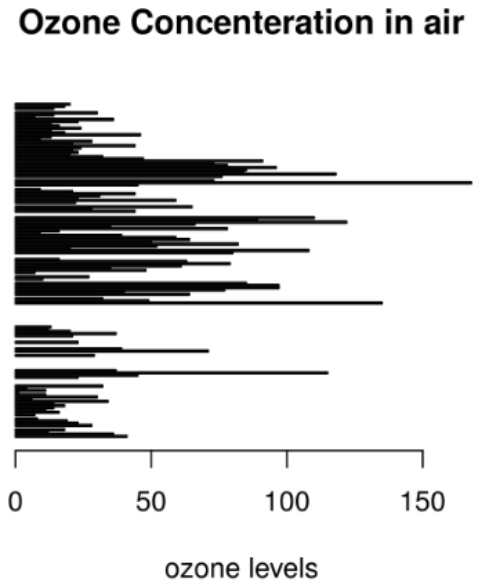
They are generally used for continuous and categorical variable plotting.

By setting the horiz parameter to true and false, we can get horizontal and vertical bar plots respectively.

Example 1:

```
# Horizontal Bar Plot for Ozone concentration in air
barplot (airquality$Ozone, main = 'Ozone Concenteration in air',
        xlab = 'ozone levels', horiz = TRUE)
```

Output:

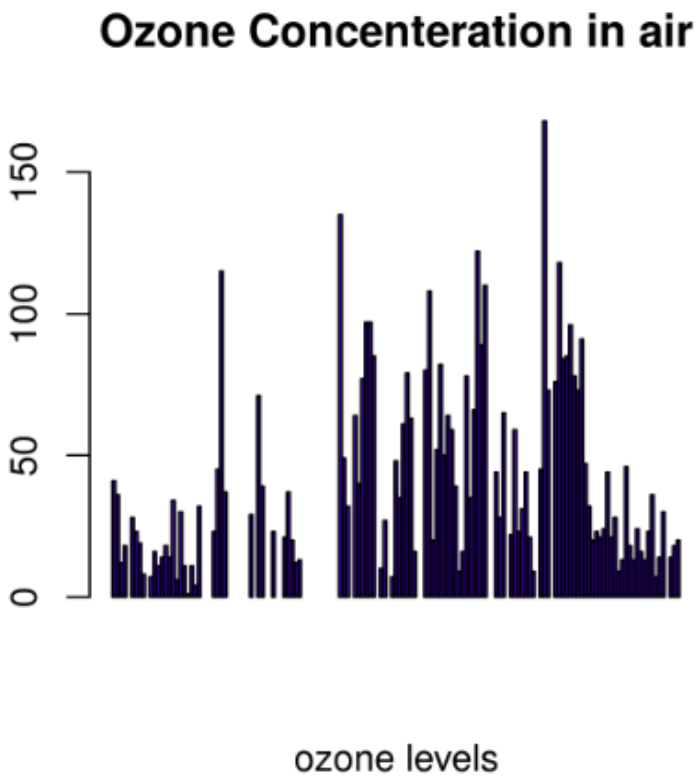


Example 2:

 # Vertical Bar Plot for Ozone concentration in air

barplot(airquality$Ozone, main = 'Ozone Concenteration in air', xlab = 'ozone levels', col ='blue', horiz = FALSE)

Output:



Bar plots are used for the following scenarios:

- To perform a comparative study between the various data categories in the data set.
- To analyze the change of a variable over time in months or years.

Histogram

A histogram is like a bar chart as it uses bars of varying height to represent data distribution.

However, in a histogram values are grouped into consecutive intervals called bins.

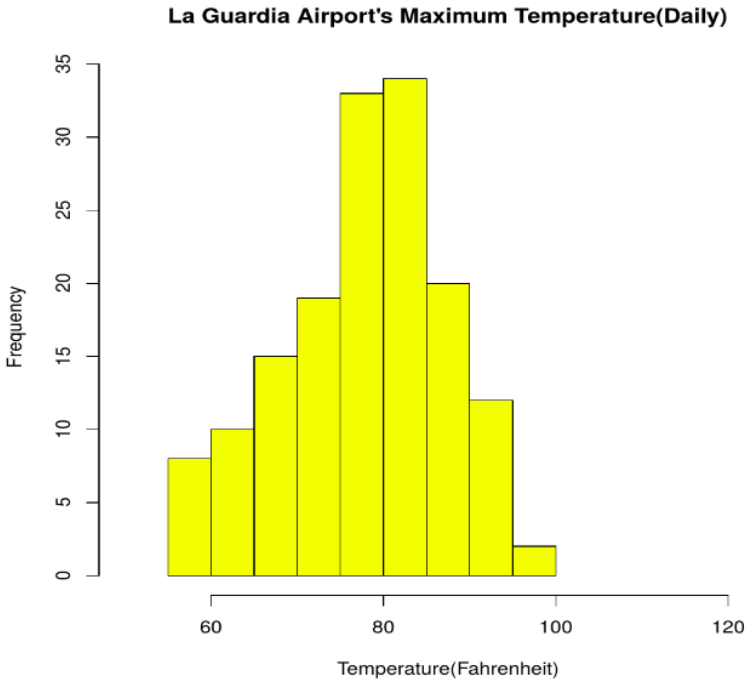
In a Histogram, continuous values are grouped and displayed in these bins whose size can be varied.

Example:

Histogram for Maximum Daily Temperature data (airquality)

```
 hist(airquality$Temp, main ="La Guardia Airport's\maximum
Temperature(Daily)",     xlab ="Temperature(Fahrenheit)",     xlim = c(50, 125),
col ="yellow",     freq = TRUE)
```

Output:



For a histogram, the parameter **xlim** can be used to specify the interval within which all values are to be displayed.

Another parameter **freq** when set to *TRUE* denotes the frequency of the various values in the histogram.

Histograms are used in the following scenarios:

- To verify an equal and symmetric distribution of the data.
- To identify deviations from expected values.

Box Plot

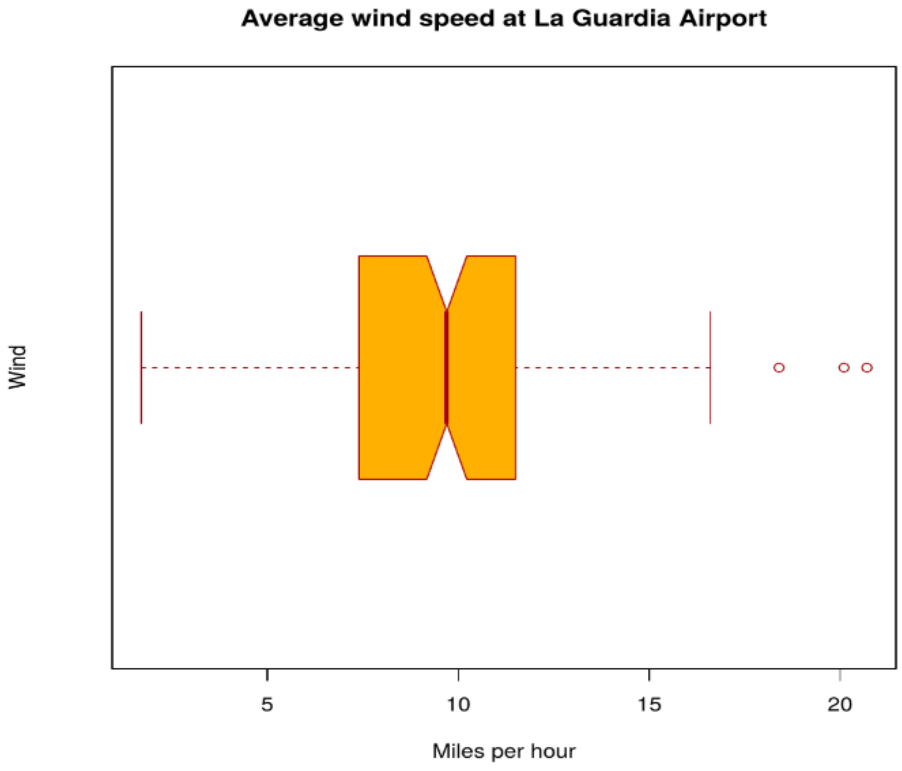
The statistical summary of the given data is presented graphically using a boxplot. A boxplot depicts information like the minimum and maximum data point, the median value, first and third quartile, and interquartile range.

Example:

 # Box plot for average wind speed data(airquality)

```
boxplot(airquality$Wind, main = "Average wind speed\at La Guardia Airport",
xlab = "Miles per hour", ylab = "Wind",
col = "orange",
border = "brown",
horizontal = TRUE)
```

Output:



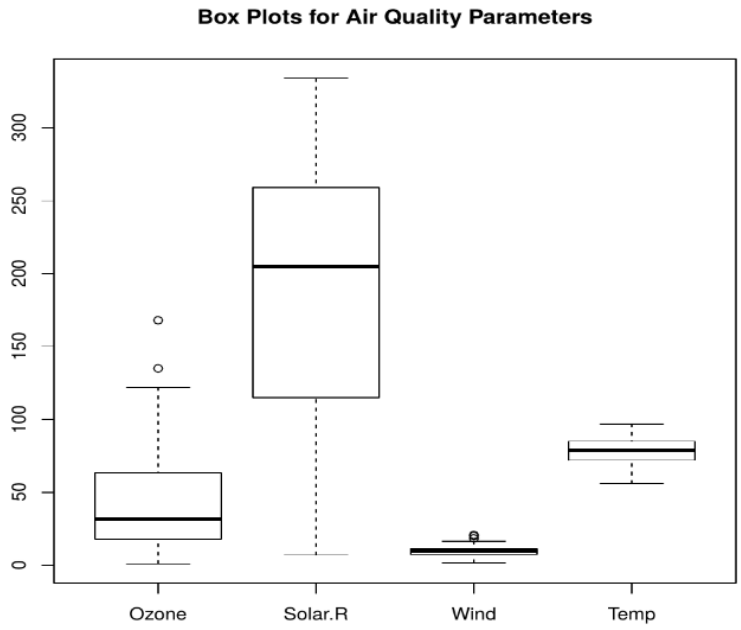
Multiple box plots can also be generated at once through the following code:

Example:

 # Multiple Box plots, each representing an Air Quality Parameter

boxplot(airquality[, 1:4], main ='Box Plots for Air Quality Parameters')

Output:



Box Plots are used for:

- **To give a comprehensive statistical description of the data through a visual cue.**
- **To identify the outlier points that do not lie in the inter-quartile range of data.**

Scatter Plot

A scatter plot is composed of many points on a Cartesian plane. Each point denotes the value taken by two parameters and helps us easily identify the relationship between them**.**

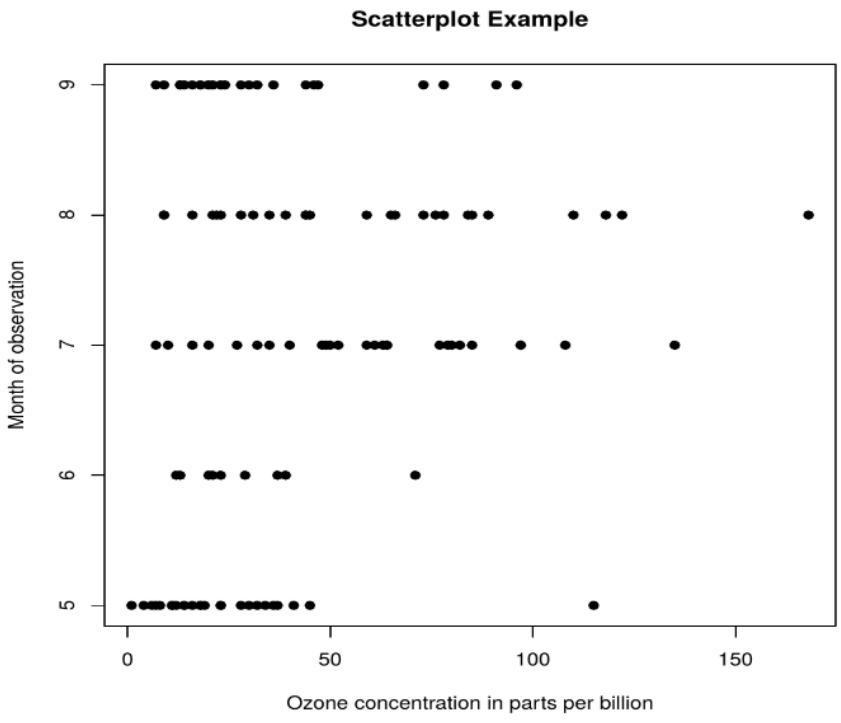
Example:

 # Scatter plot for Ozone Concentration per month

plot(airquality$Ozone, airquality$Month, main ="Scatterplot Example",

xlab ="Ozone Concentration in parts per billion", ylab =" Month of observation ")

Output:



Scatter Plots are used in the following scenarios:

- **To show whether an association exists between bivariate data.**
- **To measure the strength and direction of such a relationship.**

Heat Map

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. heatmap() function is used to plot heatmap.

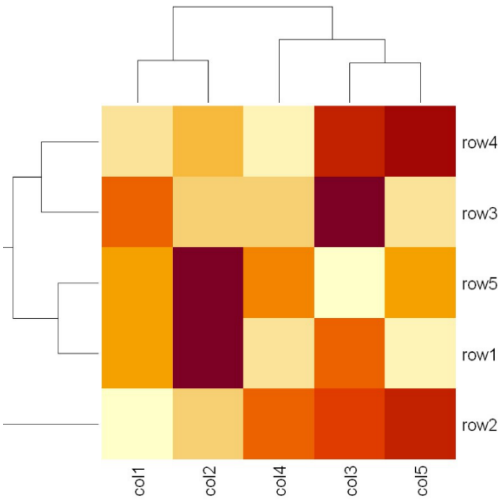
Syntax: heatmap(data)

Parameters: data: It represent matrix data, such as values of rows and columns

Return: This function draws a heatmap.

```
# Set seed for reproducibility
# set.seed(110)
# Create example data
data <- matrix(rnorm(25, 0, 5), nrow = 5, ncol = 5)
# Column names
colnames(data) <- paste("col", 1:5)
rownames(data) <- paste("row", 1:5)
  # Draw a heatmap
heatmap(data)
```

Output:



Map visualization in R

Here we are using maps package to visualize and display geographical maps using an R programming language.

install.packages("maps")

Link of the dataset: worldcities.csv

 # Read dataset and convert it into Dataframe

data <- read.csv("worldcities.csv")

df <- data.frame(data)

 # Load the required libraries

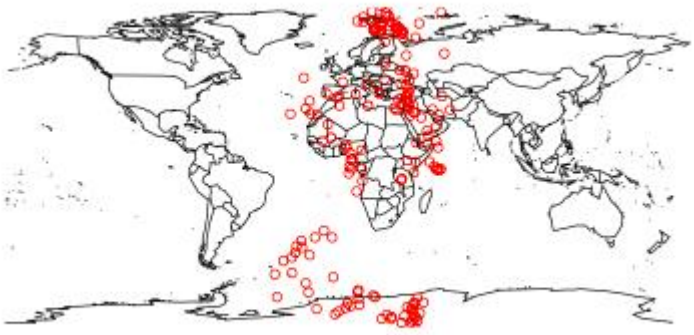
library(maps)

map(database = "world")

 # marking points on map

```
points(x = df$lat[1:500], y = df$lng[1:500], col = "Red")
```

Output:



Lab Session 7

What is dplyr?

The dplyr is a powerful R-package to manipulate, clean and summarize unstructured data. In short, it makes data exploration and data manipulation easy and fast in R.

What's special about dplyr?

- 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. Also easy to recall these functions. For example, **filter()** is used to filter rows.

How to install and load dplyr package

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

dplyr vs. Base R Functions

- dplyr functions process faster than base R functions.
- It is because dplyr functions were written in a computationally efficient manner.
- They are also more stable in the syntax and better supports data frames than vectors.

SQL Queries vs. dplyr

The names of dplyr functions are similar to SQL commands such as select() for selecting variables, group_by() - group data by grouping variable, join() - joining two data sets. Also includes inner_join() and left_join(). It also supports sub queries for which SQL was popular for.

Data : Income Data by States

- we are using the following data which contains income generated by states from year 2002 to 2015.

- This dataset contains 51 observations (rows) and 16 variables (columns). The snapshot of first 6 rows of the dataset is shown below.

```
 Index     State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387
Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
5 1624509 1639670 1921845 1156536 1388461 1644607
6 1913275 1665877 1491604 1178355 1383978 1330736
```

How to load Data

Submit the following code to load data directly from link. If you want to load the data from your local drive, you need to change the file path in the code below.

```
mydata =
read.csv("https://raw.githubusercontent.com/deepanshu88/data/master/sampledata.csv")
```

dplyr Practical Examples

Example 1 : Selecting Random N Rows

The **sample_n** function selects random rows from a data frame (or table). The second parameter of the function tells R the number of rows to select.

```
sample_n(mydata,3)
```

```
  Index    State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
2     A   Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
8     D Delaware 1330403 1268673 1706751 1403759 1441351 1300836 1762096 1553585
33    N New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216
   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
2  1629616 1230866 1512804 1985302 1580394 1979143
8  1370984 1318669 1984027 1671279 1803169 1627508
33 1604531 1683687 1500089 1718837 1619033 1367705
```

Example 2 : Selecting Random Fraction of Rows

The **sample_frac** function returns randomly N% of rows. In the example below, it returns randomly 10% of rows.

```
sample_frac(mydata,0.1)
```

Example 3 : Remove Duplicate Rows based on all the variables (Complete Row)

The **distinct function** is used to eliminate duplicates.

```
x1 = distinct(mydata)
```

In this dataset, there is not a single duplicate row so it returned same number of rows as in mydata.

Example 4 : Remove Duplicate Rows based on a variable

The **.keep_all** function is used to retain all other variables in the output data frame.

```
x2 = distinct(mydata, Index, .keep_all= TRUE)
```

Example 5 : Remove Duplicates Rows based on multiple variables

In the example below, we are using two variables - **Index, Y2010** to determine uniqueness.

```
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)
```

select( ) Function

It is used to select only desired variables.

```
select() syntax : select(data , ....)

data : Data Frame

.... : Variables by name or by function
```

Example 6 : Selecting Variables (or Columns)

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

```
mydata2 = select(mydata, Index, State:Y2008)
```

Example 7 : Dropping Variables

The **minus sign** before a variable tells R to drop the variable.

```
Mydata2 = select(mydata, -Index, -State)
```

The above code can also be written like :

```
Mydata2 = select(mydata, -c(Index,State))

print(select(mydata,1:5))
```

Example 8 : Selecting or Dropping Variables starts with 'Y'

The **starts_with()** function is used to select variables starts with an alphabet.

```
mydata3 = select(mydata, starts_with("Y"))
```

Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'

```
mydata33 = select(mydata, -starts_with("Y"))
```

The following functions helps you to select variables based on their names.

Helpers	Description
starts_with()	Starts with a prefix
ends_with()	Ends with a prefix
contains()	Contains a literal string
matches()	Matches a regular expression
num_range()	Numerical range like x01, x02, x03.
one_of()	Variables in character vector.
everything()	All variables.

Example 9 : Selecting Variables contain 'I' in their names

```
mydata4 = select(mydata, contains("I"))
print(select(mydata, matches("y20*")))
print(select(mydata, matches("y200[67]")))
```

Example 10 : Reorder Variables

The code below keeps variable **'State'** in the front and the remaining variables follow that.

```
mydata5 = select(mydata, State, everything())
```

New order of variables are displayed below -

```
 [1] "State" "Index" "Y2002" "Y2003" "Y2004" "Y2005" "Y2006" "Y2007" "Y2008" "Y2009"
[11] "Y2010" "Y2011" "Y2012" "Y2013" "Y2014" "Y2015"
```

rename( ) Function

It is used to change variable name.

```
rename() syntax : rename(data , new_name = old_name)

data : Data Frame

new_name : New variable name you want to keep

old_name : Existing Variable Name
```

Example 11 : Rename Variables

The rename function can be used to rename variables.

In the following code, we are renaming **'Index'** variable to **'Index1'**.

```
mydata6 = rename(mydata, Index1=Index)
```

```
> names(mydata6)
 [1] "Index1" "State"  "Y2002"
[10] "Y2009"  "Y2010"  "Y2011"
```

Output

filter( ) Function

It is used to subset data with matching logical conditions.

```
filter() syntax : filter(data , ....)

data : Data Frame

.... : Logical Condition
```

Example 12 : Filter Rows

Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.

```
mydata7 = filter(mydata, Index == "A")
```

```
 Index   State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009
1     A  Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2     A   Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3     A  Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4     A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980
Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
```

Example 13 : Multiple Selection Criteria

The **%in%** operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.

```
mydata7 = filter(mydata6, Index %in% c("A", "C"))
```

Example 14 : 'AND' Condition in Selection Criteria

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.

```
mydata8 = filter(mydata6, Index %in% c("A", "C") & Y2002 >= 1300000 )
```

Example 15 : 'OR' Condition in Selection Criteria

The 'I' denotes OR in the logical condition. It means any of the two conditions.

```
mydata9 = filter(mydata6, Index %in% c("A", "C") | Y2002 >= 1300000)
```

Example 16 : NOT Condition

The "!" sign is used to reverse the logical condition.

```
mydata10 = filter(mydata6, !Index %in% c("A", "C"))
```

Example 17 : CONTAINS Condition

The **grepl function** is used to search for pattern matching. In the following code, we are looking for records wherein column **state** contains **'Ar'** in their name.

```
mydata10 = filter(mydata6, grepl("Ar", State))
```

summarise( ) Function

It is used to summarize data.

```
summarise() syntax : summarise(data , ....)

data : Data Frame

..... : Summary Functions such as mean, median etc
```

Example 18 : Summarize selected variables

In the example below, we are calculating mean and median for the variable Y2015.

```
summarise(mydata, Y2015_mean = mean(Y2015), Y2015_med=
median(Y2015))
```

```
Y2015_mean Y2015_med
   1588297    1627508
```

Output

Example 19 : Summarize Multiple Variables

In the following example, we are calculating number of records, mean and median for variables Y2005 and Y2006. The **summarise_at** function allows us to select multiple variables by their names.

```
summarise_at(mydata, vars(Y2005, Y2006), funs(n(), mean, median))
```

Example 20 : Summarize with Custom Functions

Incase you want to add additional arguments for the functions mean and median (for example **na.rm = TRUE**), you can do it like the code below.

You can use the argument na. rm = TRUE to exclude missing values when calculating descriptive statistics in R.

```
summarise_at(mydata, vars(Y2011, Y2012),funs(mean, median), na.rm = TRUE)
```

We can also use custom functions in the summarise function. In this case, we are computing the number of records, number of missing values, mean and median for variables Y2011 and Y2012. The **dot (.)** denotes each variables specified in the second argument of the function.

```
summarise_at(mydata, vars(Y2011, Y2012),
funs(n(), missing = sum(is.na(.)), mean(., na.rm = TRUE), median(.,na.rm = TRUE)))
```

Instead of funs( ), you should make a habit of using list( ) as funs( ) can be dropped in future versions of dplyr package.

```
summarise_at(mydata, vars(Y2011, Y2012),
list(~n(), missing = ~sum(is.na(.)), ~mean(., na.rm = TRUE), ~median(.,na.rm = TRUE)))
```

```
Y2011_n Y2012_n Y2011_missing Y2012_missing Y2011_mean Y2012_mean Y2011_median
     51      51             0             0    1574968    1591135      1575533
Y2012_median
     1643855
```

Summarize : Output

Example 21 : Summarize all Numeric Variables

The **summarise_if** function allows you to summarise conditionally.

```
summarise_if(mydata, is.numeric, funs(n(),mean,median))
```

Second, the **summarise_all** function calculates summary statistics for all the columns in a data frame

```
summarise_all(mydata, funs(n(),mean,median))
```

arrange() function

Use : Sort data

Syntax

```
arrange(data_frame, variable(s)_to_sort)

or

data_frame %>% arrange(variable(s)_to_sort)
```

To sort a variable in descending order, use **desc(x)**.

Example 23 : Sort Data by Multiple Variables

The default sorting order of **arrange() function** is ascending. In this example, we are sorting data by multiple variables.

```
arrange(mydata, Index, Y2011)
```

Suppose you need to sort one variable by descending order and other variable by ascending oder.

```
arrange(mydata, desc(Index), Y2011)
```

Pipe Operator %>%

It is important to understand the pipe (%>%) operator before knowing the other functions of dplyr package. dplyr utilizes pipe operator from another package **(magrittr)**.

It allows you to write sub-queries like we do it in sql.

Note : All the functions in dplyr package can be used **without** the pipe operator. The question arises **"Why to use pipe operator %>%". The answer is** it lets to wrap multiple functions together with the use of %>%.

Syntax :

```
filter(data_frame, variable == value)

or

data_frame %>% filter(variable == value)
```

The %>% is NOT restricted to filter function. It can be used with any function.

Example :

The code below demonstrates the usage of pipe %>% operator. In this example, we are selecting 10 random observations of two variables "Index" "State" from the data frame "mydata".

```
dt = sample_n(select(mydata, Index, State),10)

or

dt = mydata %>% select(Index, State) %>% sample_n(10)

a=mydata%>% select(State,Y2002) %>%filter(State=='California')
```

	Index	State
44	T	Texas
32	N	New Mexico
51	W	Wyoming
9	D	District of Columbia
5	C	California
40	R	Rhode Island
22	M	Massachusetts
4	A	Arkansas
42	S	South Dakota
46	V	Vermont

Output

group_by() function

Use : Group data by categorical variable

Syntax :

```
group_by(data, variables)

or

data %>% group_by(variables)
```

Example 24 : Summarise Data by Categorical Variable

We are calculating count and mean of variables Y2011 and Y2012 by variable Index.

```
t = summarise_at(group_by(mydata, Index), vars(Y2011, Y2012), funs(n(), mean(., na.rm = TRUE)))
```

The above code can also be written like

```
t = mydata %>% group_by(Index) %>%
  summarise_at(vars(Y2011:Y2015), funs(n(), mean(., na.rm = TRUE)))
```

```
Index Y2011_n Y2012_n  Y2013_n Y2014_n Y2015_n Y2011_mean Y2012_mean
A 4 4 4 4 4 1432642 1455876
C 3 3 3 3 3 1750357 1547326
D 2 2 2 2 2 1336059 1981868
F 1 1 1 1 1 1497051 1131928
G 1 1 1 1 1 1851245 1850111
H 1 1 1 1 1 1902816 1695126
I 4 4 4 4 4 1690171 1687056
K 2 2 2 2 2 1489353 1899773
L 1 1 1 1 1 1210385 1234234
M 8 8 8 8 8 1582714 1586091
N 8 8 8 8 8 1448351 1470316
O 3 3 3 3 3 1882111 1602463
P 1 1 1 1 1 1483292 1290329
R 1 1 1 1 1 1781016 1909119
S 2 2 2 2 2 1381724 1671744
T 2 2 2 2 2 1724080 1865787
U 1 1 1 1 1 1288285 1108281
V 2 2 2 2 2 1482143 1488651
W 4 4 4 4 4 1711341 1660192
```

Since dplyr >= 1.0.0 version you may get the following warnings.

```
#`summarise()` ungrouping output (override with `.groups` argument)
#`summarise()` regrouping output by xxx (override with `.groups` argument)
```

To suppress this warning you can use the following command.

```
options(dplyr.summarise.inform=F)
```

do() function

Use : Compute within groups

Syntax :

```
do(data_frame, expressions_to_apply_to_each_group)
```

Note : *The **dot (.)** is required to refer to a data frame.*

Example 25 : Filter Data within a Categorical Variable

Suppose you need to pull top 2 rows from 'A', 'C' and 'I' categories of variable Index.

```
t = mydata %>% filter(Index %in% c("A", "C","I")) %>% group_by(Index) %>%
  do(head( . , 2))
```

	Index	State	Y2002	Y2003	Y2004	Y2005	Y2006
1	A	Alabama	1296530	1317711	1118631	1492583	1107408
2	A	Alaska	1170302	1960378	1818085	1447852	1861639
3	C	California	1685349	1675807	1889570	1480280	1735069
4	C	Colorado	1343824	1878473	1886149	1236697	1871471
5	I	Idaho	1353210	1438538	1739154	1541015	1122387
6	I	Illinois	1508356	1527440	1493029	1261353	1540274

Output : do() function

Example 26 : Selecting 3rd Maximum Value by Categorical Variable

We are calculating third maximum value of variable Y2015 by variable Index. The following code first selects only two variables Index and Y2015. Then it filters the variable Index with 'A', 'C' and 'I' and then it groups the same variable and sorts the variable Y2015 in descending order. At last, it selects the third row.

```
t = mydata %>% select(Index, Y2015) %>%
  filter(Index %in% c("A", "C","I")) %>%  group_by(Index) %>%
  do(arrange(.,desc(Y2015))) %>%  slice(3)
```

The **slice() function** is used to select rows by position.

	Index	Y2015
1	A	1647724
2	C	1330736
3	I	1583516

Output

Using Window Functions

Like SQL, dplyr uses window functions that are used to subset data within a group. It returns a vector of values. We could use **min_rank() function** that calculates rank in the preceding example,

```
t = mydata %>% select(Index, Y2015) %>%
  filter(Index %in% c("A", "C","I")) %>%
  group_by(Index) %>%
  filter(min_rank(desc(Y2015)) == 3)
```

```
  Index   Y2015
1     A 1647724
2     C 1330736
3     I 1583516
```

Example 27 : Summarize, Group and Sort Together

In this case, we are computing mean of variables Y2014 and Y2015 by variable Index. Then sort the result by calculated mean variable Y2015.

```
t = mydata %>%   group_by(Index)%>%
  summarise(Mean_2014 = mean(Y2014, na.rm=TRUE),
            Mean_2015 = mean(Y2015, na.rm=TRUE)) %>%
arrange(desc(Mean_2015))
```

mutate() function

Use : Creates new variables

Syntax :

```
mutate(data_frame, expression(s) )
or
data_frame %>% mutate(expression(s))
```

Example 28 : Create a new variable

The following code calculates division of Y2015 by Y2014 and name it "change".

```
mydata1 = mutate(mydata, change=Y2015/Y2014)
```

Example 29 : Multiply all the variables by 1000

It creates new variables and name them with suffix "_new".

```
mydata11 = mutate_all(mydata, funs("new" = .* 1000))
```

Y2002_new	Y2003_new	Y2004_new	Y2005_new
1296530000	1317711000	1118631000	1492583000
1170302000	1960378000	1818085000	1447852000
1742027000	1968140000	1377583000	1782199000
1485531000	1994927000	1119299000	1947979000
1685349000	1675807000	1889570000	1480280000

Output

The output shown in the image above is truncated due to high number of variables.

Note - The above code returns the following error messages -

Warning messages:

1: In Ops.factor(c(1L, 1L, 1L, 1L, 2L, 2L, 2L, 3L, 3L, 4L, 5L, 6L, : '*' not meaningful for factors

2: In Ops.factor(1:51, 1000) : '*' not meaningful for factors

It implies you are multiplying 1000 to string(character) values which are stored as factor variables. These variables are 'Index', 'State'. It does not make sense to apply multiplication operation on character variables. For these two variables, it creates newly created variables which contain only NA.

Solution : See Example 45 - Apply multiplication on only numeric variables

Example 30 : Calculate Rank for Variables

Suppose you need to calculate rank for variables Y2008 to Y2010.

```
mydata12 = mutate_at(mydata, vars(Y2008:Y2010), funs(Rank=min_rank(.)))
```

Y2008_Rank	Y2009_Rank	Y2010_Rank
47	46	8
27	9	38
33	14	12
8	24	40
24	27	36
43	31	47
37	50	48

Output

By default, **min_rank()** assigns 1 to the smallest value and high number to the largest value. In case, you need to assign rank 1 to the largest value of a variable, use **min_rank(desc(.))**

```
mydata13 = mutate_at(mydata, vars(Y2008:Y2010), funs(Rank=min_rank(desc(.))))
```

Example 31 : Select State that generated highest income among the variable 'Index'

Install.packages("magrittr")

Library(magrittr) #if the error is %>% is not found

```
out = mydata %>% group_by(Index) %>% filter(min_rank(desc(Y2015)) == 1) %>%  select(Index, State, Y2015)
```

```
  Index        State   Y2015
1     A       Alaska 1979143
2     C  Connecticut 1718072
3     D     Delaware 1627508
4     F      Florida 1170389
```

```
5     G       Georgia 1725470
6     H        Hawaii 1150882
7     I         Idaho 1757171
8     K      Kentucky 1913350
9     L     Louisiana 1403857
10    M      Missouri 1996005
11    N New Hampshire 1963313
12    O        Oregon 1893515
13    P  Pennsylvania 1668232
14    R  Rhode Island 1611730
15    S  South Dakota 1136443
16    T         Texas 1705322
17    U          Utah 1729273
18    V      Virginia 1850394
19    W       Wyoming 1853858
```

Example 32 : Cumulative Income of 'Index' variable

The **cumsum function** calculates cumulative sum of a variable. With **mutate function,** we insert a new variable called 'Total' which contains values of cumulative income of variable Index.

```
out2 = mydata %>% group_by(Index) %>%
mutate(Total=cumsum(Y2015)) %>%select(Index, Y2015, Total)
```

join() function

Use : Join two datasets

Syntax :

```
inner_join(x, y, by = )
left_join(x, y, by = )
right_join(x, y, by = )
```

```
full_join(x, y, by = )
semi_join(x, y, by = )
anti_join(x, y, by = )
```

x, y - datasets (or tables) to merge / join

by - common variable (primary key) to join by.

Example 33 : Common rows in both the tables

```
df1 = data.frame(ID = c(1, 2, 3, 4, 5),
w = c('a', 'b', 'c', 'd', 'e'),
x = c(1, 1, 0, 0, 1),
y=rnorm(5),
z=letters[1:5])
df2 = data.frame(ID = c(1, 7, 3, 6, 8),
a = c('z', 'b', 'k', 'd', 'l'),
b = c(1, 2, 3, 0, 4),
c =rnorm(5),
d =letters[2:6])
```

INNER JOIN returns rows when there is a match in both tables. In this example, we are merging df1 and df2 with ID as common variable (primary key).

```
df3 = inner_join(df1, df2, by = "ID")
```

```
  ID w x          y z a b          c d
1  1 a 1 -0.9934455 a z 1 -0.6556326 b
2  3 c 0 -1.4342218 c k 3 -1.4055054 d
```

Output : INNER JOIN

If the primary key does not have same name in both the tables, try the following way:

```
inner_join(df1, df2, by = c("ID"="ID1"))
```

Example 34 : Applying LEFT JOIN

LEFT JOIN : It returns all rows from the left table, even if there are no matches in the right table.

```
left_join(df1, df2, by = "ID")
```

	ID	w	x	y	z	a	b	c	d
1	1	a	1	-0.9934455	a	z	1	-0.6556326	b
2	2	b	1	-1.3061685	b	<NA>	NA	NA	<NA>
3	3	c	0	-1.4342218	c	k	3	-1.4055054	d
4	4	d	0	-0.8628479	d	<NA>	NA	NA	<NA>
5	5	e	1	1.7037992	e	<NA>	NA	NA	<NA>

Output : LEFT JOIN

Combine Data Vertically

intersect(x, y)

Rows that appear in both x and y.

union(x, y)

Rows that appear in either or both x and y.

setdiff(x, y)

Rows that appear in x but not y.

Example 35 : Applying INTERSECT

Prepare Sample Data for Demonstration

```
mtcars$model <- rownames(mtcars)

first <- mtcars[1:20, ]

second <- mtcars[10:32, ]
```

INTERSECT selects unique rows that are common to both the data frames.

```
intersect(first, second)

union(first,second)

union_all(first,second)

setdiff(first,second)
```

Example 36 : Applying UNION

UNION displays all rows from both the tables and removes duplicate records from the combined dataset. By using **union_all function**, it allows duplicate rows in the combined dataset.

```
x=data.frame(ID = 1:6, ID1= 1:6)

y=data.frame(ID = 1:6,  ID1 = 1:6)

union(x,y)

union_all(x,y)
```

Example 37 : Rows appear in one table but not in other table

```
setdiff(first, second)
```

Example 38 : IF ELSE Statement

Syntax :

```
if_else(condition, true, false, missing = NULL)
```

true : Value if condition meets

false : Value if condition does not meet

missing : Value if missing cases. It will be used to replace missing values (Default : NULL)

```
df <- c(-10,2, NA)

if_else(df < 0, "negative", "positive", missing = "missing value")
```

Create a new variable with IF_ELSE

If a value is less than 5, add it to 1 and if it is greater than or equal to 5, add it to 2. Otherwise 0.

```
df =data.frame(x = c(1,5,6,NA))

df %>% mutate(newvar=if_else(x<5, x+1, x+2,0))
```

x	newvar
1	2
5	7
6	8
NA	0

Output

Nested IF ELSE

Multiple IF ELSE statement can be written using if_else() function. See the example below -

```
mydf =data.frame(x = c(1:5,NA))

mydf %>% mutate(newvar= if_else(is.na(x),"I am missing",

if_else(x==1,"I am one",

if_else(x==2,"I am two",

if_else(x==3,"I am three","Others")))))
```

Output

```
  x        flag
1  1     I am one
2  2     I am two
3  3   I am three
4  4       Others
5  5       Others
6 NA I am missing
```

SQL-Style CASE WHEN Statement

We can use **case_when()** function to write nested if-else queries. In case_when(), you can use variables directly within case_when() wrapper. **TRUE** refers to ELSE statement.

```
mydf %>% mutate(flag = case_when(is.na(x) ~ "I am missing",
x == 1 ~ "I am one",
x == 2 ~ "I am two",
x == 3 ~ "I am three",
TRUE ~ "Others"))
```

Important Point

Make sure you set is.na() condition at the beginning in nested ifelse. Otherwise, it would not be executed.

Example 39 : Apply ROW WISE Operation

Suppose you want to find maximum value in each row of variables 2012, 2013, 2014, 2015. The **rowwise()** function allows you to apply functions to rows.

```
df = mydata %>%
  rowwise() %>% mutate(Max= max(Y2012,Y2013,Y2014,Y2015)) %>%
  select(Y2012:Y2015,Max)
```

Example 40 : Combine Data Frames

Suppose you are asked to combine two data frames. Let's first create two sample datasets.

```
df1=data.frame(ID = 1:6,  x=letters[1:6])
df2=data.frame(ID = 7:12, x=letters[7:12])
```

df1	
ID	x
1	a
2	b
3	c
4	d
5	e
6	f

df2	
ID	x
7	g
8	h
9	i
10	j
11	k
12	l

Input Datasets

The **bind_rows() function** combine two datasets with rows. So combined dataset would contain **12 rows (6+6) and 2 columns.**

```
xy = bind_rows(df1,df2)
```

It is equivalent to base R function rbind.

```
xy = rbind(df1,df2)
```

The **bind_cols() function** combine two datasets with columns. So combined dataset would contain **4 columns and 6 rows.**

```
xy = bind_cols(x,y)
or
xy = cbind(x,y)
```

The output is shown below-

ID	x	ID	x
1	a	7	g
2	b	8	h
3	c	9	i
4	d	10	j
5	e	11	k
6	f	12	l

cbind Output

Example 41 : Calculate Percentile Values

The **quantile()** function is used to determine Nth percentile value. In this example, we are computing percentile values by variable Index.

```
mydata %>% group_by(Index) %>%
summarise(Pecentile_25=quantile(Y2015, probs=0.25),
Pecentile_50=quantile(Y2015, probs=0.5),
Pecentile_75=quantile(Y2015, probs=0.75),
Pecentile_99=quantile(Y2015, probs=0.99))
```

The **ntile()** function is used to divide the data into N bins.

```
x= data.frame(N= 1:10)
x = mutate(x, pos = ntile(x$N,5))
```

Example 42 : Automate Model Building

This example explains the advanced usage of **do() function**. In this example, we are building linear regression model for each level of a categorical variable. There are 3 levels in variable cyl of dataset mtcars.

```
length(unique(mtcars$cyl))
```

Result : 3

```
by_cyl <- group_by(mtcars, cyl)
models <- by_cyl %>% do(mod = lm(mpg ~ disp, data = .))
summarise(models, rsq = summary(mod)$r.squared)
models %>% do(data.frame(
var = names(coef(.$mod)),
coef(summary(.$mod)))
)
```

rsq
0.64840514
0.01062604
0.27015777

Output : R-Squared Values

if() Family of Functions

It includes functions like select_if, mutate_if, summarise_if. They come into action only when logical condition meets. See examples below.

Example 43 : Select only numeric columns

The **select_if()** function returns only those columns where logical condition is TRUE. The **is.numeric** refers to retain only numeric variables.

```
mydata2 = select_if(mydata, is.numeric)
```

Similarly, you can use the following code for selecting factor columns -

```
mydata3 = select_if(mydata, is.factor)
```

Example 44 : Number of levels in factor variables

Like select_if() function, summarise_if() function lets you to summarise only for variables where logical condition holds.

```
summarise_if(mydata, is.factor, funs(nlevels(.)))
```

It returns 19 levels for variable Index and 51 levels for variable State.

Example 45 : Multiply by 1000 to numeric variables

```
mydata11 = mutate_if(mydata, is.numeric, funs("new" = .* 1000))
```

Example 46 : Convert value to NA

In this example, we are converting "" to NA using **na_if()** function.

```
k <- c("a", "b", "", "d")
na_if(k, "")
```

Result : "a" "b" NA "d"

Example 49 : How to use SQL rank() over(partition by)

In SQL, `rank() over(partition by)` is used to compute rank by a grouping variable. In dplyr, it can be achieved very easily with a single line of code. See the example below. Here we are calculating rank of variable Y2015 by variable Index.

```
t = mydata %>% select(Index, Y2015) %>%
 group_by(Index) %>%
 mutate(rank = min_rank(desc(Y2015)))%>%
 arrange(Index, rank)
```

In dplyr, there are many functions to compute rank other than `min_rank( )`. These are `dense_rank( )`, `row_number( )`, `percent_rank()`.

Endnotes

There are hundreds of packages that are dependent on this package. The main benefit it offers is to take off fear of R programming and make coding effortless and lower processing time. However, some R programmers prefer **data.table** package for its speed. I would recommend learn both the packages. The data.table package wins over dplyr in terms of speed if data size greater than 1 GB.

Lab Session 8

1. mtcars dataset

a. Check to see if you have the mtcars dataset by entering the command
   mtcars.
b. What class is mtcars? class(mtcars)
c. How many observations (rows) and variables (columns) are in the mtcars dataset?
   dim(mtcars)
   nrow(mtcars)
   ncol(mtcars)
d. Copy mtcars into an object called cars and rename mpg in cars to MPG. Use rename().
   cars=mtcars
   cars=rename(cars, MPG=mpg)
e. Convert all the column names of cars to upper case. Use rename_all, and the toupper command (or colnames).
   Mt_upper=rename_all(cars,toupper)
   toupper(colnames(cars))
f. Convert the rownames of cars to a column called car using rownames_to_column. Subset the columns from cars that end in "p" and call it pvars using ends_with().
   cars = tibble::rownames_to_column(mtcars, var = "car")
   head(cars)
   pvars = select(cars, ends_with("p"))
   head(pvars)
g. Create a subset cars that only contains the columns: wt, qsec, and hp and assign this object to carsSub. What are the dimensions of carsSub? (Use select() and dim().)
   carsSub = select(cars, wt, qsec, hp)
   dim(carsSub)
h. Convert the column names of carsSub to all upper case. Use rename_all(), and toupper() (or colnames()).
   carsSub = rename_all(carsSub, toupper)
i. Subset the rows of cars that get more than 20 miles per gallon (mpg) of fuel efficiency. How many are there? (Use filter().)
   cars_mpg = filter(cars, mpg > 20)
   select(cars_mpg,mpg,hp)
j. Subset the rows that get less than 16 miles per gallon (mpg) of fuel efficiency and have more than 100 horsepower (hp). How many are there? (Use filter().)
   nrow(filter(cars, mpg < 16 & hp > 100))
k. Create a subset of the cars data that only contains the columns: wt, qsec, and hp for cars with 8 cylinders (cyl) and reassign this object to carsSub. What are the dimensions of this dataset?
   carsSub = filter(cars, cyl == 8)
   carsSub = select(carsSub, wt, qsec, hp, car)
   dim(carsSub)
l. Re-order the rows of carsSub by weight (wt) in increasing order. (Use arrange().)
   carsSub = arrange(carsSub, wt)
m. Create a new variable in carsSub called wt2, which is equal to wt^2, using mutate() and piping %>%.
   carsSub %>% mutate(wt2 = wt^2)

2. Bike_Lane dataset

bike = read.csv("http://johnmuschelli.com/intro_to_r/data/Bike_Lanes.csv")

bike

1. How many bike "lanes" are currently in Baltimore? You can assume each observation/row is a different bike "lane"

```
dim(bike)
```

2. How many (a) feet and (b) miles of bike "lanes" are currently in Baltimore?

(a) sum(bike$length)

```
sum(bike$length)/5280
```

3. **How many types of bike lanes are there? Which type has (a) the most number of lanes and (b) longest average bike lane length?**

```
colnames(bike)
length(unique(bike$type))  #  n_distinct(bike$type)
m1=bike %>%  group_by(type) %>%  summarise(number_of_rows = n(),
        mean_lane = mean(length)) %>%  arrange(desc(mean_lane))
m1[1,]
filter(m1,number_of_rows==max(number_of_rows)) %>% select(type, number_of_rows)
filter(m1,mean_lane==max(mean_lane)) %>% select(type, mean_lane)
```

4. **How many different projects (project) do the bike lanes fall into? Which project category has the longest average bike lane length?**

```
length(unique(bike$project))
avg = bike %>%  group_by(project) %>%  summarize(mn = mean(length,
na.rm = TRUE)) %>%  filter(mn == max(mn))
avg
```

5. **What was the average bike lane length per year that they were installed? (Be sure to first set `dateInstalled` to NA if it is equal to zero.)**

```
bike = bike %>% mutate(
 dateInstalled = ifelse( dateInstalled == 0, NA, dateInstalled))
print(mean(bike$length[ !is.na(bike$dateInstalled)]))
```

6. (a)Numerically and (b) graphically describe the distribution of bike lane lengths (`length`).

```
# Numeric summary
quantile(bike$length)
hist(bike$length)
boxplot(bike$length~bike$type)
```

Data visualization with ggplot2

ggplot() creates a coordinate system that you can add layers to. The first argument of ggplot() is the dataset to use in the graph. You complete your graph by adding one or more layers to ggplot(). The function geom_point() adds a layer of points to your plot, which creates a scatterplot. The mapping argument is always paired with aes(), and the x and y arguments of aes() specify which variables to map to the x and y-axes.

```
ggplot(data = mpg) +
geom_point(mapping = aes(x = displ, y = hwy)) #geom_point() adds
 a layer of points to the plot which creates a scatterplot
```

The general plotting functions of `ggplot2` is `ggplot` and is very powerful using the **g**rammar of **g**raphics. When creating a plot, there are two essential attributes of the plot you need to specify:
aesthetics and geoms
Aesthetics are mappings between the variables in the data and visual properties in the plots. Aesthetics are set in the `aes()` function and the most common aesthetics are

- **x**
- y
- color/colour
- size

- fill
- shape
- linetype
- group

If you set these in `aes`, then you set them to a variable. If you want to set them for all values, set them in a `geom`.
The other essential element of a `ggplot` is a `geom` layer to determine how the data will be plotted.

- `geom_point` - **add points**
- `geom_line` - **add lines**
- `geom_density` - **add density plot**
- `geom_histogram` - **add a histogram**
- `geom_smooth` - **add a smoother**
- `geom_boxplot` - **add a boxplot**
- `geom_bar` - **add a bar chart**
- `geom_tile` - **rectangles/heatmaps**

You add these layer with `+` sign. If you assign a plot to an object, you must call `print` to display it (this is the same a submitting the name of the object to the console).

```
install.packages("tidyverse")

library(tidyverse)

g=mpg%>%ggplot(aes(x = displ, y = hwy))

g

g+geom_line()

gg <- g + geom_line() +
  labs(x = "displacement", y = "hwy", title = "disp vs hwy" )
gg
s1=mpg %>% filter(year %in% 1999)
s1
g = s1 %>% ggplot(aes(x = displ, y = cyl, group = year))
g + geom_line()

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```

```
ggplot(data = mpg) +
  geom_line(mapping = aes(x = displ, y = hwy))
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
ggplot(data = mpg) +
  geom_boxplot(mapping = aes(x = displ, y = hwy))
```

```
qplot(x, y=NULL, data, geom="auto", xlim = c(NA, NA), ylim =c(NA, NA))
```

- **x** : x values
- **y** : y values (optional)
- **data** : data frame to use (optional).
- **geom** : Character vector specifying geom to use. Defaults to “point” if x and y are specified, and “histogram” if only x is specified.
- **xlim**, **ylim**: x and y axis limits

Other arguments including *main*, *xlab*, *ylab* and *log* can be used also:

- **main**: Plot title
- **xlab, ylab**: x and y axis labels

```
# Use data from numeric vectors
x <- 1:10
 y = x*x
# Basic plot qplot(x,y)
# Add line qplot(x, y, geom=c("point", "line"))
#  Use  data  from  a  data  frame  qplot(mpg,  wt, data=mtcars)
# Smoothing
```

qplot(mpg, wt, data = mtcars, geom = c("point", "smooth"))

Smoothed line by groups

The argument **color** is used to tell **R** that we want to color the points by groups:

```
# Linear fits by group
qplot(mpg, wt, data = mtcars, color = factor(cyl),geom=c("point", "smooth"))
```

```
qplot(mpg, data=mtcars, geom="histogram",xlab="miles per gallon",ylab="count", main="histogram")
```

Change the shape and the size of points

Like color, the **shape** and the **size** of points can be controlled by a continuous or discrete variable.

```
# Change the size of points according to
  # the values of a continuous variable
qplot(mpg, wt, data = mtcars, size = mpg)
# Change point shapes by groups
qplot(mpg, wt, data = mtcars, shape = factor(cyl))
```

Lab Sheet-9

Statistical Analysis

Correlation is a statistical measure that expresses the extent to which two variables are linearly related (meaning they change together at a constant rate). We describe correlations with a unit-free measure called the correlation coefficient which ranges from -1 to +1 and is denoted by *r*

- The closer *r* is to zero, the weaker the linear relationship.
- Positive *r* values indicate a positive correlation, where the values of both variables tend to increase together.
- Negative *r* values indicate a negative correlation, where the values of one variable tend to increase when the values of the other variable decrease.

s1=select(mtcars, gear, carb)

s1

cor(s1)

```
cor(s1, use = "complete.obs")
```

Proportion tests

`prop.test()` can be used for testing the null that the proportions (probabilities of success) in several groups are the same, or that they equal certain given values.

```
prop.test(c(15,25),c(100,100))
```

```
prop.test(c(45,66),c(100,110))
```

```
Null hypothesis: The two proportions are statistically same
```

```
Alternative Hypothesis: the two proportions are statistically different
```

```
If p value<0.05, reject Null Hypothesis
```

Chi-Square test is a statistical method to determine if two categorical variables have a significant correlation between them. Both those variables should be from same population and they should be categorical like − Yes/No, Male/Female, Red/Green etc.

Syntax

The function used for performing chi-Square test is **chisq.test()**.

The basic syntax for creating a chi-square test in R is −

chisq.test(data)

Following is the description of the parameters used −

- **data** is the data in form of a table containing the count value of the variables in the observation.

Example

We will take the Cars93 data in the "MASS" library which represents the sales of different models of car in the year 1993.

For our model we will consider the variables "AirBags" and "Type". Here we aim to find out any significant correlation between the types of car sold and the type of Air bags it has. If correlation is observed we can estimate which types of cars can sell better with what types of air bags.

```
# Load the library.
library("MASS")

# Create a data frame from the main data set.
car.data <- data.frame(Cars93$AirBags, Cars93$Type)

# Create a table with the needed variables.
cardata = table(Cars93$AirBags, Cars93$Type)
print(cardata)

# Perform the Chi-Square test.
print(chisq.test(cardata))
```

When we execute the above code, it produces the following result −

```
                  Compact Large Midsize Small Sporty Van
Driver & Passenger       2     4       7     0      3   0
Driver only              9     7      11     5      8   3
None                     5     0       4    16      3   6
```

Pearson's Chi-squared test

data: car.data
X-squared = 33.001, df = 10, p-value = 0.0002723

Warning message:
In chisq.test(car.data) : Chi-squared approximation may be incorrect

Conclusion:

Null Hypothesis: the variables are independent

Alternative Hypothesis: the variables are dependent

P<0.05: reject null hypothesis and accept A.H

The result shows the p-value of less than 0.05 which indicates a strong correlation.

Fisher's Exact Test

Independence tests are used to determine if there is a significant relationship between two categorical variables. There exists two different types of independence test:

- the Chi-square test (the most common)
- the Fisher's exact test

On the one hand, the Chi-square test is used when the sample is large enough. On the other hand, the Fisher's exact test is used when the sample is small

Hypotheses

The hypotheses of the Fisher's exact test are the same than for the Chi-square test, that is:

- H_0 : the variables are independent, there is **no** relationship between the two categorical variables. Knowing the value of one variable does not help to predict the value of the other variable
- H_1 : the variables are dependent, there is a relationship between the two categorical variables. Knowing the value of one variable helps to predict the value of the other variable

Example

Data

For our example, we want to determine whether there is a statistically significant association between smoking and being a professional athlete. Smoking can only be “yes” or “no” and being a professional athlete can only be “yes” or “no”. The two variables of interest are qualitative variables and we collected data on 14 persons.

Observed frequencies

Our data are summarized in the contingency table below reporting the number of people in each subgroup:

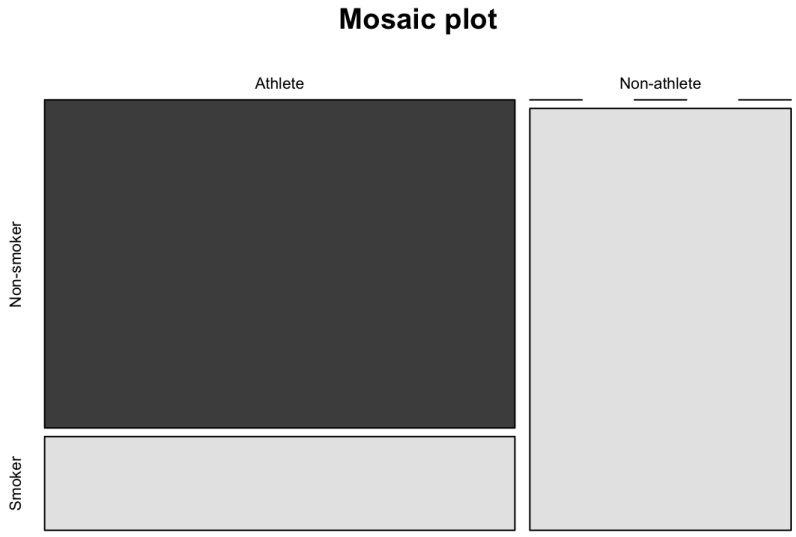
```
dat <- data.frame(
  "smoke_no" = c(7, 0),
  "smoke_yes" = c(2, 5),
  row.names = c("Athlete", "Non-athlete"),
  stringsAsFactors = FALSE
)
colnames(dat) <- c("Non-smoker", "Smoker")
```

```
dat
```

```
##             Non-smoker Smoker
## Athlete              7      2
## Non-athlete          0      5
```

It is also a good practice to draw a mosaic plot to visually represent the data:

```
mosaicplot(dat,
  main = "Mosaic plot",
  color = TRUE
)
```



We can already see from the plot that the proportion of smokers in the sample is higher among non-athletes than athlete. The plot is however not sufficient to conclude that there is such a significant association in the population.

Expected frequencies

Remember that the Fisher's exact test is used when there is at least one cell in the contingency table of the expected frequencies below 5. To retrieve the expected frequencies, use the `chisq.test()` function together with `$expected`:

```
chisq.test(dat)$expected
## Warning in chisq.test(dat): Chi-squared approximation may be incorrect
##             Non-smoker Smoker
## Athlete            4.5    4.5
## Non-athlete        2.5    2.5
```

The contingency table above confirms that we should use the Fisher's exact test instead of the Chi-square test because there is at least one cell below 5.

Fisher's exact test in R

To perform the Fisher's exact test in R, use the `fisher.test()` function as you would do for the Chi-square test:

```
test <- fisher.test(dat)
test
```

```
##  Fisher's Exact Test for Count Data
##
## data:  dat
## p-value = 0.02098
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  1.449481      Inf
## sample estimates:
## odds ratio
##        Inf
```

The most important in the output is the p-value. You can also retrieve the p-value with:

```
test$p.value
## [1] 0.02097902
```

Note that if your data is not already presented as a contingency table, you can simply use the following code:

```
fisher.test(table(dat$variable1, dat$variable2))
```

where `dat` is the name of your dataset, `variable1` and `variable2` correspond to the names of the two variables of interest.

Conclusion and interpretation

From the output and from `test$p.value` we see that the p-value is less than the significance level of 5%. Like any other statistical test, if the p-value is less than the significance level, we can reject the null hypothesis.

⇒⇒ In our context, rejecting the null hypothesis for the Fisher's exact test of independence means that there is a significant relationship between the two categorical variables (smoking habits and being an athlete or not). Therefore, knowing the value of one variable helps to predict the value of the other variable.

ANOVA Test

ANOVA also known as Analysis of variance is used to investigate relations between categorical variables and continuous variable in R Programming. It is a type of hypothesis testing for population variance.

R – ANOVA Test

ANOVA test involves setting up:

Null Hypothesis: All population means are equal.

Alternate Hypothesis: Atleast one population mean is different from other.

ANOVA tests are of two types:

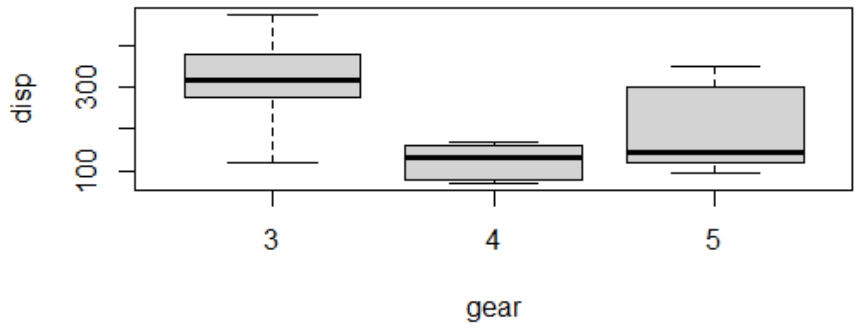
One way ANOVA: It takes one categorical group into consideration.

Two way ANOVA: It takes two categorical group into consideration.

Performing One Way ANOVA test

```
# Installing the package
install.packages("dplyr")

# Loading the package
library(dplyr)

# Variance in mean within group and between group
boxplot(mtcars$disp~factor(mtcars$gear),xlab = "gear", ylab =
"disp")

# Step 1: Setup Null Hypothesis and Alternate Hypothesis
# H0 = mu = mu01 = mu02(There is no difference between average
displacement for different gear)
# H1 = Not all means are equal
```

```
# Step 2: Calculate test statistics using aov function
mtcars_aov <-
aov(mtcars$disp~factor(mtcars$gear))
summary(mtcars_aov)

# Step 3: Calculate F-Critical Value
# For 0.05 Significant value, critical value = alpha = 0.05

# Step 4: Compare test statistics with F-Critical value
# and conclude test p < alpha, Reject Null Hypothesis
```



The box plot shows the mean values of gear with respect of displacement. Here, categorical variable is gear on which factor function is used and continuous variable is disp.

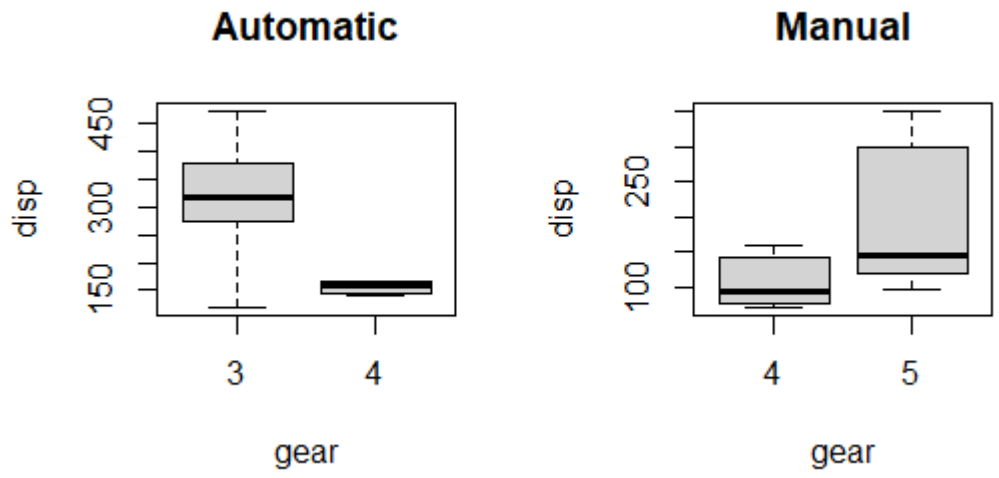
The summary shows that the gear attribute is very significant to displacement(Three stars denoting it). Also, the P value is less than 0.05, so proves that gear is significant to displacement i.e related to each other and we reject the Null Hypothesis.

Performing Two Way ANOVA test in R

Two-way ANOVA test is performed using mtcars dataset which comes preinstalled with dplyr package between disp attribute, a continuous attribute and gear attribute, a categorical attribute, am attribute, a categorical attribute.

```
# Installing the package
install.packages("dplyr")

# Loading the package
library(dplyr)
```

```
# Variance in mean within group and between group
boxplot(mtcars$disp~mtcars$gear, subset = (mtcars$am == 0),
        xlab = "gear", ylab = "disp", main = "Automatic")
boxplot(mtcars$disp~mtcars$gear, subset = (mtcars$am == 1),
            xlab = "gear", ylab = "disp", main = "Manual")

# Step 1: Setup Null Hypothesis and Alternate Hypothesis
# H0 = mu0 = mu01 = mu02(There is no difference between
# average displacement for different gear)
# H1 = Not all means are equal

# Step 2: Calculate test statistics using aov function
mtcars_aov2 <- aov(mtcars$disp~factor(mtcars$gear) *
                                factor(mtcars$am))
summary(mtcars_aov2)

# Step 3: Calculate F-Critical Value
# For 0.05 Significant value, critical value = alpha = 0.05

# Step 4: Compare test statistics with F-Critical value
# and conclude test p < alpha, Reject Null Hypothesis
```



The box plot shows the mean values of gear with respect to displacement. Here, categorical variables are gear and am on which factor function is used and continuous variable is disp.

The summary shows that the gear attribute is very significant to displacement (Three stars denoting it) and am attribute is not much significant to displacement. P-value of gear is less than 0.05, so it proves that gear is significant to displacement i.e related to each other. P-value of am is greater than 0.05, am is not significant to displacement i.e not related to each other.

Results

We see significant results from boxplots and summaries.

- Displacement is strongly related to Gears in cars i.e displacement is dependent on gears with $p < 0.05$.
- Displacement is strongly related to Gears but not related to transmission mode in cars with p 0.05 with am.

Lab Sheet-10

Linear Regression

Simple linear regression is a statistical method you can use to understand the relationship between two variables, x and y.

One variable, x, is known as the predictor variable.

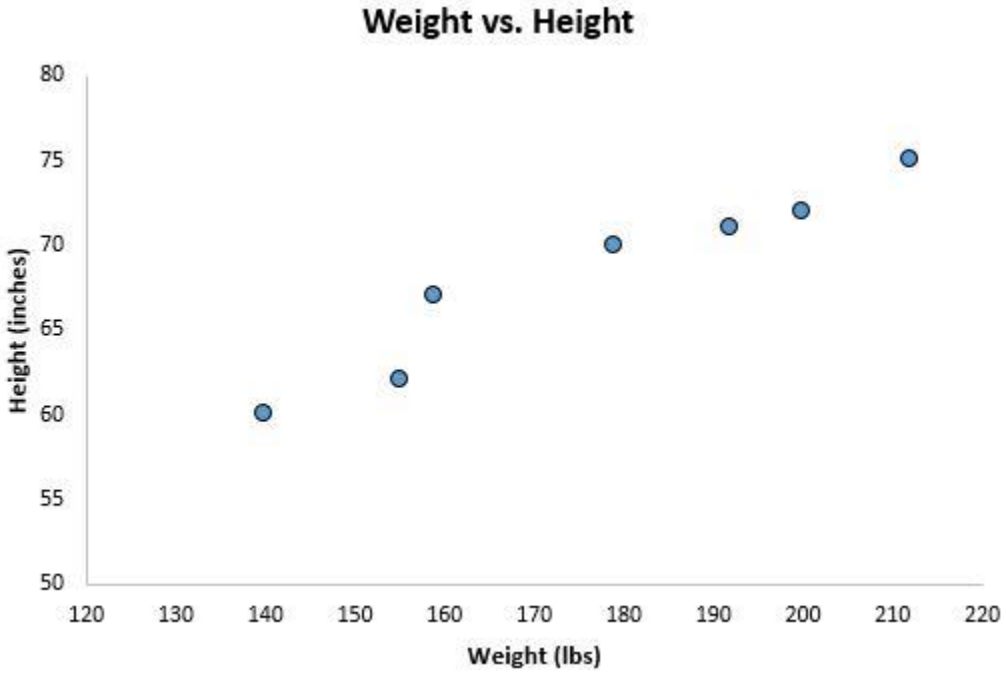
The other variable, y, is known as the response variable.

For example, suppose we have the following dataset with the weight and height of seven individuals:

Weight (lbs)	Height (inches)
140	60
155	62
159	67
179	70
192	71
200	72
212	75

Let weight be the predictor variable and let height be the response variable.

If we graph these two variables using a scatterplot, with weight on the x-axis and height on the y-axis, here's what it would look like:



Suppose we're interested in understanding the relationship between weight and height. From the scatterplot we can clearly see that as weight increases, height tends to increase as well, but to actually quantify this relationship between weight and height, we need to use linear regression.

Using linear regression, we can find the line that best "fits" our data. This line is known as the least squares regression line and it can be used to help us understand the relationships between weight and height.

The formula for the line of best fit is written as:

$$\hat{y} = b0 + b1x$$

y=mx+c

where $\hat{y}$ is the predicted value of the response variable, b0 is the y-intercept, b1 is the regression coefficient, and x is the value of the predictor variable.

How to Interpret a Least Squares Regression Line

Here is how to interpret this least squares regression line: $\hat{y} = 32.7830 + 0.2001x$

b0 = 32.7830. This means when the predictor variable weight is zero pounds, the predicted height is 32.7830 inches. Sometimes the value for b0 can be useful to know, but in this specific example it doesn't actually make sense to interpret b0 since a person can't weight zero pounds.

b1 = 0.2001. This means that a one unit increase in x is associated with a 0.2001 unit increase in y. In this case, a one pound increase in weight is associated with a 0.2001 inch increase in height.

How to Use the Least Squares Regression Line

Using this least squares regression line, we can answer questions like:

For a person who weighs 170 pounds, how tall would we expect them to be?

To answer this, we can simply plug in 170 into our regression line for x and solve for y:

$\hat{y} = 32.7830 + 0.2001(170) = 66.8$ inches

For a person who weighs 150 pounds, how tall would we expect them to be?

To answer this, we can plug in 150 into our regression line for x and solve for y:

$\hat{y} = 32.7830 + 0.2001(150) = 62.798$ inches

The Coefficient of Determination(R-squared)

One way to measure how well the least squares regression line "fits" the data is using the coefficient of determination, denoted as R2.

The coefficient of determination is the proportion of the variance in the response variable that can be explained by the predictor variable.

The coefficient of determination can range from 0 to 1. A value of 0 indicates that the response variable cannot be explained by the predictor variable at all. A value of 1 indicates that the response variable can be perfectly explained without error by the predictor variable.

An R2 between 0 and 1 indicates just how well the response variable can be explained by the predictor variable. For example, an R2 of 0.2 indicates that 20% of the variance in the response variable can be explained by the predictor variable; an R2 of 0.77 indicates that 77% of the variance in the response variable can be explained by the predictor variable.

Notice in our output from earlier we got an R2 of 0.9311, which indicates that 93.11% of the variability in height can be explained by the predictor variable of weight:

Coefficient of determination in linear regression

This tells us that weight is a very good predictor of height.

Assumptions of Linear Regression

For the results of a linear regression model to be valid and reliable, we need to check that the following four assumptions are met:

1. Linear relationship: There exists a linear relationship between the independent variable, x, and the dependent variable, y.

2. Independence: The residuals are independent. In particular, there is no correlation between consecutive residuals in time series data.

3. Homoscedasticity: The residuals have constant variance at every level of x.

4. Normality: The residuals of the model are normally distributed.

If one or more of these assumptions are violated, then the results of our linear regression may be unreliable or even misleading.

Y=mx+c

```
# x represents the weight of the people
x <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

#y represents the height of the people
y<-c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
```

```
#Apply the lm() function to find the relation between
the height and weight of the people
relation <- lm(y~x)  # output variable ~ input variable
print(relation)
#Get the summary of the Relationship
print(summary(relation))
```

```
#predict the height of new person with weight 70.
```

```
a <- data.frame(x = 70)
result <- predict(relation, a)
print(result)      # y=1.414*x+61.3    # y=160.4
```

```
#visualize the Regression Graphically
png(file = "linearregression.png")
plot(y,x,col="blue",main="Height and Weight
Regression")
abline(lm(y ~ x),cex = 1.3, pch = 16, xlab = "Weight in
Kg", ylab = "Height in cm")

#Save the file.
dev.off()
```

Example: Predict the age of a person with height 78.5

Age	height
18	76.1
19	77.7
20	78.1
21	78.2
22	78.8
23	79.7

Logistic Regression

When we want to understand the relationship between one or more predictor variables and a continuous response variable, we often use linear regression.

However, when the response variable is categorical we can instead use logistic regression.

Logistic regression is a type of classification algorithm because it attempts to "classify" observations from a dataset into distinct categories.

Here are a few examples of when we might use logistic regression:

We want to use credit score and bank balance to predict whether or not a given customer will default on a loan. (Response variable = "Default" or "No default")
We want to use average rebounds per game and average points per game to predict whether or not a given basketball player will get drafted into the NBA (Response variable = "Drafted" or "Not Drafted")
We want to use square footage and number of bathrooms to predict whether or not a house in a certain city will be listed at a selling price of $200k or more. (Response variable = "Yes" or "No")
Notice that the response variable in each of these examples can only take on one of two values. Contrast this with linear regression in which the response variable takes on some continuous value.

Assumptions of Logistic Regression
Logistic regression uses the following assumptions:

1. The response variable is binary. It is assumed that the response variable can only take on two possible outcomes.

2. The observations are independent. It is assumed that the observations in the dataset are independent of each other. That is, the observations should not come

from repeated measurements of the same individual or be related to each other in any way.

3. There is no severe multicollinearity among predictor variables. It is assumed that none of the predictor variables are highly correlated with each other.

4. There are no extreme outliers. It is assumed that there are no extreme outliers or influential observations in the dataset.

5. There is a linear relationship between the predictor variables and the logit of the response variable. This assumption can be tested using a Box-Tidwell test.

The Logistic Regression Equation
Logistic regression uses a method known as maximum likelihood estimation to find an equation of the following form:

$$\log[p(X) / (1-p(X))] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

where:

X_j: The jth predictor variable
β_j: The coefficient estimate for the jth predictor variable
The formula on the right side of the equation predicts the log odds of the response variable taking on a value of 1.

Thus, when we fit a logistic regression model we can use the following equation to calculate the probability that a given observation takes on a value of 1:

$$\mathbf{p(X) = e(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p) / (1 + e(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p))}$$

We then use some probability threshold to classify the observation as either 1 or 0.

For example, we might say that observations with a probability greater than or equal to 0.5 will be classified as “1” and all other observations will be classified as “0.”

How to Interpret Logistic Regression Output

Suppose we use a logistic regression model to predict whether or not a given basketball player will get drafted into the NBA based on their average rebounds per game and average points per game.

Here is the output for the logistic regression model:

Interpret logistic regression output

Using the coefficients, we can compute the probability that any given player will get drafted into the NBA based on their average rebounds and points per game using the following formula:

P(Drafted) = e-2.8690 + 0.0698*(rebs) + 0.1694*(points) / (1+e-2.8690 + 0.0698*(rebs) + 0.1694*(points))

For example, suppose a given player averages 8 rebounds per game and 15 points per game. According to the model, the probability that this player will get drafted into the NBA is 0.557.

P(Drafted) = e-2.8690 + 0.0698*(8) + 0.1694*(15) / (1+e-2.8690 + 0.0698*(8) + 0.1694*(15)) = 0.557

Since this probability is greater than 0.5, we would predict that this player will get drafted.

Contrast this with a player who only averages 3 rebounds and 7 points per game. The probability that this player will get drafted into the NBA is 0.186.

P(Drafted) = e-2.8690 + 0.0698*(3) + 0.1694*(7) / (1+e-2.8690 + 0.0698*(3) + 0.1694*(7)) = 0.186

Since this probability is less than 0.5, we would predict that this player will not get drafted.

6. The sample size is sufficiently large.

The function used to create the regression model is the **glm()** function.

Syntax

The basic syntax for **glm()** function in logistic regression is −

glm(formula,data,family)

Following is the description of the parameters used −

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is binomial for logistic regression.

Example

The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

```
# Select some columns form mtcars.
input <- mtcars[,c("am","cyl","hp","wt")]

print(head(input))
```

Create Regression Model

We use the **glm()** function to create the regression model and get its summary for analysis.

```
input <- mtcars[,c("am","cyl","hp","wt")]

amdata = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)

print(summary(amdata))
```

Conclusion

In the summary as the p-value in the last column is more than 0.05 for the variables "cyl" and "hp", we consider them to be insignificant in contributing to the value of the variable "am". Only weight (wt) impacts the "am" value in this regression model.

Example 2:

```
# Installing the package
install.packages("caTools")    # For Logistic regression
install.packages("ROCR")       # For ROC curve to evaluate model

# Loading package
library(caTools)
library(ROCR)

# Splitting dataset
split <- sample.split(mtcars, SplitRatio = 0.8)
split

train_reg <- subset(mtcars, split == "TRUE")
test_reg <- subset(mtcars, split == "FALSE")

# Training model
logistic_model <- glm(vs ~ wt + disp,
                      data = train_reg,
                      family = "binomial")
logistic_model

# Summary
summary(logistic_model)

# Predict test data based on model
predict_reg <- predict(logistic_model,
                       test_reg, type = "response")
predict_reg

# Changing probabilities
predict_reg <- ifelse(predict_reg >0.5, 1, 0)
```

```
# Evaluating model accuracy
# using confusion matrix
table(test_reg$vs, predict_reg)
err <- mean(predict_reg != test_reg$vs)
print(paste('Accuracy =', 1 - err))
 # ROC-AUC Curve
ROCPred <- prediction(predict_reg, test_reg$vs)
ROCPer <- performance(ROCPred, measure = "tpr",
                              x.measure = "fpr")
auc <- performance(ROCPred, measure = "auc")
auc <- auc@y.values[[1]]
auc


# Plotting curve


plot(ROCPer)
```

wt influences dependent variables positively and one unit increase in wt increases the log of odds for vs =1 by 1.44. **disp** influences dependent variables negatively and one unit increase in disp decreases the log of odds for vs =1 by 0.0344. Null deviance is 31.755(fit dependent variable with intercept) and Residual deviance is 14.457(fit dependent variable with all independent variable). AIC(Alkaline Information criteria) value is 20.457 i.e the lesser the better for the model. Accuracy comes out to be 0.75 i.e 75%.

Simulations:

Simulations are a powerful statistical tool. Simulation techniques allow us to carry out statistical inference in complex models, estimate quantities that we can cannot calculate analytically.

Standard Probability Distributions

Sometimes you want to generate data from a distribution (such as normal), or want to see where a value falls in a known distribution. `R` has these distributions built in:

- Normal
- Binomial
- Beta
- Exponential
- Gamma

- Hypergeometric

Each family of functions for a distribution has 4 options:

- `r` for random number generation [e.g. `rnorm()`]
- `d` for density [e.g. `dnorm()`]
- `p` for probability [e.g. `pnorm()`]
- `q` for quantile [e.g. `qnorm()`]

For example, we can simulate a random sample of size 5 from a standard normal distribution by using `rnorm`.

```
rnorm(5)
[1]  0.81199910  1.32466134 -0.05470965  0.38102517 -1.83418402
```

rnorm(5, mean=10,sd=5)

To find the probability of being less than 4 in a Normal distribution with mean 5 and standard deviation 2, we would use `pnorm`.

```
pnorm(4, mean = 5, sd = 2)
[1] 0.3085375
```

To find the 97.5 percentile in a standard normal distribution (i.e the number z such that 97.5% of the probability is to the left of z in a standard normal distribution), we would use `qnorm`.

```
qnorm(0.975)
[1] 1.959964
```

Acceptance-Rejection Technique to Generate Random Variate

- Example: use following steps to generate uniformly distributed random numbers between 1/4 and 1.

  Step 1.
  Generate a random number R
  Step 2a.
  If $R \geq 1/4$, accept $X = R$, goto Step 3

Step 2b.

If $R < 1/4$, reject *R*, return to Step 1

Step 3.

If another uniform random variate on [1/4, 1] is needed, repeat the procedure begining at Step 1. Otherwise stop.

- Do we know if the random variate generated using above methods is indeed uniformly distributed over [1/4, 1]? The answer is Yes.