

Module 1: Introduction and Learning R

Fundamentals of Genomic Prediction and Data-Drive Crop Breeding

(August 4-8, 2025)



Waseem Hussain

Senior Scientist-I
International Rice Research Institute
Rice Breeding Innovations Platform
waseem.hussain@cgiar.org
whussain2.github.io

Mahender Anumalla

Scientist-I
International Rice Research Institute
South-Asian Hub, Hyderabad
m.anumalla@cgiar.org

August 2, 2025

Contents

Section 1 : Overview of R and R Studio	2
What is R Software	2
What is R Studio	2
Installation of R and R Studio	2
R Package and Installation	3
Section 2: R Essentials	4
Function Calls	4
Operators in R	5
Data Types in R	6
Data Structures in R	7
Scalars	8
Vectors	8
Matrices	9
Data Frames	12
Lists	14
Section 3: Control Structures in R	15
If statement	15
If and else statement	16
Else If statement	16
Multiple Condition Statements	17
ifelse Function	18
Apply Function	19
lapply function	20
sapply Function	21
tapply Function	22
Loops in R	22
for loop	22
while loop	23
repeat loop	23
Section 4: String Manipulations	24
Basic String Manipulations	24
Set Operations	26
Regular Expressions	27
Section 5: Importing and Exporting Data	29
Set Working Directory	29
Import and Export .csv Files	30
Import and Export .txt Files	30
Import and Export .xlsx Files	30
Import from Web	31
Import and Export .rds	31
Section 6: Data Wrangling and Manipulations	31
Section 7: Basic and Advanced Graphics	34
Basic Graphics with R	34
Scatter Plot	34
Barplot	35
Histograms	36
Boxplots	37

Advanced Graphics with ggplot2	38
SCATTER PLOT	38
BAR PLOT	42
HISTOGRAMS	43
Section 8: Writting Own Functions	44
Section 9: R Markdown	46

Section 1 : Overview of R and R Studio

What is R Software

- R is a free open-source software and programming language.
 - Summarize, explore and model the data
- Reproducible research (code +text).
- Huge learning Resources and Community [Resource 1](#); [Resource 2](#); [Resource 3](#); and [Resource 4](#)
- Popular graphical capabilities.
- Dominant and useful variety of scientific disciplines.

What is R Studio

RStudio is an integrated development environment (IDE) for R

- Easy to control and manage the R scripts (point and click)
 - View and interact with the objects in single environment.
 - Easy to set your working directory and access files on your computer
 - Graphics more accessible.
 - More features see the link: [Click here](#)
-

Installation of R and R Studio

To download **R software** [Click here](#) and to download **R Studio** [Click here](#). Go over these resources to know how to download and install R and R Studio; [Link 1](#); [Link 2](#); [Link 3](#)

R Package and Installation



What is R package?

- Bundles of codes build by the people to perform certain tasks
- Maintained at Comprehensive R Archive Network (CRAN)/Bioconductor/GitHub

Install from CRAN

```
install.package("ggplot2")
```

```
install.packages("ggplot2")
```

Resources CRAN: [Link 1](#); [Link 2](#); and [Link 3](#)

Install from Bioconductor [Click Here](#)

- Packages for life sciences related data
- BiocManager handles all of the packages hosted on Bioconductor

```
install.packages("BiocManager")
```

```
BiocManager::install("GWASTools")
```

```
# Installs BiocManager if not installed
if (!require("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install(version = "3.20")
# Now Install R Package
BiocManager::install("GWASTools")
```

Resources Bioconductor: [Link 1](#); [Link 2](#); [Link 3](#); [Link 4](#)

Install from GitHub [Click Here](#)

```
install.packages("devtools")
```

```
devtools::install_github("tidyr")
```

```
# First Install Devtools
install.packages("devtools")
# Then Install Package tidyr
devtools::install_github("tidyr")
```

Resources GitHub: [Link 1](#); [Link 2](#); and [Link 3](#)

Section 2: R Essentials

- R works on expression and objects. It is like a big gigantic calculator
- For example, Users enters expression (for example 2+2)
 - Expression involves operators or function calls.
 - Expression work on Objects
- R evaluates it
- And Print the Results , 4

```
# Examples of R Expression
# Add
2+2
```

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

```
# Subtract
4-2
```

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

```
# Square
2^2
```

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

```
# Division
10/2
```

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

```
# Multiple
5*5
```

```
## [1] 25
```

```
# Log
log(10)
```

```
## [1] 2.302585
```

```
# Exponential
exp(10)
```

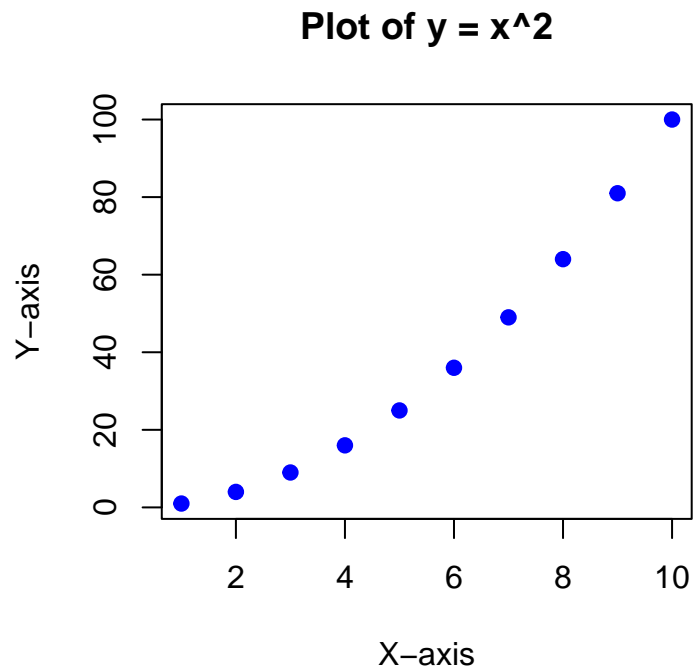
```
## [1] 22026.47
```

Function Calls

- Calling a function which involves one or more variables.
 - For example *sum(x)* or *plot(x)* are function call.
 - It is like expression or object.
- Function format is followed by a set of parentheses containing one or more arguments. *function(argument1, argument2,...)*. For example: *plot(height, weight)* here height and weight are the arguments for the function call *plot()*
- See more details as: *plot(height, weight, pch=2, color="red".....)*
- Positional matching; define the position of arguments *plot(x=height, y=weight, pch=2...)*

```
# Create sample data
x <- seq(1, 10, by=1) # x values from 1 to 10
y <- x^2 # y values are the squares of x
```

```
# Plot the data
plot(x, y,
     main="Plot of  $y = x^2$ ", # Title of the plot
     xlab="X-axis",           # Label for the x-axis
     ylab="Y-axis",           # Label for the y-axis
     col="blue",              # Color of the points
     pch=19)
```



```
# Question: How many Arguments are in function plot()?
# Can you figure positional matching
```

Operators in R

- 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.
- In R we have **Arithmetic Operators**, **Relational Operators**, and **Logical Operators**. Find more on Operators in R [Click Here](#)

Arithmetic Operators in R	
Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%%/%	Integer Division

Logical Operators in R	
Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
	Element-wise logical OR
	Logical OR

Relational Operators in R	
Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

```
# RATIONAL OPERATORS (Returns TRUE or FALSE)
# Is 3 > 4
3>4
```

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

```
# Is 5 > 4
5>4
```

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

```
# Is 5==5
5==5
```

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

```
# Is 2 not equal to 3
2!=3
```

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

```
# LOGICAL OPERATORS
vector <- c(1, 2, 3, 4, 5) # Creating a simple vector
# Check which elements are greater than 2 and less than 5
result <- vector[vector > 2 & vector < 5] # result will be c(3, 4)
result
```

```
## [1] 3 4
```

Data Types in R

- Six data types are in R:

Character: “Block”, “Replication”

Numeric (real or decimal)- 2.4, 2, 10

Integer: 2L, 3L

Logical: TRUE, FALSE

Complex: 1+6i

- In R we use these function call to check type of data

class(): what kind of object is it

length(): how long it is

```
# NUMERIC TYPE
num1 <- 5      # Integer
num2 <- 3.14   # Decimal (floating-point)
num2
```

```
## [1] 3.14
```

```
# INTEGER TYPE
int_num <- 10L # Integer type
int_num
```

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

```
# CHARACTER TYPE
char <- "Hello, R!" # Character string
char
```

```
## [1] "Hello, R!"
```

```
# LOGICAL
True_type <- TRUE
False_type <- FALSE
False_type
```

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

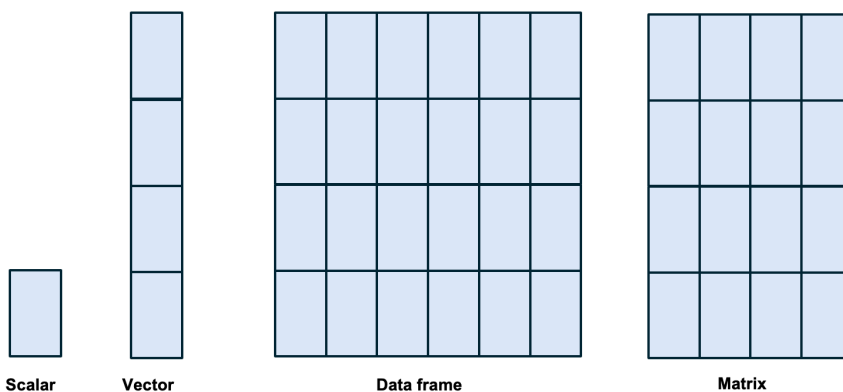
```
# COMPLEX
complex_num <- 3 + 2i # 3 is the real part, and 2 is the imaginary part
complex_num
```

```
## [1] 3+2i
```

#Assignment operator in R is <-, why not Equal sign =
#Question: Could you use class() function to check the type of data

Data Structures in R

- In R we have following data structures:



Scalars

- A scalar object is just a single value like a number or a name.
- For example, `a <- 100` `X<-“name”`
- Scalars don't have to be numeric, they can also be characters (also known as strings)

```
# Examples of numeric scalars
```

```
a <- 100  
a
```

```
## [1] 100
```

```
b <- 3 / 100  
b
```

```
## [1] 0.03
```

```
c <- (a + b) / b  
c
```

```
## [1] 3334.333
```

```
# Examples of character scalars
```

```
d <- "ship"  
d
```

```
## [1] "ship"
```

```
e <- "cannon"  
e
```

```
## [1] "cannon"
```

```
f <- "Do any modern armies still use cannons?"  
f
```

```
## [1] "Do any modern armies still use cannons?"
```

```
# Could you add two scalars of type character?
```

Vectors

- Vector is a basic data structure in R which contains a list of same elements
- Vectors are created using function `c()`, concatenate the elements

```
# Example of Vector
```

```
X<-c(1,2,3,4,5, 6) # five components  
X
```

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

```
# Check length
```

```
length(X)
```

```
## [1] 6
```

```
# Creating a vector of 2s eight times
```

```
d<-rep(2, 8)  
d
```

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

```

d<-rep(c(1,2,3,4), 5)
d

## [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4

d<-rep(c(1:4), 2)
d

## [1] 1 2 3 4 1 2 3 4
#INDEXING or SUBSETTING

x[1] # Extract first element

## [1] 1

x[c(1,2, 6)] # Extract first, second and 6 element in vector x (sub vector)

## [1] 1 2 6

x[1:3] # Extracts elemnts from 1 to thrid position

## [1] 1 2 3

x[-6] # Drops last element

## [1] 1 2 3 4 5 7 8 9 10

sum(x) # sum function adds all elements

## [1] 55

mean(x) # get mean of x

## [1] 5.5

min(x) # get minimal value

## [1] 1

max(x) # Get maximum value

## [1] 10

```

Matrices

- Matrices are **numeric** array of rows and columns.
- Think as Stacked version of vectors where each row and column is basically a vector; Combination of n vectors
- We use function *matrix()*, to create a matrix in R

```

# Creat a matrix
## 2.3.1 First approach
m2<-matrix(1:9, nrow = 3, ncol = 3) # Matrix with 3 rows and columns
m2 # Matrix filled by rows

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

```

```

m2<-matrix(1:9, nrow=3, ncol=3, byrow = TRUE)
# Assign names to rows and columns
dimnames(m2)<-list(c("X","Y","Z"), c("A","B","C"))
m2

##   A B C
## X 1 2 3
## Y 4 5 6
## Z 7 8 9

# Access and change the row and column names
colnames(m2) # Get column names

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

row.names(m2)

## [1] "X" "Y" "Z"

# Change the names of columns
colnames(m2)<-c("A.1", "B.1", "C.1")
m2

##   A.1 B.1 C.1
## X   1   2   3
## Y   4   5   6
## Z   7   8   9

# Change the name of just first column
colnames(m2)[1]<-"B.B"
m2

##   B.B B.1 C.1
## X   1   2   3
## Y   4   5   6
## Z   7   8   9

## Second approach to create the matrix

# We can use cbind() and rbind() functions, column and row bind
x<-c(4,2,3,6) # create a vector of x
y<-c(8,5,6,9) # create a vector of y
m.col<-cbind(x,y) # now use cbind to bind two vectors column wise
m.col

##      x y
## [1,] 4 8
## [2,] 2 5
## [3,] 3 6
## [4,] 6 9

m.row<-rbind(x,y) # now use rbind to bind two vectors row wise
m.row

##   [,1] [,2] [,3] [,4]
## x    4    2    3    6
## y    8    5    6    9

### Extract the elements of matrices.
## Square bracket [] indexing method. Elements can be accessed as var[row, column].

```

```

# First create a new matrix
m2<-matrix(1:12, nrow=3, ncol=4, byrow = TRUE)
class(m2)

## [1] "matrix" "array"

m2[1,] # Extracts first row

## [1] 1 2 3 4

m2[,4] # Extracts 4th column

## [1] 4 8 12

m2[1,4] # Extracts first element in row 1 and column 4

## [1] 4

m2[c(1,3), c(1,2)]

##      [,1] [,2]
## [1,]    1    2
## [2,]    9   10

m2[-1,] # Leaves first row

##      [,1] [,2] [,3] [,4]
## [1,]    5    6    7    8
## [2,]    9   10   11   12

## Modify the matrix
m2[1,1]<-10 # Changes single element in first row and first column
m2

##      [,1] [,2] [,3] [,4]
## [1,]   10    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12

m2[m2>11]<-20 # Change the elements in matrix greater than 12
m2

##      [,1] [,2] [,3] [,4]
## [1,]   10    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   20

# Add column or row to existing matrix
m2<-cbind(m2, c(4,8,12))
m2

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   10    2    3    4    4
## [2,]    5    6    7    8    8
## [3,]    9   10   11   20   12

x<-c(4,2,3,6,7) # create a vector of x
m2<-rbind(m2,x)
m2

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

```

```
##      10      2      3      4      4
##      5      6      7      8      8
##      9     10     11     20     12
## x      4      2      3      6      7

# Note dimensions should be same to add vectors
m2<-rbind(m2, c(1,2))

## Warning in rbind(m2, c(1, 2)): number of columns of result is not a multiple of
## vector length (arg 2)

m2

##      [,1] [,2] [,3] [,4] [,5]
##      10      2      3      4      4
##      5      6      7      8      8
##      9     10     11     20     12
## x      4      2      3      6      7
##      1      2      1      2      1

# Check the dimensions of matrix
dim(m2)

## [1] 5 5

# Transpose the matrix
t(m2)

##              x
## [1,] 10 5 9 4 1
## [2,]  2 6 10 2 2
## [3,]  3 7 11 3 1
## [4,]  4 8 20 6 2
## [5,]  4 8 12 7 1
```

Data Frames

- Data Frames are more general than matrix, which has columns with different modes (numeric, character, factor).
- A data frame can be constructed by the `data.frame()`

```
# Create a data.frame
Genotypes <- c("Genotyp1", "Genotype1", "Genotype2", "Genotype2")
Replication <- c("1", "1", "2", "2")
Block<- c("Block1", "Block2", "Block1", "Block2")
Yield<-c(2500,3500,3200,4500)
mydata <- data.frame(Genotypes,Replication,Block,Yield)
class(mydata)

## [1] "data.frame"

# Check the structure
str(mydata)

## 'data.frame':  4 obs. of  4 variables:
## $ Genotypes : chr  "Genotyp1" "Genotype1" "Genotype2" "Genotype2"
## $ Replication: chr  "1" "1" "2" "2"
## $ Block      : chr  "Block1" "Block2" "Block1" "Block2"
## $ Yield      : num  2500 3500 3200 4500
```

```
# Change the variables
mydata$Block<-as.factor(mydata$Block)
mydata$Replication<-as.factor(mydata$Replication)
# Check the structure again
str(mydata)

## 'data.frame': 4 obs. of 4 variables:
## $ Genotypes : chr "Genotyp1" "Genotype1" "Genotype2" "Genotype2"
## $ Replication: Factor w/ 2 levels "1","2": 1 1 2 2
## $ Block : Factor w/ 2 levels "Block1","Block2": 1 2 1 2
## $ Yield : num 2500 3500 3200 4500

levels(mydata$Replication) # Determine the number of levels for replication.
```

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

```
# Check column names
names(mydata)
```

```
## [1] "Genotypes" "Replication" "Block" "Yield"
```

```
# Subsetting or Indexing
mydata[1:2,]
```

	Genotypes	Replication	Block	Yield
	Genotyp1	1	Block1	2500
	Genotype1	1	Block2	3500

```
mydata[c(1,3),]
```

	Genotypes	Replication	Block	Yield
1	Genotyp1	1	Block1	2500
3	Genotype2	2	Block1	3200

```
# Select columns using $ sign
mydata$Yield # select Yield column
```

```
## [1] 2500 3500 3200 4500
```

```
mydata[mydata$Yield>3500,] # Extract row that has yield greater than 3500
```

	Genotypes	Replication	Block	Yield
4	Genotype2	2	Block2	4500

```
# Subset based on factor levels
help("subset")
mydata2<-subset(mydata,Block=="Block1") # select just Block1
mydata2
```

	Genotypes	Replication	Block	Yield
1	Genotyp1	1	Block1	2500
3	Genotype2	2	Block1	3200

```
mydata2<-subset(mydata,Block=="Block1" & Replication=="1") # select just Block1
mydata2
```

	Genotypes	Replication	Block	Yield
	Genotyp1	1	Block1	2500

Lists

- List is a data structure having components of mixed data types.
- To create a list we use *function list()*

```
# Creating a simple list
```

```
my_list <- list(name = "Waseem", age = 39, height = 5.5, is_student = FALSE)
my_list
```

```
## $name
## [1] "Waseem"
##
## $age
## [1] 39
##
## $height
## [1] 5.5
##
## $is_student
## [1] FALSE
```

```
# Mixed List: Creating a list with different types of elements
```

```
mixed_list <- list(vector = c(1, 2, 3),
                  matrix = matrix(1:4, nrow = 2),
                  char = "Hello",
                  logical = c(TRUE, FALSE))
```

```
mixed_list
```

```
## $vector
## [1] 1 2 3
##
## $matrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $char
## [1] "Hello"
##
## $logical
## [1] TRUE FALSE
```

```
# Creating a nested list
nested_list <- list(person1 = list(name = "Alice", age = 25),
                    person2 = list(name = "Bob", age = 30))
# Extracting using $
# Accessing elements in the nested list
person1_name <- nested_list$person1$name
person2_age <- nested_list[["person2"]][["age"]]
```

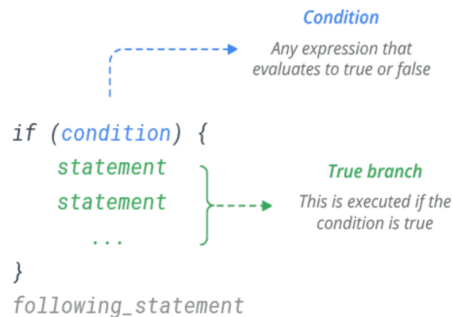
Section 3: Control Structures in R

Allow users to control the flow of execution of a series of R expressions

- Commonly used control structures are:
 - if and else*: testing a condition and acting on it
 - for*: execute a loop a fixed number of times
 - while*: execute a loop while a condition is true
 - repeat*: execute an infinite loop (must break out of it to stop)
 - break*: break the execution of a loop
 - next*: skip an iteration of a loop

If statement

- Execute a block of code, if a specified condition is true



Adopted from: <https://www.learnbyexample.org/r-if-else-elseif-statement/>

```
# Example 1
x <- 10
y <- 12
if(x < y) {
  print("x is less than 12!")
}
```

```
## [1] "x is less than 12!"
```

```
# Example 2
data("iris")
if(mean(iris$Sepal.Length) != mean(iris$Petal.Length)){
```



```
print("It is true")
}
```

```
## [1] "It is true"
```

If and else statement

- Execute a block of code, if the condition is false

Syntax

```
if (condition) {
  statement
  statement
  ...
} else {
  statement
  statement
  ...
}
following_statement
```

The diagram shows the execution flow of an if-else statement. The 'if' block is labeled 'True branch' and 'This is executed if the condition is true'. The 'else' block is labeled 'False branch' and 'This is executed if the condition is false'.

Adopted from: <https://www.learnbyexample.org/r-if-else-elseif-statement/>

```
# Example 1
x <- 10
y <- 12
if(x > y) {
  print("x is greater than y")
} else {
  print("y is greater than x")
}
```

```
## [1] "y is greater than x"
```

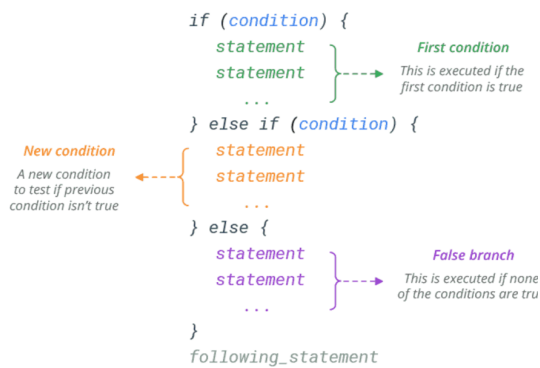
```
# Example 2
if(mean(iris$Sepal.Length)<mean(iris$Petal.Length)) {
  print("This is true and mean of sepal length is less than petal length")
} else {
  print("This is false and mean of petal length is less than sepal length")
}
```

```
## [1] "This is false and mean of petal length is less than sepal length"
```

Else If statement

- Specify a new condition to test, if the first condition is false.

Syntax



Adopted from: <https://www.learnbyexample.org/r-if-else-elseif-statement/>

```
# Example 1  
x <- 12  
y <- 13  
if(x > y) {  
  print("x is greater")  
} else if(x < y) {  
  print("y is greater")  
} else {  
  print("x and y are equal")  
}
```

```
## [1] "y is greater"
```

```
# Example 2  
x <- 12  
y <- 12  
if(x > y) {  
  print("x is greater")  
} else if(x < y) {  
  print("y is greater")  
} else {  
  print("x and y are equal")  
}
```

```
## [1] "x and y are equal"
```

Multiple Condition Statements

- Join two or more conditions into a single if statement
- Logical operators: && (and), ||(or) and ! (not). && (and) expression is True, if all the conditions are true
- For more details [Click Here](#)

```
# Example of Multiple conditions  
# First let us check means  
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

```
mean(iris$Petal.Length)
```

```
## [1] 3.758
```

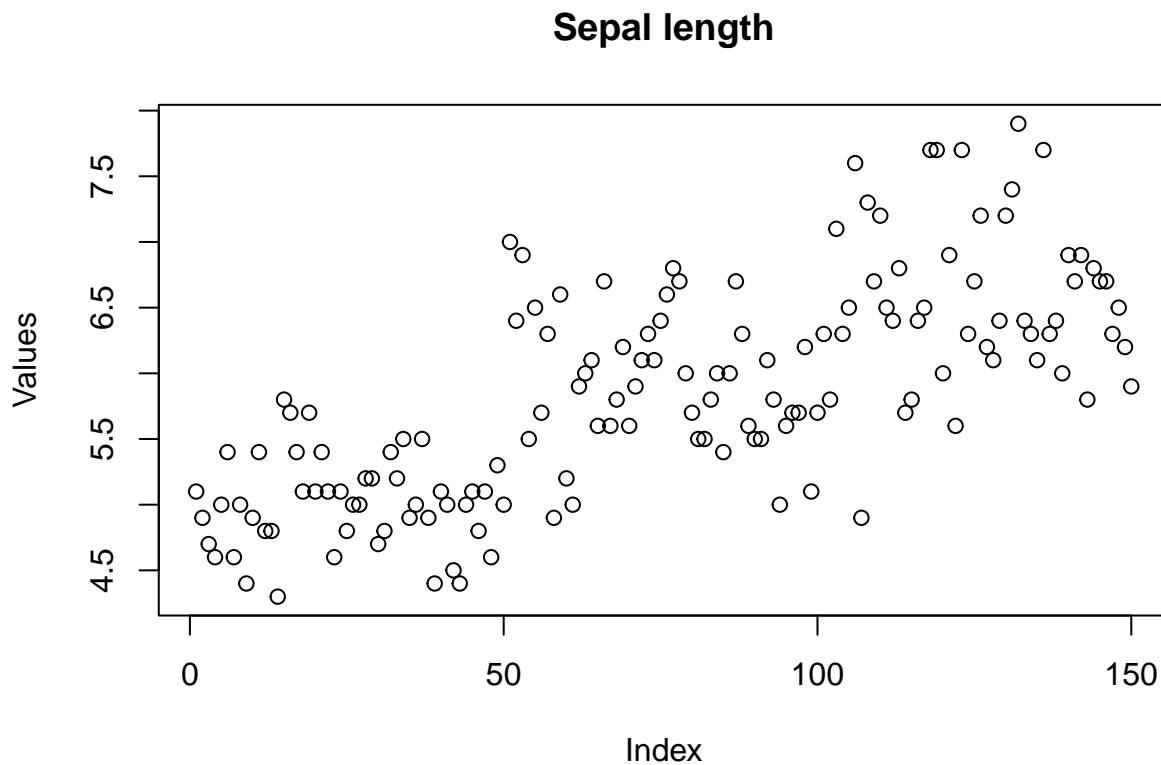
```
mean(iris$Sepal.Width)
```

```
## [1] 3.057333
```

```
mean(iris$Petal.Width)
```

```
## [1] 1.199333
```

```
# Test the multiple conditions using operators  
if(mean(iris$Sepal.Length)>mean(iris$Petal.Length) &&  
    mean(iris$Sepal.Width)>mean(iris$Petal.Width) ) {  
  plot(iris$Sepal.Length, main="Sepal length",  
       ylab="Values")  
} else {  
  plot(iris$Petal.Length, main="Petal length",  
       ylab="Values")  
}
```



ifelse Function

- Function checks the condition for every element of a vector and selects elements from the specified vector depending upon the result.

Syntax

`ifelse (condition, TrueVector, FalseVector)`

Condition

Condition is checked for every element of a vector

True branch

Select element from this if the condition is true

False branch

Select element from this if the condition is false

```
# IFELSE Example
# Example for ifelse function
iris$mycolumn<-ifelse(iris$Sepal.Length>iris$Petal.Length,
                      "TRUE", "FALSE")
iris$mycolumn<-ifelse(iris$Sepal.Length<iris$Petal.Length,
                      iris$Sepal.Length/iris$Petal.Length, NA)
# Let us change the values of sepal length
iris$Sepal.Length<-iris$Sepal.Length/3
iris$mycolumn2<-ifelse(iris$Sepal.Length>iris$Petal.Length,
                      iris$Sepal.Length/iris$Petal.Length, NA)
```

Apply Function

- Repetitively perform an action on multiple chunks of data.
- Runs faster than loop and requires less coding

Basic function: `apply(X, MARGIN, FUN)`

- X is an array or matrix.
- Margin specifies whether you want to apply the function across rows (1) or columns (2)
- FUN is the function you want to use

Apply function

```
# First create a matrix
my.matrx <- matrix(c(1:10, 11:20, 21:30), nrow = 10, ncol = 3)
my.matrx
```

```
##      [,1] [,2] [,3]
## [1,]    1  11  21
## [2,]    2  12  22
## [3,]    3  13  23
## [4,]    4  14  24
## [5,]    5  15  25
## [6,]    6  16  26
## [7,]    7  17  27
## [8,]    8  18  28
## [9,]    9  19  29
## [10,]  10  20  30
```

```

colnames(my.matrx)<-c("Length", "Breadth", "Width")
#Get sum across rows by using apply function
sumrow<-apply(my.matrx, 2, mean)
sumrow

## Length Breadth Width
##      5.5      15.5      25.5

# Creating own function
sumrow<-apply(my.matrx, 1, function (x) sum(x)*2)
sumrow

## [1] 66 72 78 84 90 96 102 108 114 120

# Creating function outside and then apply
# Creating a function to calculate Coefficient of variation
my.cofvar<- function(x){
  (sd(x)/mean(x))*100
}
# Now apply it to dataframe or matrix
cv<-apply(my.matrx,1, my.cofvar)
cv

## [1] 90.90909 83.33333 76.92308 71.42857 66.66667 62.50000 58.82353 55.55556
## [9] 52.63158 50.00000

```

lapply function

- *lapply()* operates on list and always returns a list, 'l' in *lapply()* refers to 'list'
- Syntax: *lapply(X, FUN, ...)*
 - X is a list
 - Fun, function to be applied
- Additional arguments passed to function
- *lapply()* always returns a list whereas *apply()* can return a vector, list, matrix or array.
- No scope of MARGIN in *lapply()*, always to columns

```

## lapply Function
# Create a list first
list.1<-list(Length=c(6,4,8,6.5),breadth=c(7,8,6,8),width=c(6.8,7.2,6.6,8))
list.1

## $Length
## [1] 6.0 4.0 8.0 6.5
##
## $breadth
## [1] 7 8 6 8
##
## $width
## [1] 6.8 7.2 6.6 8.0

# Get mean of all lists using lapply function
mean.all <- lapply(list.1,mean)
mean.all # Mean for all the lists

```

```
## $Length
## [1] 6.125
##
## $breadth
## [1] 7.25
##
## $width
## [1] 7.15
```

sapply Function

- *syntax for sapply() is as follows: sapply(x, fun,...)**
- *sapply() and lapply() work basically the same.*
- The only difference is that *lapply()* always returns a list, whereas *sapply()* tries to simplify the result into a vector or matrix.
- Additional argument if `simplify = F` then *sapply()* returns a list similar to *lapply()*

```
## sapply Function
# Create a vector first
vec<-c(1,2,3,4,5,6,7,8)
mean.vec <- sapply(vec,mean)
mean.vec
```

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

```
# Additional argument simplify
vec<-c(1,2,3,4,5,6,7,8)
mean.vec <- sapply(vec,mean, simplify = FALSE) # RETURNS REULTS AS LAPPLY
mean.vec
```

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

tapply Function

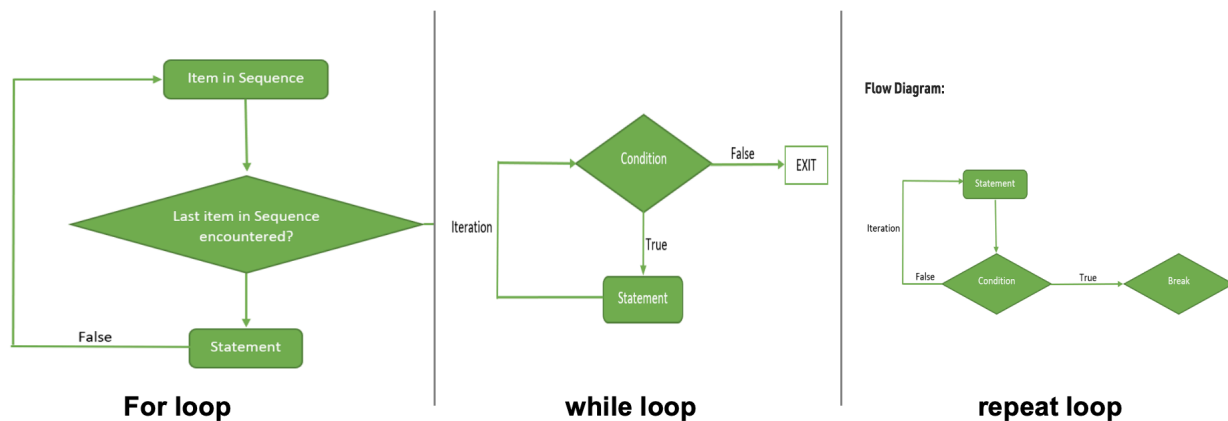
- `tapply()` function breaks the data set up into groups and applies a function to each group.
- The syntax: `tapply(x, INDEX, FUN, ..., simplify)`
- `x` is required vector
- A grouping factor or a list of factors
- The function to be applied
- Additional arguments
- Simplify return simplified results

```
# tapply Function
# First creating a simple data frame
sample.data <- data.frame(Genotypes=c("Geno1", "Geno2", "Geno3", "Geno4",
                                      "Geno5", "Geno6", "Geno7"),
                          Yield=c(240, 220, 211, 230, 203, 241, 212),
                          Type=factor(c("LR", "LR", "LR", "CV", "CV", "CV", "LR")))
# Now apply tapply function to get mean for various factor levels
mean<-tapply(sample.data$Yield, sample.data$Type, sum)
mean
```

```
## CV LR
## 674 883
```

Loops in R

- Loops run in cycling or iterating manner. They control statement that allows multiple executions of a statement or a set of statements.



Adopted from <https://www.geeksforgeeks.org/loops-in-r-for-while-repeat/>

for loop

- For loop, loops over texts, data frames etc.
- Loops repeatedly depending upon the number of elements

Syntax

```
for (var in vector) {
```

```
statement(s)
```

```
}
```

while loop

- Runs a statement or a set of statements repeatedly unless the given condition becomes false.
- Entry controlled loop.

repeat loop

- repeat loop run the same statement or a group of statements repeatedly until the stop condition has been encountered
- Iterate infinitely if no condition given

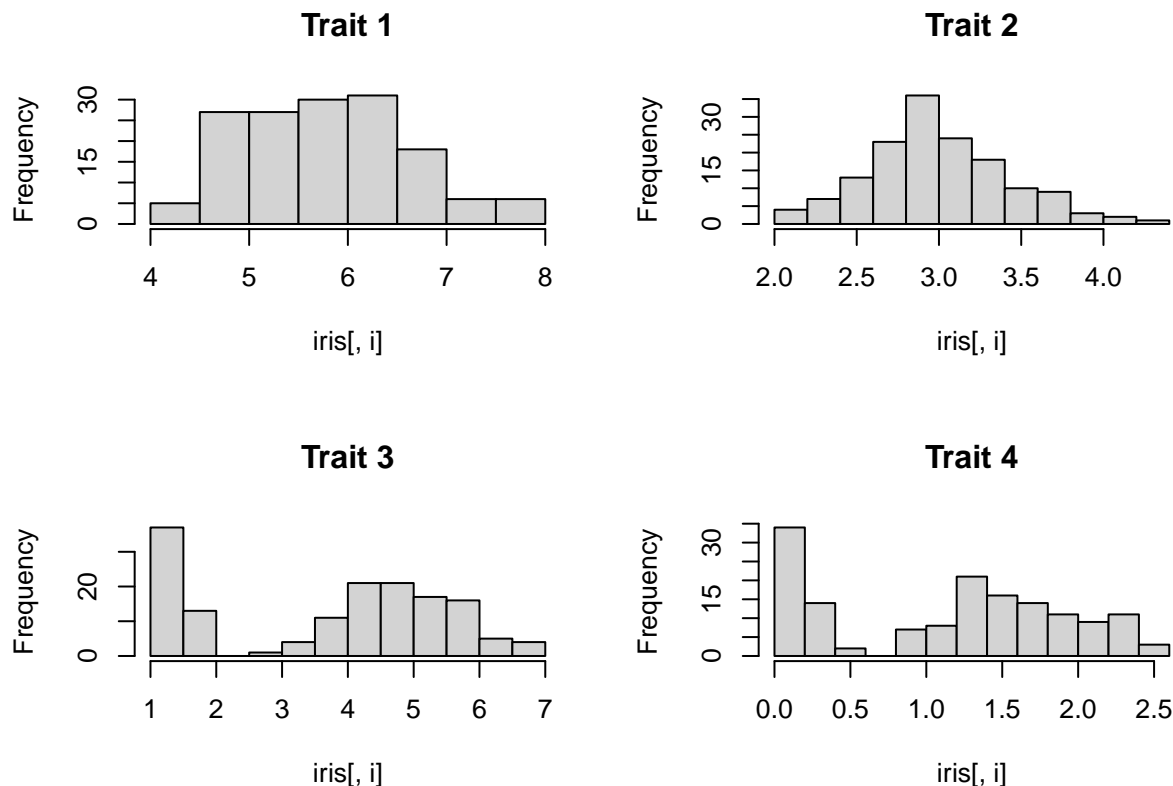
```
# For loop
# Example 1
for (i in 1:5) {
  print(i^3)
}
```

```
## [1] 1
## [1] 8
## [1] 27
## [1] 64
## [1] 125
```

```
# Example 2
x <- c(-8, 9, 11, 45)
for (i in x) {
  y<-x/2
  print(y)
}
```

```
## [1] -4.0  4.5  5.5 22.5
## [1] -4.0  4.5  5.5 22.5
## [1] -4.0  4.5  5.5 22.5
## [1] -4.0  4.5  5.5 22.5
```

```
# Example 3
# Histogram for iris data all columns
# Get iris data
data(iris)
# Histograms
par(mfrow = c(2, 2)) # Create 2 x 2 plotting matrix
for (i in 1:ncol(iris[,c(1:4)])){ # loop over columns
  plot <- hist(iris[,i], main=paste("Trait", i))
  plot
}
```

Your Assignment to run a simpl example of WHILE AND REPEAT LOOPS

Section 4: String Manipulations

- String manipulation refers to operations that modify, analyze, or otherwise work with text data (strings) in programming.
- In R, there are numerous functions for manipulating strings, allowing you to perform actions like concatenation, substring extraction, pattern matching, and more.

Basic String Manipulations

- `nchar()`: number of characters
- `tolower()`: convert to lower case
- `toupper()`: convert to upper case
- `casefold()`: case folding
- `chartr()`: character translation
- `abbreviate()`: abbreviation
- `substr()`: substrings of a character vector
- `paste()` or `paste0()`: combine strings, `paste0()` removes spaces.

More extensive details can be found in book [Handling and Processing Strings](#)

- We will use library stringr to do more manipulations

```

# Load library
library(stringr)
# First create a string
x<-c("ICAR", "IRRI", "Training")
# Get number of characters
nchar(x)

## [1] 4 4 8

# Convert to lower case
y<-tolower(x)
y

## [1] "icar"      "irri"      "training"

# Convert to upper case
y<-toupper(x)
y

## [1] "ICAR"      "IRRI"      "TRAINING"

# Upper or lower case conversion with casefold()
# upper case folding
y<-casefold(x, upper = TRUE)
y

## [1] "ICAR"      "IRRI"      "TRAINING"

# Replace 'IRRI' by 'IRRIGO'
help("chartr")
z<-"IRRI ICAR TRAINING"
z2<-chartr("I", "G",z)
z2

## [1] "GRRG GCAR TRAGNGNG"

# abbreviate species name column in iris data set
iris$Species<- abbreviate(iris$Species, minlength = 3)
head (iris$Species)

## [1] "sts" "sts" "sts" "sts" "sts" "sts"

#with package stringr
head(abbreviate(iris$Species, minlength = 4, dot = FALSE,
               strict = FALSE))

## sts sts sts sts sts sts
## "sts" "sts" "sts" "sts" "sts" "sts"

# Replace substrings with substr()
help("substr")
substr(iris$Species, 1, 3)<-"MY"
# Check similar function in stringr
# Concatenating strings
x<-c("ICAR", "Colloboration", "Genomic")
y<-c("IRR", "Training", "Prediction")
yx<- paste(x, y)
yx

## [1] "ICAR IRR"      "Colloboration Training" "Genomic Prediction"

```

```
# Without space
yx<- paste0(x, y) # Output: "HelloWorld"
yx
```

```
## [1] "ICARIRR" "ColloborationTraining" "GenomicPrediction"
```

Set Operations

- *union()*: set union
- *intersect()*: intersection
- *setdiff()*: set difference
- *setequal()*: equal sets identical() exact equality
- *is.element()*: is element
- *%in%()*: contains
- *sort()*: sorting
- *rep()*: repetition

```
# First create a two character vectors
set1 = c("ICAR", "IRRI", "TRAINING", "PROGRAM")
set2 = c("IRRI", "TRAINING", "PROGRAM", "2020")
# union of set1 and set2
union(set1, set2) #discards any duplicated values in the provided vectors
```

```
## [1] "ICAR" "IRRI" "TRAINING" "PROGRAM" "2020"
```

```
#Set intersection with intersect()
intersect(set1, set2) # common between two
```

```
## [1] "IRRI" "TRAINING" "PROGRAM"
```

```
# Set difference with setdiff()
setdiff(set2, set1)
```

```
## [1] "2020"
```

```
# First create set 3 and set 4
set3=c("IRRI", "ICAR", "TRAINING", "PROGRAM")
# Set equality with setequal()
setequal(set1, set3)
```

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

```
# Exact equality with identical()
identical(set1, set3)
```

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

```
# Element contained with is.element()
```

```
# Create elements
```

```
elem1 = "IRRI"
```

```
elem2 = "2020"
```

```
# elem1 in set 1
```

```
is.element(elem1, set1) #if an element is contained in a given set of character strings
```

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

```
# elem1 in set10?
```

```
elem1 %in% set1 # alternative
```

```
## [1] TRUE

# Sorting with sort()
#sort() allows us to sort the elements of a vector, either in increasing order (by
#default) or in decreasing order using the argument decreasing:
# sort (decreasing order)
set5<-c("boy", "girl", "apple")
set6<-c("boy", "girl", "apple", "2020")
sort(set5)

## [1] "apple" "boy"   "girl"

sort(set6)

## [1] "2020" "apple" "boy"   "girl"

#Repetition with rep()
# repeat 'x' 4 times
help("rep")
rep("Apple", 4)

## [1] "Apple" "Apple" "Apple" "Apple"

help("paste")
# Check similar functions in stringr
```

Regular Expressions

Patter matching and substitution

- *grep*: match a pattern
- *grepl*: similar to grep, output as logical
- *regexpr*: similar to grepl, output different and detailed
- *gregexpr*: similar to regexpr, output as list
- *sub()*: replacing one pattern with another one
- *gsub()*: replacing one pattern with another one (all occurrences)

All adopted from the same book given above

```
# Replacing first occurrence with sub()
# string
help("sub")
Rstring = c("The r Foundation",
            "for Statistical Computing",
            "R is FREE software",
            "r is a collaborative project")
# substitute 'R' with 'RR'
sub("R", "RR", Rstring, ignore.case = TRUE)

## [1] "The RR Foundation"          "foRR Statistical Computing"
## [3] "RR is FREE software"        "RR is a collaborative project"

#Replacing all occurrences with gsub()
# substitute
gsub("R", "RR", Rstring)

## [1] "The r Foundation"          "for Statistical Computing"
## [3] "RR is FRREE software"      "r is a collaborative project"
```

```
## Splitting Character Vectors
sentence = c("R is a collaborative project with many contributors")
# split into words
strsplit(sentence, " ")
```

```
## [[1]]
## [1] "R"          "is"          "a"           "collaborative"
## [5] "project"    "with"        "many"        "contributors"
```

```
# telephone numbers
tels<-c("5_1_0_548_20")
tels = c("510-548-2238", "707-231-2440", "650-752-1300")
# split each number into its portions
strsplit(tels, "_")
```

```
## [[1]]
## [1] "510-548-2238"
##
## [[2]]
## [1] "707-231-2440"
##
## [[3]]
## [1] "650-752-1300"
```

Check similar functions in stringr: your assignment

```
## Additional on Regular expressions
# Meta-characters
# string
money = "$money"
# the naive but wrong way
sub(pattern = "$", replacement = "", x = money)
```

```
## [1] "$money"
```

```
# the right way in R
sub(pattern = "\\$", replacement = "_", x = money)
```

```
## [1] "_money"
```

```
# Sequences
#replaces the first match, while gsub() replaces all the matches.
# replace digit with '_'
sub("\\d", "_", "the dandelion war 2010")
```

```
## [1] "the dandelion war _010"
```

```
gsub("\\d", "_", "the dandelion war 2010")
```

```
## [1] "the dandelion war ____"
```

```
sub("\\D", "_", "the dandelion war 2010")
```

```
## [1] "_he dandelion war 2010"
```

```
gsub("\\D", "_", "the dandelion war 2010")
```

```
## [1] "_____2010"
```

```
## [1] "_he dandelion war 2010"
gsub("\\S", "_", "the dandelion war 2010")
```

```
## [1] "/Users/waseemhussain/Documents/Research/Workshops/IRRI-IIRR_2024/Module1"
# Create a new working directory
dir.create("Test")

## Warning in dir.create("Test"): 'Test' already exists
```

Import and Export .csv Files

- The *read.csv()* function is for reading “.csv” files.
- The syntax for reading .csv files is *read.csv(filename or path, header=TRUE, sep=“ ”...)*

```
# Import .Txt file
# Read .csv file data into R. File in in Data folder
mydata<-read.csv("./Data/iris.txt", header = TRUE)
#Check this function to know more help("read.table")
# Export the file as .csv file
write.csv(mydata, "./Data/iris_export.csv",
          row.names=TRUE)
```

Import and Export .txt Files

- The R base function *read.table()* is a general function that can be used to read a file in table format.
- The R base function *write.table()* is used to export the .txt files.
- A simple function is *write.table(x, file, append=FALSE, sep=“ “, dec=“.”, row.names=TRUE, col.names=TRUE)*
- The data will be imported as a data frame. More can be found here [Link 1](#); and [Link 2](#)

```
# Import .Txt file
# Read tabular data into R
mydata<-read.table("./Data/iris.txt", header = TRUE, sep = "\t")
#Check this function to know more help("read.table")
# Export the file as .txt file
write.table(mydata, "./Data/iris_export.txt",
            append=FALSE, sep="/t", row.names=TRUE, col.names=TRUE)
```

Import and Export .xlsx Files

- For reading excel files or sheets we will use the R package *readxl*, For more details [Click here](#)
- For writing the excel sheets or files we will use R package
- We will uplaod the files from the folder **Data**, which is sub-folder of main directory.
- More on reading and exporting excel files can be found here [Link 1](#) and [Link 2](#)

```
# IMPORT EXCEL SHEETS
# Load the Library
library(readxl)
# Read the excel file
# Specify sheet by INDEX
my_data <- read_excel("./Data/iris.xlsx", sheet = 1)
# Specify sheet by Name
my_data <- read_excel("./Data/iris.xlsx", sheet = "IRIS_COPY")
# EXPORT EXCEL FILES
```

```
library(writexl)
# Writing the Existing file
#write_xlsx(my_data, "./Data/myIrisExoprt.xlsx")
# Check with Library ("xlsx")
```

Import from Web

- Here we will read data directly from web having rice data available at <http://ricediversity.org/data/index.cfm>.
- The phenotypic data has 34 traits phenotyped for 413 genotypes.

```
# Read from web directly
rice.pheno <- read.table("http://www.ricediversity.org/data/sets/44kgwas/RiceDiversity_44K_Phenotypes",
  header = TRUE, stringsAsFactors = FALSE, sep = "\t")
# First five rows and columns
rice.pheno[1:5, 1:5]
```

HybID	NSFTVID	Flowering.time.at.Arkansas	Flowering.time.at.Faridpur	Flowering.time.at.Aberdeen
081215-A05	1	75.08333	64	81
081215-A06	3	89.50000	66	83
081215-A07	4	94.50000	67	93
081215-A08	5	87.50000	70	108
090414-A09	6	89.08333	73	101

Import and Export .rds

- To preserve the data structures, such as column data types (numeric, character or factor), saving or exporting files in .rds is best option. .rds is the R data format and can be also used to save big files.
- More on this can be found [Click here](#)

```
# Save an .rds object in folder
saveRDS(my_data, file = "./Data/my_data.rds")
# Restore or read the same object
myrds<-readRDS(file = "./Data/my_data.rds")
```

Section 6: Data Wrangling and Manipulations

- There are many ways for data wrangling and manipulation tasks to perform in R. There are handful R packages to do Data wrangling and Manipulations and they fall in **tidyverse** R packages <https://www.tidyverse.org/packages/>.
- The **dplyr**, **tidyr**, **data.table** in tidyverse R packages provide a powerful and efficient way to work with data in R.
- Some resources on data wrangling and Manipulations are [Link 1](#), [Link 2](#), [Link 3](#), [Link 4](#), [Link 5](#); and [Link 6](#)
- Here we will use some examples from dplyr and tidyr R packages


```

# Install packages and load
packages = c("dplyr", "tidyr", "reshape", "reshape2")
# Create a function which will install the package if it is not installed
package.check <- lapply(packages, FUN = function(x) { # apply lapply to list of packages
  if (!require(x, character.only = TRUE)) {
    install.packages(x, dependencies = TRUE) # install dependencies if required'library(x, character
  }
})

# Read iris data file
iris<-read.csv(file="./Data/iris.csv", header = TRUE)
iris$Species.abr<-abbreviate(iris$Species, minlength = 3)
#####PACKAGEDPLYR#####

# Function FILTER
iris1<-iris %>% filter(Species == "setosa", Species.abr == "sts")
# same in base R
iris2<-iris[iris$Species == "setosa" & iris$Species.abr == "sts", ]
# Function ARRANGE
iris1<- iris1 %>% arrange(Sepal.Length)
# Use desc() to order a column in descending order:
iris1<-iris1%>% arrange(desc(Sepal.Length))
#Choose rows using their position with slice()
iris %>% slice(1:5)

```

Lines	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Species.abr
1	5.1	3.5	1.4	0.2	setosa	sts
2	4.9	3.0	1.4	0.2	setosa	sts
3	4.7	3.2	1.3	0.2	setosa	sts
4	4.6	3.1	1.5	0.2	setosa	sts
5	5.0	3.6	1.4	0.2	setosa	sts

```

# in base R
test<-iris[c(1,5, 6:10), ]
# Drop
iris1<-iris %>% slice(-c(1:5))
# Function SELECT
# Select columns by name
iris1<-iris %>% select(Sepal.Length, Sepal.Width)
# Select all columns between
iris1<-iris %>% select(Sepal.Length:Petal.Width)
# Select all columns except one
iris1<- iris %>% select(!Sepal.Length)
#iris %>% select(!c(Sepal.Length, Petal.Width))
# Select all columns ending with length
iris1<-iris %>% select(ends_with("Species"))
iris1<-iris %>% select(ends_with("abr"))
# more: starts_with(), ends_with(), matches() and contains() (Your assignment)
#select(iris, contains("Length"))
iris1<-iris %>% select(contains("."))
#iris1<-select(iris, contains("."))
# Function MUTATE

```

```
# Add single column
iris1<-iris %>% mutate(Additional = Sepal.Length/ 100)
# Add several columns
iris1<-iris%>%mutate(Additional2 = sqrt(Sepal.Length), Check = "vrs")
# What is alternative in R base function (Your Assignment)
# Function SUMMARISE
summary<-iris %>% summarise(Sepal.Length = mean(Sepal.Length, na.rm = TRUE))
summary
```

Sepal.Length
5.843333

```
iris %>% summarise(Mean=mean(Sepal.Length, na.rm = TRUE), SD=sd(Sepal.Length, na.rm = TRUE))
```

Mean	SD
5.843333	0.8280661

```
# Summaries with column names
iris %>% summarise(mean(Sepal.Length, na.rm = TRUE),mean(Petal.Length, na.rm = TRUE))
```

mean(Sepal.Length, na.rm = TRUE)	mean(Petal.Length, na.rm = TRUE)
5.843333	3.758

```
# Summarise a subset of rows
iris%>%
  slice(1:10) %>%
  summarise(sum(Sepal.Length))
```

sum(Sepal.Length)
48.6

```
#Function DISTINCT
iris%>% distinct(across(Species))
```

Species
setosa
versicolor
virginica

```
#####PIPE MULTIPLE TASKS#####
# Pipe to perform multiple tasks
test<-iris %>%
  filter(Species == "setosa")%>% # filter
  mutate(SePAL.2=Sepal.Length/ 10)%>% # create new column
  select(SePAL.2, Species) # and select the column
# Another pipe example
```

```
test2<- iris%>%
  group_by(Species) %>%      # Group by these variables
  summarise( # Summarise data
    length.mean = mean(Sepal.Length),
    width.mean = median(Sepal.Width),
    n = n()
  )
# Assignmet: Check with Tidy, reshape2, data.table functions
```

Section 7: Basic and Advanced Graphics

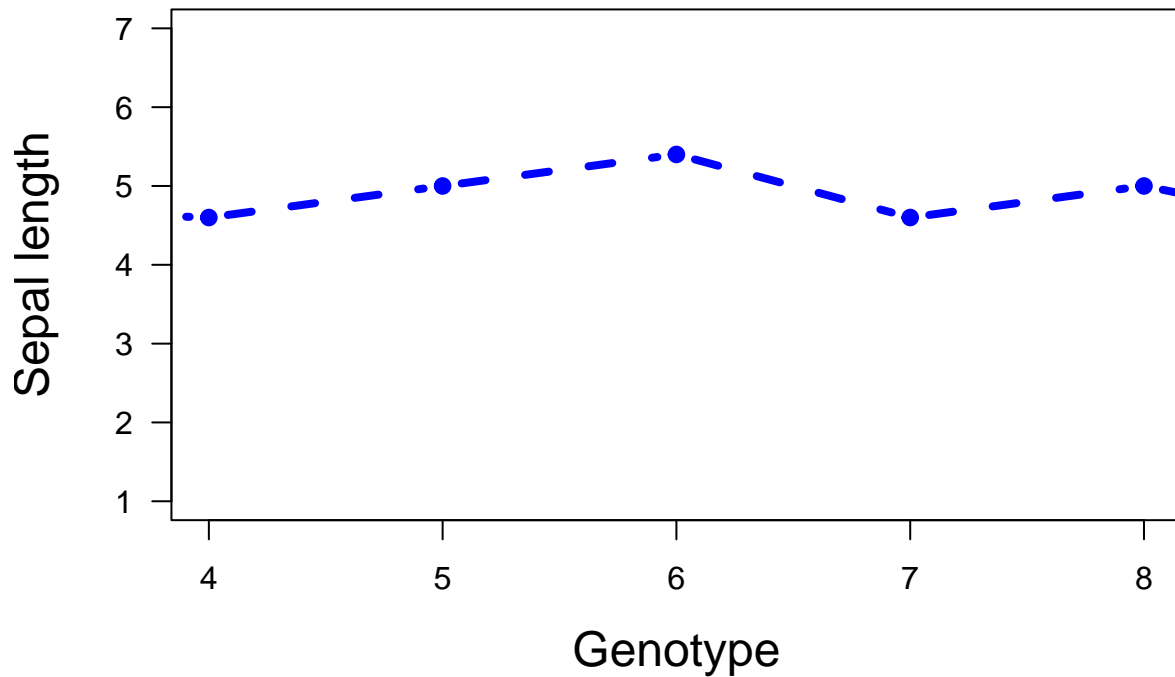
Basic Graphics with R

- In this section we will learn basic data visualization using base function *plot()*.
- More on the basic graphics with inbuilt functions in R can be found [Click here](#).
- Here will will plot various plots like scatter and line plot, bar plot, histograms etc.,

Scatter Plot

```
#SCATTER PLOT
plot(iris$Sepal.Length,
# Add features
  xlab="Genotype", # Add x label name
  ylab="Sepal length", # add y lable name
  main="Scatter plot for Sepal length",# add main title
  type="b",
# Different types of graphsx
  las=1, #0 is the default, with text always parallel to its axis.
  pch=20,
  cex=1,
  col="blue",
  lty=2, # line type
  lwd=4, # line or point width
  font.lab = 1, # font type
  col.lab="black",
  cex.lab = 1.5, # size of axis
  xlim=c(4,8), ylim=c(1,7), # adjust limits of axis
)
```

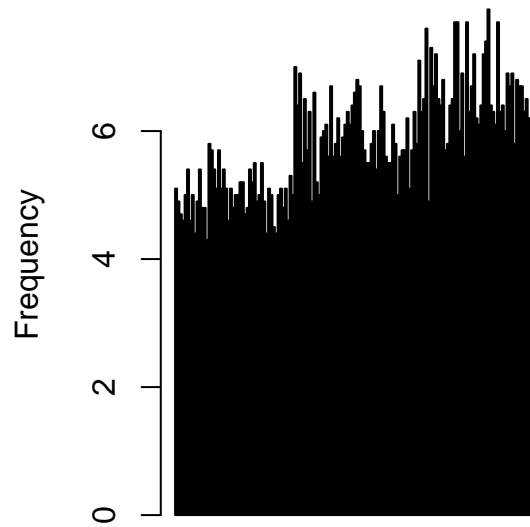
Scatter plot for Sepal length



Barplot

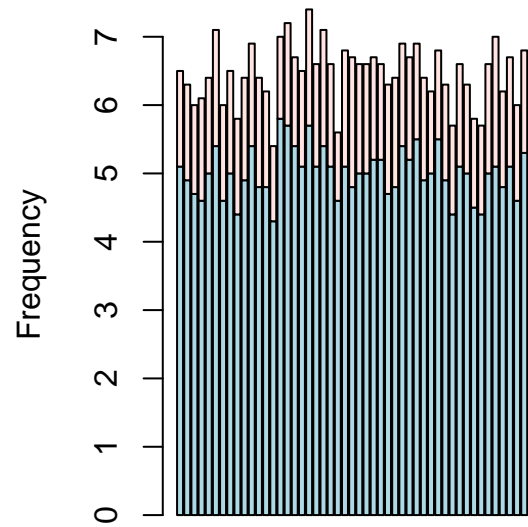
```
# BAR PLOTS
par(mfrow = c(1,2)) # Plot in One row with two Columns
barplot(iris$Sepal.Length,
main="Bar Plot for Sepal length",
xlab="Sepal length", ylab="Frequency",
col="blue",
width = 5,
horiz = FALSE)
#?barplot
# do more on bar plots
barplot(cbind(Sepal.Length, Petal.Length)~Lines, data=iris,
subset = Species=="setosa",
col = c("lightblue", "mistyrose"),
main="Bar Plot for Sepal length",
xlab="Sepal length", ylab="Frequency")
```

Bar Plot for Sepal length



Sepal length

Bar Plot for Sepal length

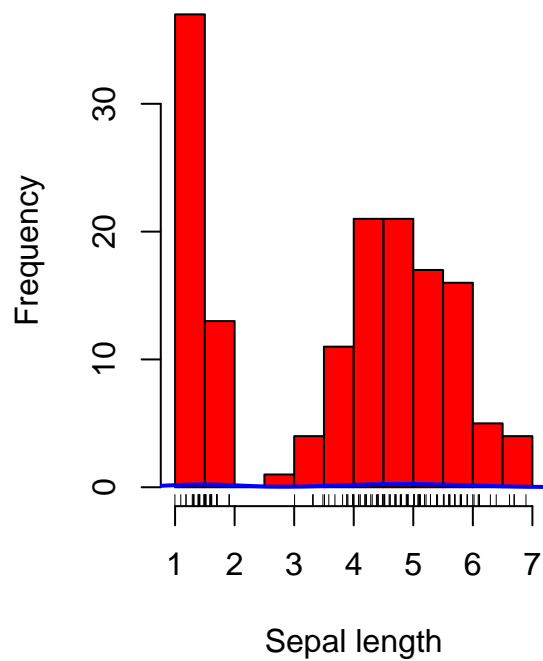


Sepal length

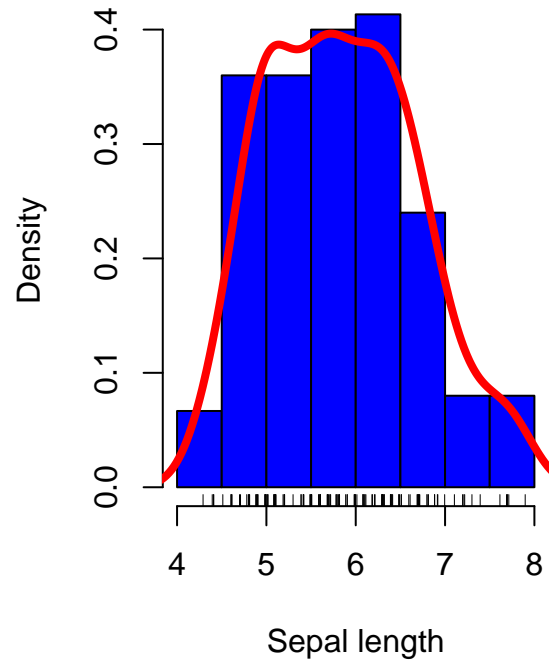
Histograms

```
par(mfrow = c(1,2))
hist(iris$Petal.Length,
     freq=TRUE,
     breaks=12,
     col="red",
     xlab="Sepal length",
     main="Histogram for Seapl length")
rug(jitter(iris$Petal.Length)) # add rugs
lines(density(iris$Petal.Length), col="blue", lwd=2) # add density curve
# Another graph
hist(iris$Sepal.Length,
     freq=FALSE,
     breaks=8,
     col="blue",
     xlab="Sepal length",
     main="Histogram for Seapl length")
rug(jitter(iris$Sepal.Length)) # add rugs
lines(density(iris$Sepal.Length), col="red", lwd=4) # add density curve
```

Histogram for Seapl length



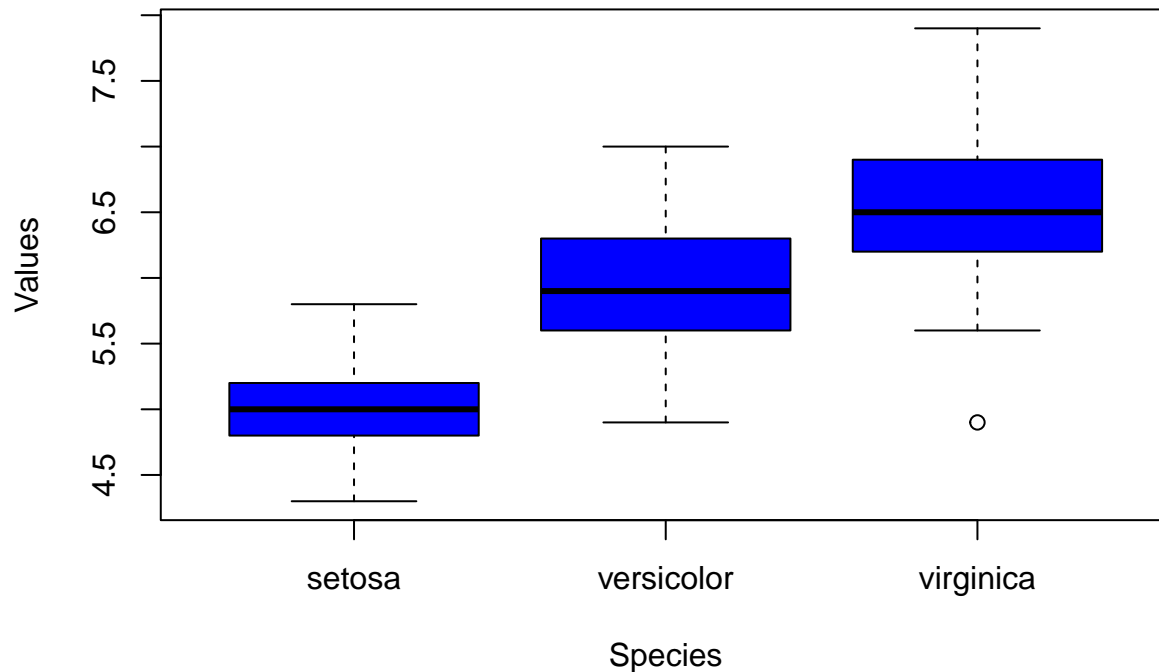
Histogram for Seapl length



Boxplots

```
#BOX PLOTS
boxplot(Sepal.Length ~ Species, data=iris,
        main="Box plot for Sepal length",col="blue",
        xlab="Species",
        ylab="Values")
```

Box plot for Sepal length



Advanced Graphics with ggplot2

- `ggplot2 ()` is a powerful R package for data visualization (<https://ggplot2.tidyverse.org/>)
- It uses “Grammar of Graphics” to create customizable and complex plots intuitively, making it ideal for both beginners and advanced users.
- It seamlessly integrates with data frames and extensive community support further enhance its appeal.

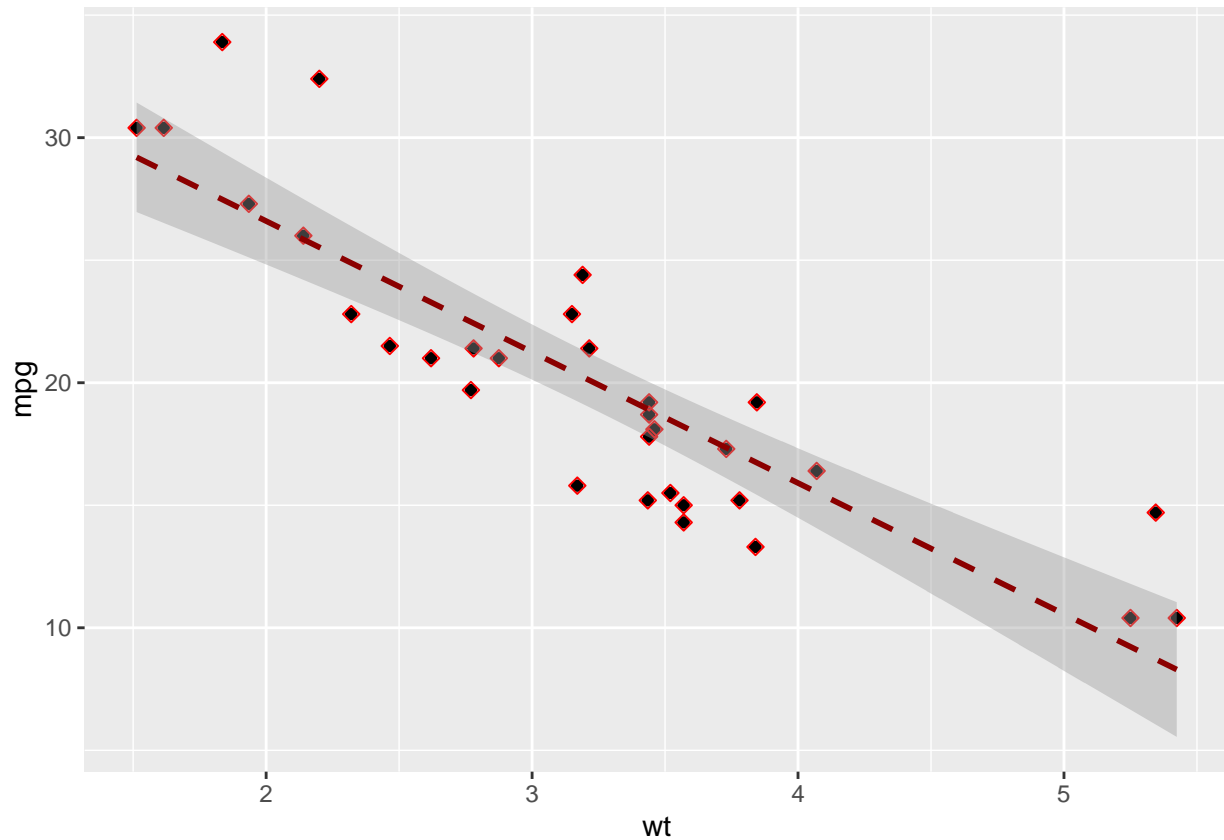
Best ggplot2 Resources: [Top 50 GRAPHS in ggplot2](#), [ggplot2 Tutorial](#), [TOP 25 PLOTS in ggplot 2](#), [VISUALIZATION in ggplot2](#); [Cheet Sheet ggplot2](#); [MAKE THIS YOUR FRIEND](#) #####

- Here we will plot some graphs using ggplot 2 and give you flavor how beautiful ggplot2 is in graphics
- We will also learn how to export high resolution figures from R [Click here for more](#)

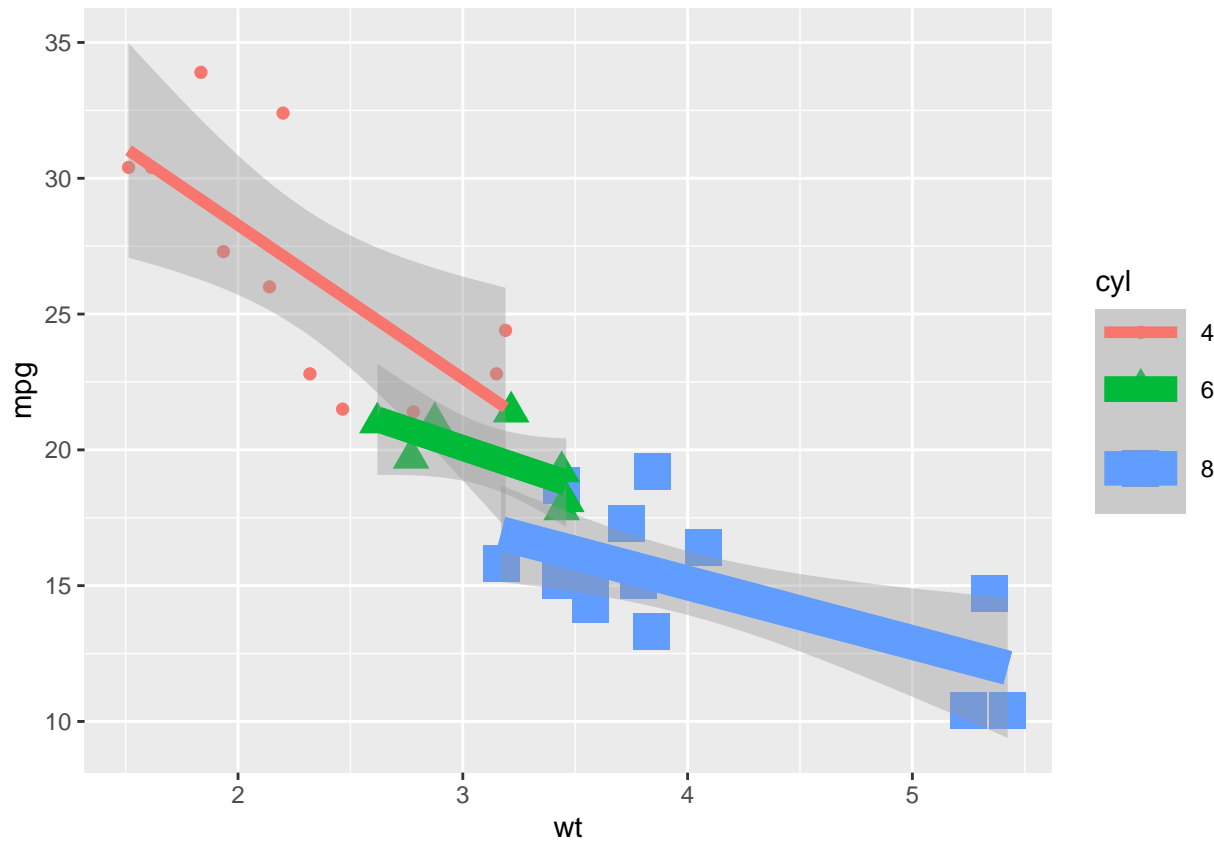
SCATTER PLOT

```
# SCATTER PLOT
# Install and Load the libraries
packages = c("ggplot2", "grid", "ggthemes", "plotly")
# Create a function which will install the package if it is not installed
package.check <- lapply(packages, FUN = function(x) { # apply lapply to list of package
  if (!require(x, character.only = TRUE)) {
    install.packages(x, dependencies = TRUE) # install dependencies if required
    library(x, character.only = TRUE) # Load the package
  }
})
# Use ggplot Function to plot
ggplot(mtcars, aes(x=wt, y=mpg))+
```

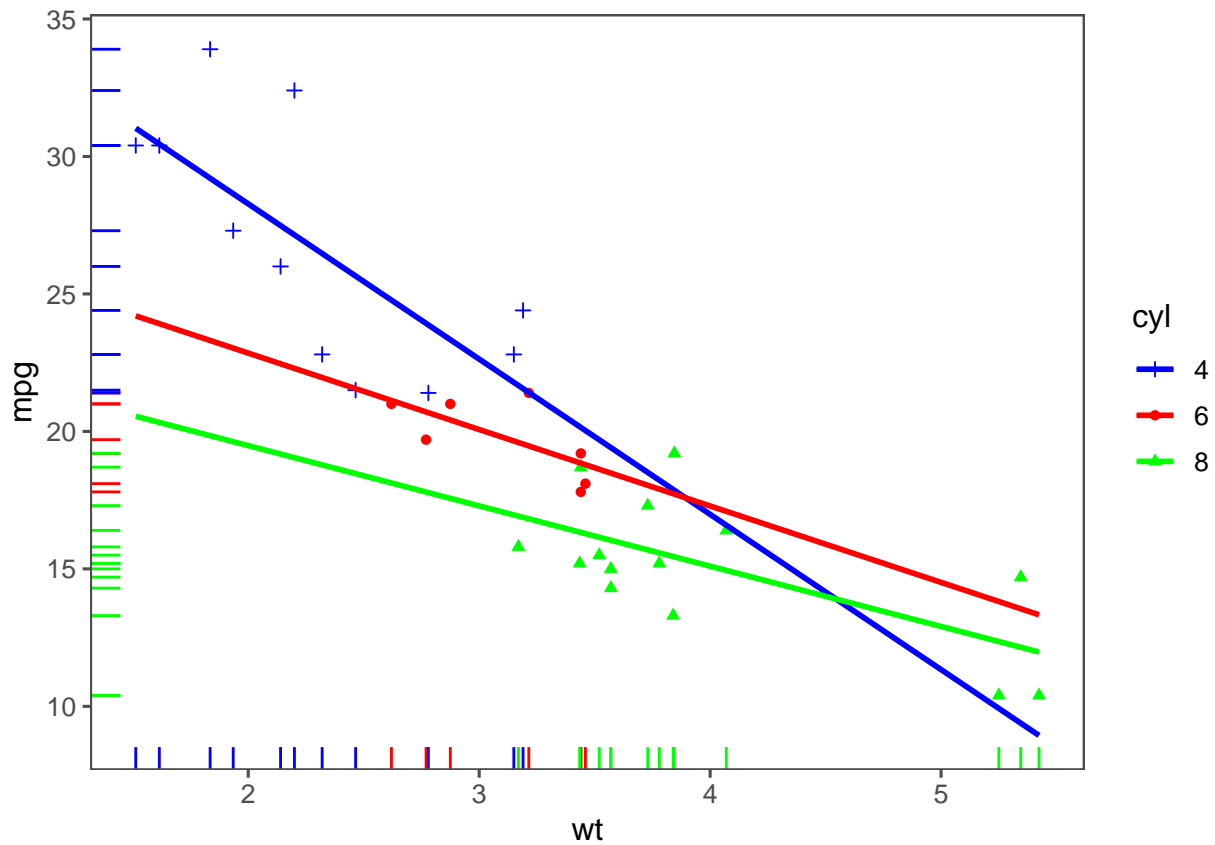
```
geom_point()+
geom_point(size=2, shape=23, color="red")+ # Change the point size, and shape
#geom_smooth(method=lm)+ # Add the regression line
#geom_smooth(method=lm, se=FALSE)+ # Remove the confidence interval
geom_smooth(method=lm, se=TRUE, linetype="dashed",
            color="darkred") # Change the line type and color
```



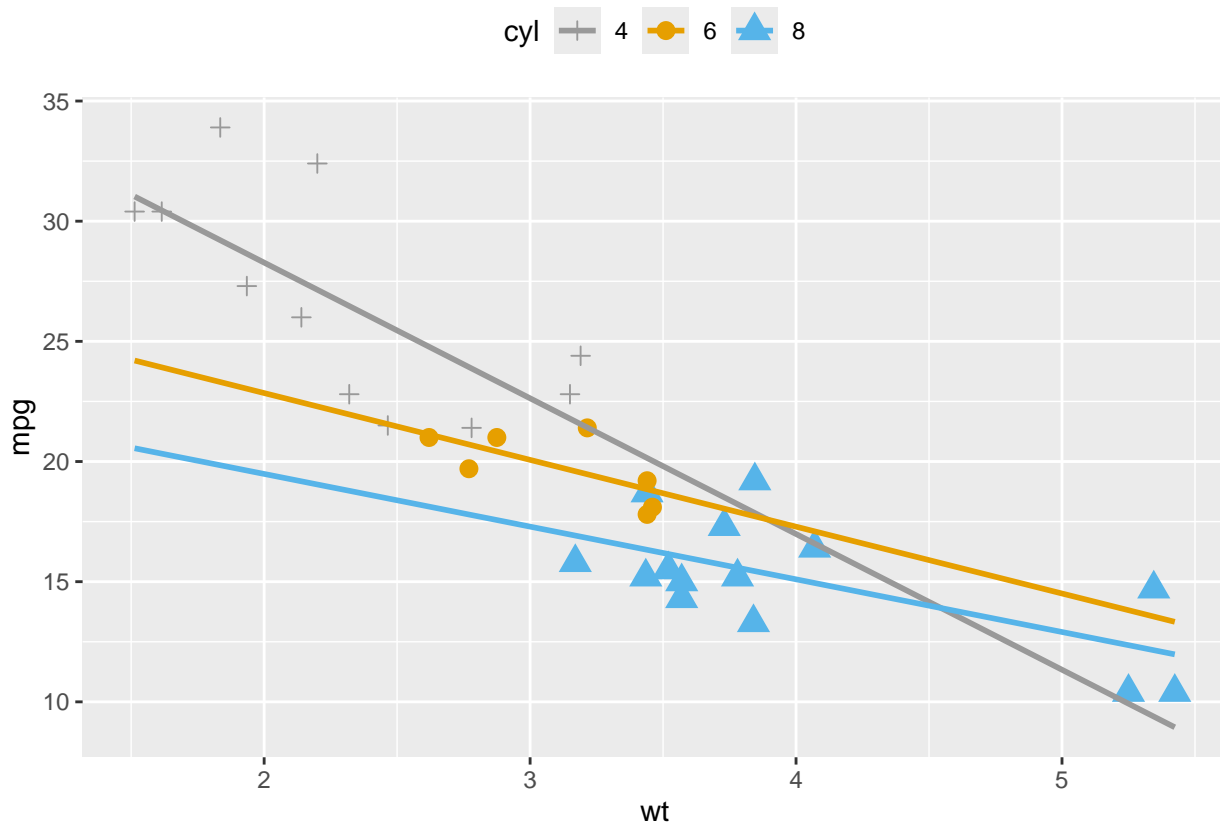
```
# SCATTER PLOT MULTIPLE GROUPS
# Change point shapes by the levels of cyl
mtcars$cyl<-as.factor(mtcars$cyl)
ggplot(mtcars, aes(x=wt, y=mpg, shape=cyl, color=cyl, size=cyl)) +
  geom_point()+
  geom_smooth(method=lm)+
  geom_smooth(method=lm, se=FALSE, fullrange=FALSE)
```

```
# Change point shapes and colors manually
ggplot(mtcars, aes(x=wt, y=mpg, color=cyl, shape=cyl))+
  geom_point()+
  geom_smooth(method=lm, se=FALSE, fullrange=TRUE)+
  scale_shape_manual(values=c(3, 16, 17))+
  scale_color_manual(values=c('blue', 'red', 'green'))+
  theme(legend.position="top")+
  geom_rug()+
  theme_few()
```



```
# Change the point sizes manually
ggplot(mtcars, aes(x=wt, y=mpg, color=cyl, shape=cyl))+
  geom_point(aes(size=cyl)) +
  geom_smooth(method=lm, se=FALSE, fullrange=TRUE)+
  scale_shape_manual(values=c(3, 16, 17))+
  scale_color_manual(values=c('#999999', '#E69F00', '#56B4E9'))+
  scale_size_manual(values=c(2,3,4))+
  theme(legend.position="top")
```

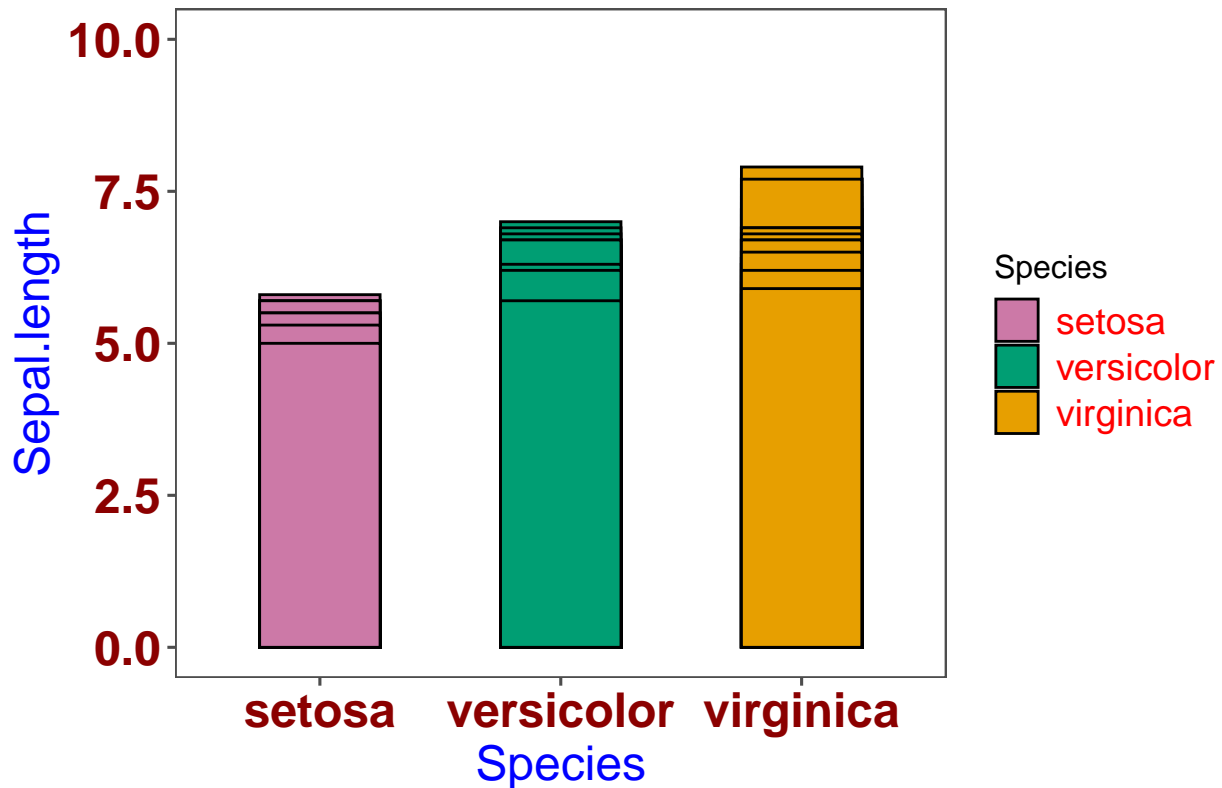


BAR PLOT

- Here we will also learn how to export the figures. To export we need to function first. For example, we will export as .png and rest of code followed the function and close it using `dev.off()` function at end.
- In R, the `dev.off()` function is used to close the current graphical device. When you create a plot, R opens a graphical device to display it (such as the R plotting window or a file like a PDF or PNG). Once you are done with your plotting and no longer need that graphical device, you call `dev.off()` to close it.

```
#png(file="myBar_plot.png", width=14, height =10,
#units = 'in',res=700) # function to Export as .png
#tiff("myBar_plot.tiff", units="in", width=5, height=5, res=500) # export as tiff
# pdf("myBar_plot.pdf", width = 4, height = 4) # Export as PDF
# ggplot 2 function
ggplot(mydata, aes(x = Species, y = Sepal.Length, fill=Species))+
  #geom_bar(stat = "identity", width=0.5, color="black")+
  geom_bar(stat = "identity", width=0.5,
           color="black", position=position_dodge())+
  theme_few() + # Select the theme
  ylim(0, 10)+ # Set Y limit
  labs(title = "", x = "Species", y = "Sepal.length")+ # add titles
  theme (axis.title.x = element_text(color="blue", size=18), # modify axis titles
         axis.title.y = element_text(color="blue", size=18))+
  theme(axis.text= element_text(face = "bold", color = "darkred", size =18))+
  scale_fill_manual(values=c("#CC79A7", "#009E73", "#e79f00"))+ # Manual coloring
  theme(legend.title = element_text(colour="black", size=12),
        legend.position = "right",
```

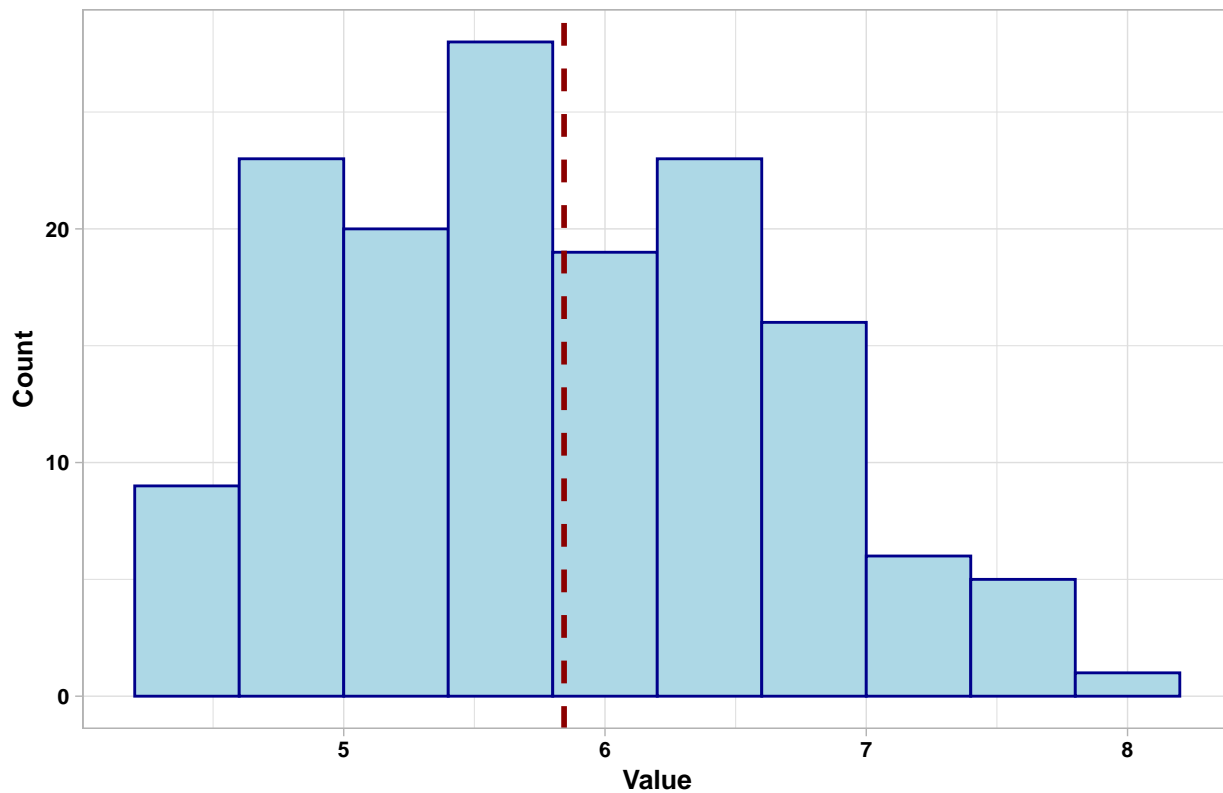
```
legend.text = element_text(colour="red", size=14))
```



```
#dev.off()
```

HISTOGRAMS

```
# Histograms in ggplot2
ggplot(mydata, aes(Sepal.Length)) +
  geom_histogram(color="darkblue", fill="lightblue", bins = 10)+
  theme_light()+
# adjust x values and breaks
#geom_histogram(breaks=seq(4, 8, by =0.5), color="darkblue", fill="lightblue")+
geom_vline(aes(xintercept=mean(Sepal.Length)), # adding the line to represent mean
  color="darkred", linetype="dashed", size=1)+
labs(title="",x="Value", y = "Count")+
theme (plot.title = element_text(color="black", size=14, face="bold", hjust=0),
  axis.title.x = element_text(color="black", size=10, face="bold"),
  axis.title.y = element_text(color="black", size=10, face="bold")) +
theme(axis.text= element_text(face = "bold", color = "black", size = 8))
```



- For more advanced plots check my [GitHub Page Additional ggplot2](#)
- For additional Correlation and Heatmap plots in R please check my [GitHub Page Correlation Plots**](#)

Section 8: Writing Own Functions

R function is pieces of code that perform a desired operation on given input(s) and return the output back to the user. In this section we will learn how to build own R functions and perform tasks. Building your own functions in R is straightforward. You can define a function using the ***function()*** keyword.

Here's the basic syntax:

Syntax

```
functionName <- function(argument1, argumen2...) {  
    #function Body  
    return(varc)  
}
```

- **Function Name** – Name of the function.
- **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument.
- **Function Body** – 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.

```
# Create simple function to calculate coefficient of variation (CV)  
# Build function  
my.cofvar<- function(x){  
    (sd(x)/mean(x))*100  
}  
# Create a data frame  
sample.data2 <- data.frame(Genotypes=c("Geno1","Geno2","Geno3","Geno4",  
                                       "Geno5","Geno6","Geno7"),  
                           Yield=c(240, 220, 211, 230, 203, 241, 212),  
                           Height=c(110, 140, 160, 135, 125, 145, 150),  
                           Type=factor(c("LR","LR","LR","CV",  
                                          "CV","CV","LR")))  
  
# Now calculate CV for Yield and Height variable  
my.cofvar(sample.data2$Yield)
```

```
## [1] 6.702938
```

```
my.cofvar(sample.data2$Height)
```

```
## [1] 12.00314
```

For handful resource on writting own R Functions:

1. [A tutorial for writing functions in R](#)
2. [Write your own R functions](#)
3. [Writing Your Own Functions in R: Introduction](#)
4. [How to Make a Custom R Package](#)

Section 9: R Markdown

R Markdown is a powerful tool for creating dynamic documents, reports, presentations, and dashboards that can combine text, code, and output. Key Features of R Markdown are:

- Data Analysis Reports
- Interactive Dashboards
- Reproducible Research
- Web Applications

Check our Analytical Pipeline created in R Markdown at <https://github.com/whussain2/Analysis-pipeline>

For handful resource on R Markdown

1. [R Markdown Tutorial 1](#)
2. [R Markdown from R Studio](#)
3. [R Markdown Tutorial 2](#)

Demo of R Markdown in R

For any suggestions or comments, please feel to reach at waseem.hussain@irri.org; and m.anumalla@irri.org
