R Training

Using this document:

There are couple of ways you could use this document -
1. Open this file in Rstudio and follow the document through each section. To execute a line of code, press 'Cntrl + Enter' while you are in that line
2. You could also refer to the PDF file (same content as markdown file), which contains all the codes and outputs

We suggest that you use option 1 while learning and option 2 to refer

R as a calculator

Let us start with some basic math operations.

```
2 + 3 # If you are in Rstudio, bring the pointer to this line and press 'Cntrl + Enter' to see the outp
```

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

```
4 * 5
```

```
## [1] 20
```

```
3 ^ 2
```

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

```
119 / 13
```

```
## [1] 9.154
```

```
119 %/% 13
```

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

```
119 %% 13
```

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

```
(9*13) + 2
```

```
## [1] 119
```

Hope you figured out what the operators '%/%' and '%%' does!!

In-built functions

Now let us look at some inbuilt functions.

```r
floor(5.7)
```

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

```r
ceiling (5.7)
```

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

```r
sum(5,3,4)
```

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

The values inside the brackets () are called parameters, which when passed to the function gets evaluated and resulting value is returned.

```r
log(10) # log to the base e
```

```
## [1] 2.303
```

```r
log10(10) # log to the base 10
```

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

```r
log(9,3) # Did you guess what is being done here? If not next step might help you
```

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

```r
exp(2) # Exponential
```

```
## [1] 7.389
```

Further functions:
sin(), cos(), tan(), asin(), acos(), atan(),
sinh(), cosh(), tanh(), asinh(), acosh(), atanh()
sum(), prod(), abs(), sqrt(), max(), min(), factorial(), choose()

```
Work out #1
1. Can you round the value to its nearest integer using either floor or ceiling function only?
   eg 5.7, 7.3,2.51 and 9.99 are converted to 6,7,3 and 10 respectively
```

R Help

R has tons of help files which you would be making use of very often. Apart from help on parameters to be provided, data type required for input, R also provides demo and examples on various uses of the functions.

```r
# When exact function is known use '?FunctionName'
# Now can you figure out what log(9,3) does?
?log
```

```
## starting httpd help server ... done
```

```r
# When exact function not known us help.search("search term")
help.search("data input")

# One can also see the example of a function
example(sort)
```

```
##
## sort> require(stats)
##
## sort> x <- swiss$Education[1:25]
##
## sort> x; sort(x); sort(x, partial = c(10, 15))
##  [1] 12  9  5  7 15  7  7  8  7 13  6 12  7 12  5  2  8 28 20  9 10  3 12
## [24]  6  1
##  [1]  1  2  3  5  5  6  6  7  7  7  7  7  8  8  9  9 10 12 12 12 12 13 15
## [24] 20 28
##  [1]  3  2  5  5  1  6  6  7  7  7  7  8  7  8  9  9 10 12 12 12 12 20 28
## [24] 13 15
##
## sort> ## illustrate 'stable' sorting (of ties):
## sort> sort(c(10:3, 2:12), method = "sh", index.return = TRUE) # is stable
## $x
##  [1]  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
##
## $ix
##  [1]  9  8 10  7 11  6 12  5 13  4 14  3 15  2 16  1 17 18 19
##
##
## sort> ## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## sort> ## $ix: 9  8 10  7 11  6 12  5 13  4 14  3 15  2 16  1 17 18 19
## sort> sort(c(10:3, 2:12), method = "qu", index.return = TRUE) # is not
## $x
##  [1]  2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
##
## $ix
##  [1]  9  8 10 11  7  6 12  5 13  4 14  3 15  2 16 17  1 18 19
##
##
## sort> ## $x : 2  3  3  4  4  5  5  6  6  7  7  8  8  9  9 10 10 11 12
## sort> ## $ix: 9 10  8  7 11  6 12  5 13  4 14  3 15 16  2 17  1 18 19
## sort>
## sort> x <- c(1:3, 3:5, 10)
##
## sort> is.unsorted(x)                  # FALSE: is sorted
## [1] FALSE
##
```

```
## sort> is.unsorted(x, strictly = TRUE) # TRUE : is not (and cannot be)
## [1] TRUE
##
## sort>                                  # sorted strictly
## sort> ## Not run:
## sort> ##D ## Small speed comparison simulation:
## sort> ##D N <- 2000
## sort> ##D Sim <- 20
## sort> ##D rep <- 1000 # << adjust to your CPU
## sort> ##D c1 <- c2 <- numeric(Sim)
## sort> ##D for(is in seq_len(Sim)){
## sort> ##D   x <- rnorm(N)
## sort> ##D   c1[is] <- system.time(for(i in 1:rep) sort(x, method = "shell"))[1]
## sort> ##D   c2[is] <- system.time(for(i in 1:rep) sort(x, method = "quick"))[1]
## sort> ##D   stopifnot(sort(x, method = "s") == sort(x, method = "q"))
## sort> ##D }
## sort> ##D rbind(ShellSort = c1, QuickSort = c2)
## sort> ##D cat("Speedup factor of quick sort():\n")
## sort> ##D summary({qq <- c1 / c2; qq[is.finite(qq)]})
## sort> ##D
## sort> ##D ## A larger test
## sort> ##D x <- rnorm(1e7)
## sort> ##D system.time(x1 <- sort(x, method = "shell"))
## sort> ##D system.time(x2 <- sort(x, method = "quick"))
## sort> ##D stopifnot(identical(x1, x2))
## sort> ## End(Not run)
## sort>
## sort>
```

```
Work out #2
    1. Can you identify the function used to round the variable to nearest integer?
    2. Can you find the function to find square root of given number and also the function that returns
```

Libraries

There are too many R commands to load them into the memory when an R session starts. Therefore R commands are organised as packages. Often we are required to install a package in order to use the functions present in them.

For instance there are many plotting functions available in the base package (default package that has all functions we used till now), but when 3d plots are required then we would need to install the specific package (one time operation) and then load it when ever we need to use it.

install.packages('scatterplot3d') # Installs the package library(scatterplot3d) # Loads the package and all functions under it for our use library(help="scatterplot3d") # To view the functions under the package

Generating sequence

Many times we might want to generate a series of numbers following certain pattern. Two functions 'rep' & 'seq' are used for this.

```r
seq(3,8)
```

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

```r
rep(2,4)  #replicate first object (here 2) four times
```

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

The obbjects generated are in fact called Vectors, We would be seeing more of them in the next section.

Vectors

Creation

Vectors can be created by 'c', 'seq' or 'rep'

Different ways one could create a vector

```r
c(1,2,3,4)     # concatenate elements to a vector
```

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

```r
seq(from=3,to=11,by=1)
```

```
## [1]  3  4  5  6  7  8  9 10 11
```

```r
seq(from=3,to=11)
```

```
## [1]  3  4  5  6  7  8  9 10 11
```

```r
seq(3,11)
```

```
## [1]  3  4  5  6  7  8  9 10 11
```

```r
3:11
```

```
## [1]  3  4  5  6  7  8  9 10 11
```

```r
c(2:5,3:7)
```

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

```
    1. Generate the following sequence using 'seq' function - 1,4,7,10,13,16,19,22 (use ?seq)
    2. Using 'rep' function, create a vector 'vector_a' where vector_a = c(7,7,8,8,8,9,9,9,9)
```

Slicing and dicing

Elements inside a vector can be accesses with [] operator

```r
x <- c(12,15,13,17,11)
x
```

```
## [1] 12 15 13 17 11
```

```r
x[4]       # fourth element of the vector x
```

```
## [1] 17
```

```r
x[3:5]     # subvector with indices 3, 4 and 5
```

```
## [1] 13 17 11
```

```r
x[-2]      # the minus means 'without'
```

```
## [1] 12 13 17 11
```

```r
x[c(TRUE, TRUE, FALSE, FALSE, TRUE)] # Equivalent to x[c(1,2,5)]
```

```
## [1] 12 15 11
```

Vector operations are performed element by element.

```r
c(2,5,3) + c(4,2,7)
```

```
## [1]  6  7 10
```

```r
2 + c(2,5,3)              # same as c(2,2,2) + c(2,5,3)
```

```
## [1] 4 7 5
```

```r
c(3,2) * c(2,5,3,4)       # same as c(3,2,3,2) * c(2,5,3,4)
```

```
## [1]  6 10  9  8
```

```r
x <- 1:5
x
```

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

```r
x > 3                     # which elements are > 3
```

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

```
x[c(TRUE,FALSE,TRUE,FALSE,FALSE)]
```

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

```
x[x>3]=0                  # set all elements which are bigger than 3 to 0
x
```

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

```
Work out #4
    1. Given the same vector x as above, slice a sub-vector 'vector_b' where vector_b = c(12,11)
    2. What happens when you do C(2,3) + c(5,6,7)
```

When 2 vectors of different lengths are used in arithmentic operation, smaller vector is cycled through larger vector - in other words when one of the vector comes to an end and if there are still elements left in other vector then the operation is continued from the first element of the smaller vector.

Some in-built functions that can be applicable on vectors are

```
x<-c(1,3,6,2,9,4,3)
x
```

```
## [1] 1 3 6 2 9 4 3
```

```
length(x)                 # returns the length of the vector
```

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

```
rev(x)                    # returns the 'rev'ersed vector
```

```
## [1] 3 4 9 2 6 3 1
```

```
sort(x)                   # returns the sorted vector
```

```
## [1] 1 2 3 3 4 6 9
```

```
order(x)                  # the index vector for sorting
```

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

```
duplicated(x)             # identifies multiple elements
```

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

```
unique(x)                 # returns vector without multiple elements
```

```
## [1] 1 3 6 2 9 4
```

```r
rank(x)                    # returns the ranks of the elements in vector
```

```
## [1] 1.0 3.5 6.0 2.0 7.0 5.0 3.5
```

```r
max(x)                     # returns the maximum value
```

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

```r
min(x)                     # returns the minimum value
```

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

```r
which.max(x)               # returns the index of the maximum (first such index)
```

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

```
Work out #5
    1. Given a vector a <- c(4,6,2,4,12,9,15,16,3,6,19,7,20,13,17); how can you pick every other element
    Result -> [1]  4  2 12 15  3 19 20 17
    2. For the same vector a <- c(4,6,2,4,12,9,15,16,3,6,19,7,20,13,17); can you replace every 3rd vari
```

Assignment operator

In order to assign values to a variable we use '<-' instead of '=' in R.

```r
a <- 5 # Assigns value 5 to the variable a
b <- 119 %/% 13
c <- a * b
a
```

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

```r
b
```

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

```r
c
```

```
## [1] 45
```

Comparison

On comparison R return either a TRUE or FALSE value. This TRUE or FALSE can be further used to manipulate values later on.

```r
5 < 3
```

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

```
b > a
```

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

```
b*a == c # Double '=' is used for comparing 2 values
```

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

Instead of comparing with one value, you could also check if the value is present in a series

```
a <- 3
b <- seq(1,5)
a %in% b # Returns true if a is present in b
```

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

Logical operations

```
(5 < 3) | (b > a) # Logical OR
```

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

```
(5 < 3) & (b > a) # Logical AND
```

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

```
!(5 < 3) | (b > a) # The ! symbol reverses the subsequent results (TRUE into FALSE and vice versa)
```

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

```
!((5 < 3) | (b > a))
```

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

```
Work out #6
    1. Given a vector a <- c(4,6,2,4,12,9,15,16,3,6,19,7,20,13,17); how can you pick only the elements
    2. For the same vector a <- c(4,6,2,4,12,9,15,16,3,6,19,7,20,13,17); can you pick all the elements
```

Special values in R (na, nan, Inf)

Real life data would be replete with missing values. For instance in a survey data, certain respondents might not prefer to answer specific questions (like their salary for instance). While analysing it would be unwise to assign 0 to those values that we do not have. Hence missing values in R are replaced by a special value called 'NA'. NAs exibit some special chareacteristics during various operations

```r
# Assigning NA
x <- NA
x
```

```
## [1] NA
```

```r
# Comparison
x == NA # This would not tell us whether 'x' is NA or not
```

```
## [1] NA
```

```r
is.na(x) # This returns a TRUE or FALSE value depending on whether x is NA
```

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

```r
# NAs in arithmetic functions
sum(a,b,c)
```

```
## [1] 63
```

```r
sum(a,b,c,x)
```

```
## [1] NA
```

```r
# But many fucntions come with inbuilt NA handling capabilities
sum(a,b,c,x, na.rm = T)
```

```
## [1] 63
```

The internal constants Inf and -Inf represents infinity and minus infinity. Everything outside a certain range is Inf or -Inf for R.

```r
1.7e107
```

```
## [1] 1.7e+107
```

```r
1.8e710
```

```
## [1] Inf
```

```r
Inf*(-3)
```

```
## [1] -Inf
```

Another internal constant is NaN (not a number). Every calculation which is not defined results in NaN.

```r
0/0
```

```
## [1] NaN
```

```r
Inf-Inf
```

```
## [1] NaN
```

```
Work out #7
    1. Given a vector a <- c(23,43,23,7,5,487,NA,45,43,23,78,54,NA,45,12); how would you replace all NAs
    2. For the same vector a <- c(23,43,23,7,5,487,NA,45,43,23,78,54,NA,45,12), find the mean excluding
    3. Consider a <- 3; b <- NA;
         Then can you explain the results of (a < 5 | b < 5) ; (a > 5 | b > 5)
         Why only 2nd condition results in NA?
```

Analysis

Now you have enough basics to peform an analysis. Before any analysis is done there are series of steps taken in order to understand the data by either grouping them in form of summary or plotting them, manipulate the data to transformm them, create new variables etc..

Let us now look at series of steps usually done before any analysis.

Inbuilt data

R has many inbuilt data for practice. iris is one such data which we will be using for this analysis You can find list of in-built data by using the function data()

Understanding the data

In any analysis we first would try to understand the data at hand. Few things we generally observe are 1. Number of rows and columns 2. Names of the columns 3. Data types of the columns 4. Top few and bottom few rows

```r
class(iris) # Data type
```

```
## [1] "data.frame"
```

```r
dim(iris) # Gives the dimension (rows and columns)
```

```
## [1] 150   5
```

```r
nrow(iris) # Number of rows
```

```
## [1] 150
```

```r
ncol(iris) # Number of columns
```

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

```r
colnames(iris) # Column names
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```r
head(iris) # Top 5 rows
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
tail(iris) # Bottome 5 rows
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```r
str(iris) # Data type and few values of each column
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```r
summary(iris) # Summary of each columns are presented
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.30   Min.   :2.00   Min.   :1.00   Min.   :0.1
##  1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60   1st Qu.:0.3
##  Median :5.80   Median :3.00   Median :4.35   Median :1.3
##  Mean   :5.84   Mean   :3.06   Mean   :3.76   Mean   :1.2
##  3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10   3rd Qu.:1.8
##  Max.   :7.90   Max.   :4.40   Max.   :6.90   Max.   :2.5
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
View(iris)
```

Reading/ Writing data

Luckily here we are using in-built data. But in real analysis we need to import data from outside and also write the results to an outside file. Here we shall write the iris data and import it back

First we need to set the working directory getwd() setwd("C:/Users/Desktop/Training")

```
write.csv(iris,"./iris.csv")
iris_read <-  read.csv("./iris.csv")
```

```
Work out #8
    1. Can you obtain top 10 rows of iris data
    2. Can you obain only column names of first 4 columns of iris data?
```

Accessing the data

Data in a data frame is accessed using '[]' notation. Syntax - data_frame_name[rows to be selected , columns to be selected]

Columns to be selected can be indicated by column names, index of the column or through vector containing list of names or index. Rows can be selected based on meeting specified condition

Accessing a particular Column & Row

```
### Using column name
sep_wid <- iris$Sepal.Width
sep_wid <- iris[,'Sepal.Width'] # Notice the comma before column name? This is because the row value is
class(sep_wid) # This is a numeric vector
```

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

```
sep_wid <- iris[,c('Sepal.Width','Petal.Length')] # using multiple column names
class(sep_wid) # Since this has more than one column, it is now data frame
```

```
## [1] "data.frame"
```

```
names <- c('Sepal.Width','Petal.Length')
sep_wid <- iris[,names] #

### Using column index
sep_wid <- iris[,2] # Picks the 2nd column in the data frame
sep_wid <- iris[,c(2,3)]

### Combination of rows and index
sep_wid <- iris$Sepal.Width[1:50] # Extracts first 50 rows of the column Sepal.Width
```

Extracting rows based on applied condition

Suppose we want to pick the rows with setosa as species, how do we achieve it? Let us see what 'iris$Species == "setosa"' returns

```
iris$Species == "setosa"
```

```
##   [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [34]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [45]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

In the series of TRUE and FALSE, TRUE corresponds to rows which we want to retain while FALSE to rows we do not need. Hence if we feed this to the formula, rows with species as setosa is picked.

```
sep_wid <- iris[iris$Species == "setosa",c(2,3)]
dim(sep_wid) # Only 50 rows of setosa is retained now
```

```
## [1] 50  2
```

```
sep_wid <- subset(iris,Species=="setosa") # Another way to pick setosa
sep_wid <- iris[c(1:50),c(2,3)] # This picks the first 50 rows
```

What if we want to exclude certain columns or rows

```
sep_wid <- iris[!iris$Species == "setosa",-c(2,3)] # Excludes species setosa (due to ! symbol, which me
dim(sep_wid)
```

```
## [1] 100   3
```

```
Work out #8
    1. How can we select everything except the third column. Hint: help(subset)
    2. How do we get the mean of petal width for setosa species
```

House Keeping

Let us do some house keeping and remove unwanted objects or data

```
# List of variables in current session
objects() #or
```

```
## [1] "a"         "b"         "c"         "iris_read" "names"     "sep_wid"  
## [7] "x"
```

```
ls()
```

```
## [1] "a"         "b"         "c"         "iris_read" "names"     "sep_wid"  
## [7] "x"
```

```
# List of libraries and dataframes attached
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics" 
## [4] "package:grDevices" "package:utils"     "package:datasets" 
## [7] "package:methods"   "Autoloads"         "package:base"
```

To quickly change the variable names or values, use fix function

```
fix("iris_read")
```

Let us now remove the data that we read now ('iris_read')

```
rm(iris_read)

# To remove all the objects
rm(list = ls())
```

Fixing the data

Often we find the data we import to be of wrong data type. For example gender might be denoted by 1 & 2 and the data type of the column could be 'numeric' instead of being factor. We change the data type of such columns before starting the analysis. Though there is no need to change data type in this data set, we have listed few syntax for illustration.

```
iris$Species <- factor(iris$Species, c("setosa","versicolor","virginica"), ordered=FALSE)

#Or simply
iris$Species <- as.factor(iris$Species)

# Convert to numeric
iris$Petal.Length <- as.numeric(iris$Petal.Length)
```

Creating new variables

We often create additional variables to the one existing already - this could be new data or extracted from existing data

```r
# Say, we add a column to distinguish every time we perform analysis
iris$batch <- 1 # 1 will be repeated in all the rows

# Create a new variable from existing ones
iris$Sep_Pet_len <- with(iris, Sepal.Length/Petal.Length)

# Create categorical variable out of continuous variables
iris$Pet_len_large <- ifelse(iris$Petal.Length > 5,1, 0)

# After creating variables we see if data types are right
str(iris)
```

```
## 'data.frame':    150 obs. of  8 variables:
##  $ Sepal.Length : num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width  : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length : num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width  : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species      : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ batch        : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ Sep_Pet_len  : num  3.64 3.5 3.62 3.07 3.57 ...
##  $ Pet_len_large: num  0 0 0 0 0 0 0 0 0 0 ...
```

```r
# We see that variable 'Pet_len_large' is numerical. But we need this to be factor
iris$Pet_len_large <- as.factor(iris$Pet_len_large)
```

Ifelse function

ifelse is a simple conditional statement in R. Syntax - ifelse(Condition, To be executed if TRUE, To be executed if FALSE)

grep function

Grep is a very handy function used to extract objects that satisfy certain conditions from any list. For instance say we need names of all columns in iris data that corresponds to Width, we will need to search for the string "Width" in colummn names of iris. This can be done in following way:

```r
#The below statement returns the position of columns with names containing Width in them
grep("Width",colnames(iris))
```

```
## [1] 2 4
```

```
Work out #9
    1. Using ifelse, create a variable that takes value 1 if species is setosa and Sepal.Width less than
    2. Can you replicate the above step using 'if' condition (use help)
    3. Can you  remove only specific objects you want from the environment?
    4. For those rows corresponding to setosa species, can you change the value for barch column to be
    5. In the 'grep' example, can you use these indexes to return the corresponding names?
```

Plots

Next, we try to identfy the patterns between the variables by plotting them.

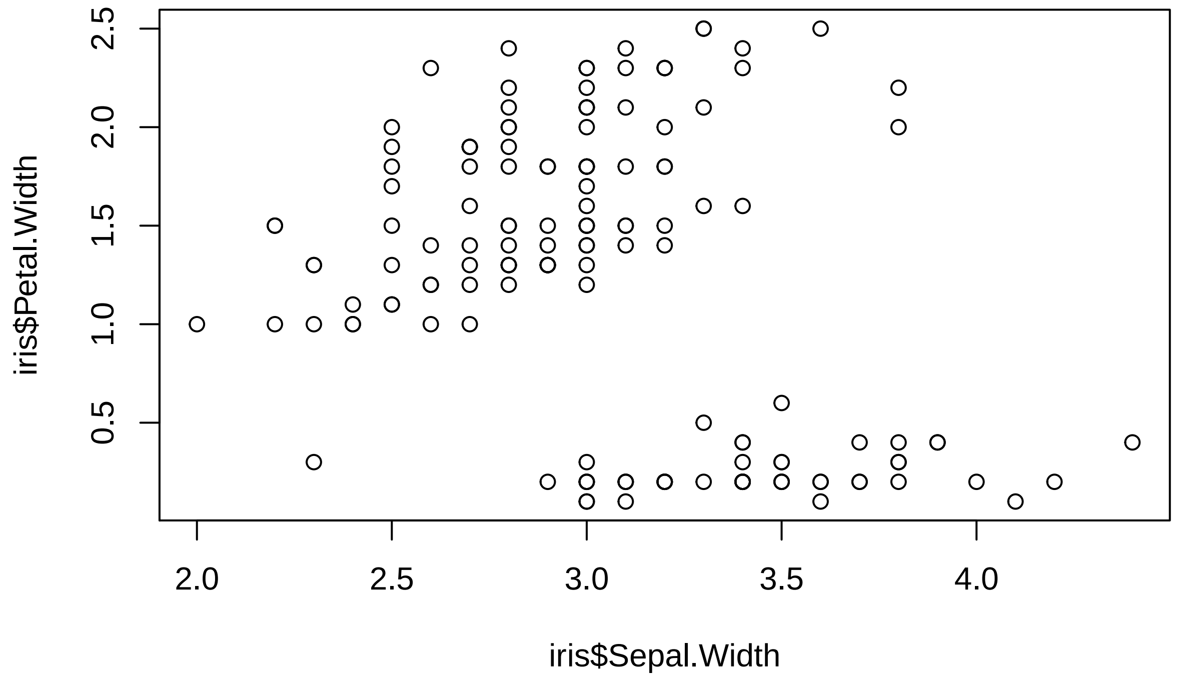
Plotting commands are of two types majorly

High-level plotting functions create a new plot (usually with axes, labels, titles and so on)

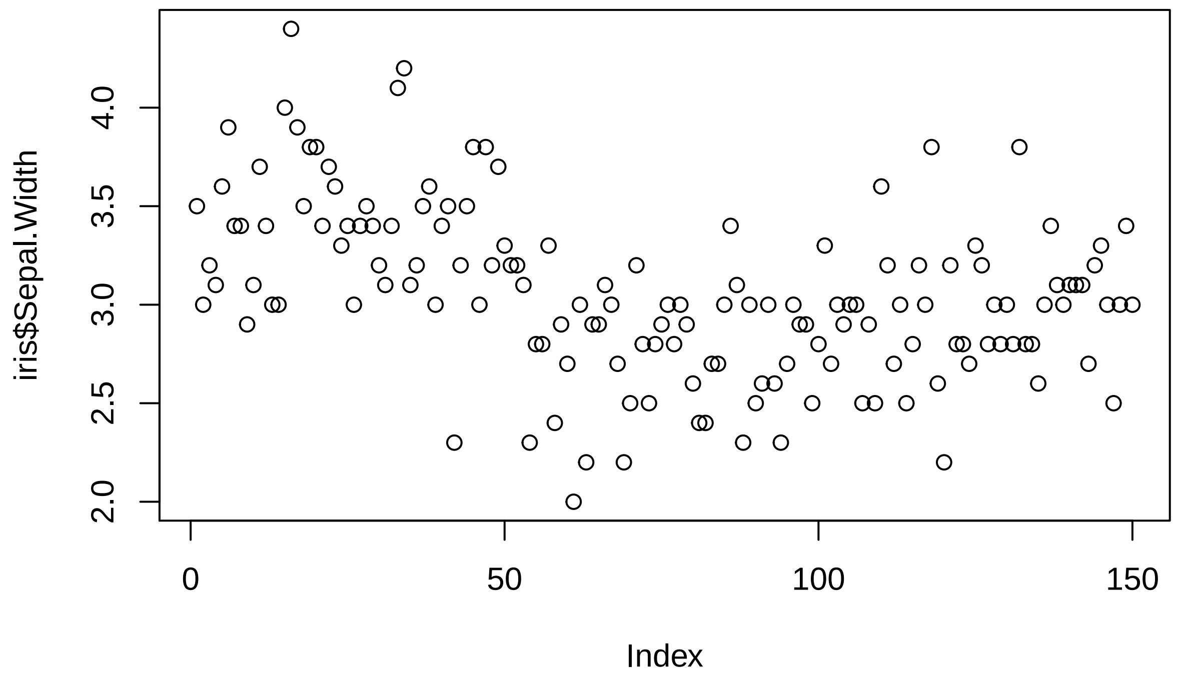
Low-level plotting functions add more information to an existing plot, such as extra points, lines or labels

High-level plotting commands

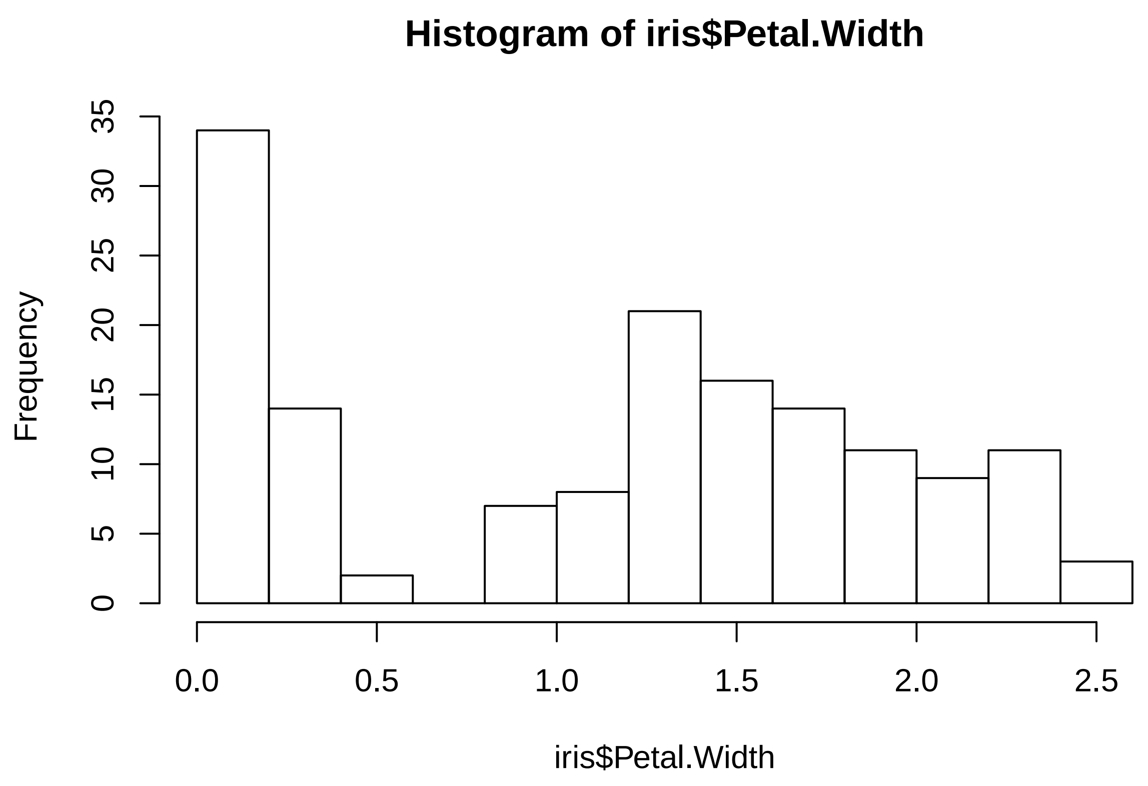
```
### produces a scatter plot between two variables
plot(iris$Sepal.Width, iris$Petal.Width)
```



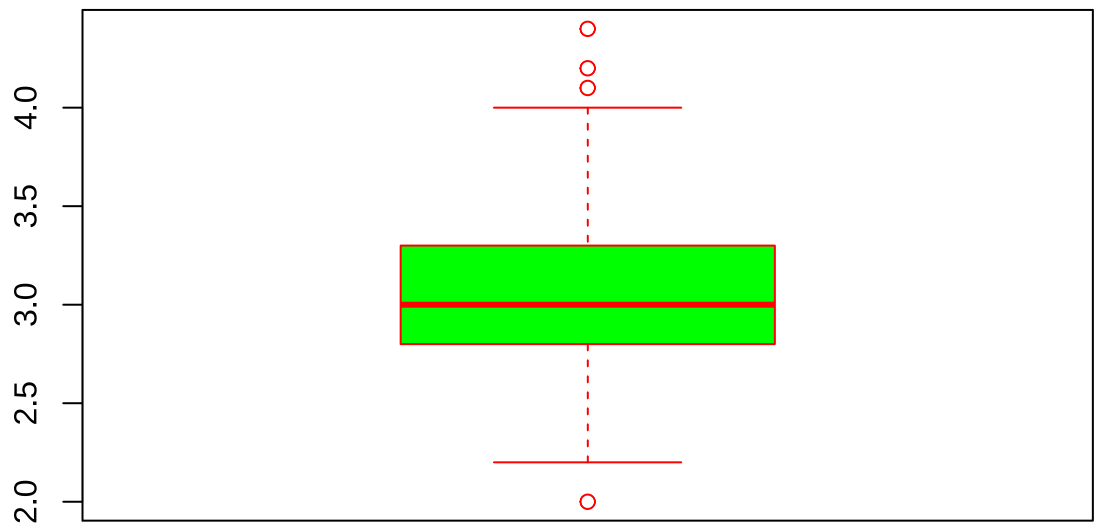
```
# produces the scatter plot of the variable
plot(iris$Sepal.Width)
```



```
# hist command produces a histogram
hist(iris$Petal.Width)
```



```
# boxplot command produces a box-plot
boxplot(iris$Sepal.Width,col='green',border='red')
```



There are a variety of arguments that can be passed to plotting functions. Try '?plot' to know these arguments

```
Work out #10
Now that we have created plots,
    1. Add a title `Sepal vs Petal width` to the plot `plot(iris$Sepal.Width, iris$Petal.Width)`
    2. Change the scatterplot to line plot in `plot(iris$Sepal.Width)`
    3. Can you break down the histogram plot into only 5 categories?
```

Low-level plotting commands

Low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot. Some of the more useful low-level plotting functions are:

points(x,y)

lines(x,y)

abline(a,b)

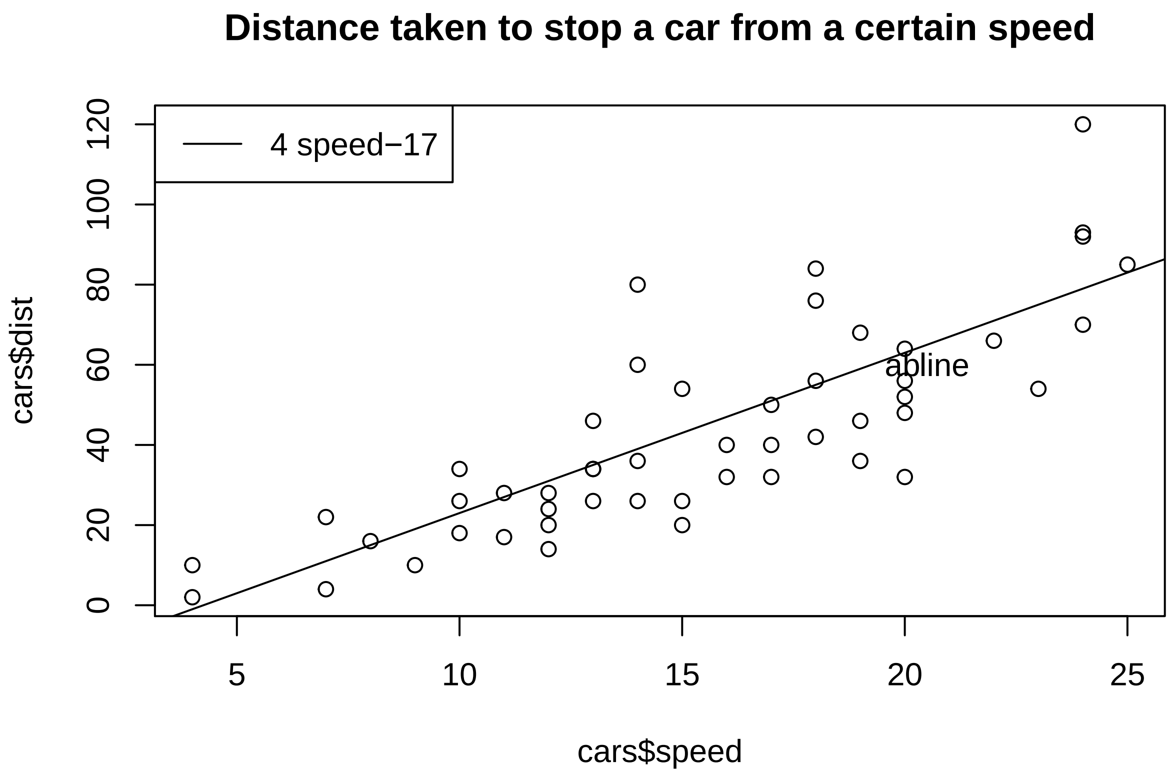
title(main=main,sub=sub)

legend(x, y, legend, . . .)

Let us see some example and for this example we shall use another inbuilt data - cars

```
plot(cars$speed,cars$dist)

# Creates a legend in the given plot
```

```r
legend("topleft",legend="4 speed-17",lty="solid")

# Creates an abline that passes through the points
abline(-17,4)

# creates a title in the plot
title(main="Distance taken to stop a car from a certain speed")

# Adds a text near the abline
text(20.5,60,"abline", cex=1)
```



Loops and functions

Let us consider an problem where we take first 4 columns (sepal's, petal's width and length) and convert those values into a normalized new variable that holds the difference between original value and mean of that column. Let us approach this problem step by step.

Step 1 - Build a function that returns absolute difference between the value provided and mean of the column

When certain operations are used multiple times we make use of user defined functions. When functions are called and provided input values, they return the required output. For instance 'Sum' is an inbuilt function. When we use sum(3,4), we are calling the function and providing it with values 3 and 4 for which the function returns with their sum. Let us now build a function which when provided a particular row number and column number, returns the absolute difference between the value in that place and mean of that column.

```
abs_diff <- function(row_no,col_no)
                {
                    abs(iris[row_no,col_no] - mean(iris[,col_no]))
                }
```

Here **abs_diff** is the name of the function. The 2 parameters ** row_no & col_no ** are 2 required input by the function

abs_diff(3,4) will call this function and assign 3 to row_no and 4 to col_no. Then the block of code within { and } is executed and resulting value returned.

Here mean is caluclated every single time the function is called, but can you recreate the code such that mean function is used only once?

Step 2 Consider column 1 and loop through each rows of that column and obtain the absolute difference.

```
for ( i in 1:nrow(iris))
        {
            iris$Sepal.Length_norm[i] <- abs_diff(i,1)
        }
```

This above block of code moves through each row of 1st column (Sepal.Length) and obtain the absolute difference and store it in another column (Sepal.Length_norm)

nrow(iris) returns number of rows in the iris data frame and 1:nrow(iris) generates a vector 1:150. In the code 'for (i in 1:nrow(iris))' i is looped through each value of 1:150 and for each value of i, the code block between { and } is executed.

Step 3 Without for function to loop through rows In R one need not use for function to loop through rows of a column. The above example is only for illustration purpose and seldom used in practice. Even without any 'for' loops, R will loop through the vectors to perform any operation. So that above code simplifies to:

```
iris$Sepal.Length_norm <- abs_diff(1:nrow(iris),1)
```

In the above example R automatically loops through each row.

Step 4 Looping through all 4 columns We could add another loop to loop through the 4 columns, but how do we change the column names of the new column in each loop? This we could achieve by appending _norm to each column name and using that as names of newly created column. The below code does that. 'paste' is another helpful function in such scenarios, we suggest exploring what paste does.

for (j in colnames(iris)[1:4]) { iris[,paste(j,"norm",sep = "_")] <- abs_diff(1:nrow(iris),j) }

apply functions

In R it is not advisable to use loop functions. As seen earlier, rows can be manipulated without having to loop through, while for columns there are family of functions called 'apply' which can loop through column.

To find the mean of each column, we shall use sapply

```
lapply(iris[,1:4],mean)
```

```
## $Sepal.Length
## [1] 5.843
##
```

```
## $Sepal.Width
## [1] 3.057
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199
```

```r
sapply(iris[,1:4],mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.843        3.057        3.758        1.199
```

```r
apply(iris[,1:4],2,mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.843        3.057        3.758        1.199
```

```r
apply(iris[,1:4],1,mean)
```

```
##   [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700
##  [12] 2.500 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
##  [23] 2.350 2.650 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725
##  [34] 2.825 2.425 2.400 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675
##  [45] 2.800 2.375 2.675 2.350 2.675 2.475 4.075 3.900 4.100 3.275 3.850
##  [56] 3.575 3.975 2.900 3.850 3.300 2.875 3.650 3.300 3.775 3.350 3.900
##  [67] 3.650 3.400 3.600 3.275 3.925 3.550 3.800 3.700 3.725 3.850 3.950
##  [78] 4.100 3.725 3.200 3.200 3.150 3.400 3.850 3.600 3.875 4.000 3.575
##  [89] 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525 3.525 3.675 2.925
## [100] 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575 4.200 4.850
## [111] 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675 4.525
## [122] 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875
## [144] 4.550 4.550 4.300 3.925 4.175 4.325 3.950
```

```r
by(iris$Sepal.Length,iris$Species,mean)
```

```
## iris$Species: setosa
## [1] 5.006
## --------------------------------------------------------
## iris$Species: versicolor
## [1] 5.936
## --------------------------------------------------------
## iris$Species: virginica
## [1] 6.588
```

In the above function lapply loops through each columnn and obtains their means. sapply does exactly same as lapply but simplifies the results as you can see.

apply function is similar to sapply, but it can be applied row wise as well as column wise. For row wise the 2nd parameter is given as 1 and for column wise 2 is given.

The last function ‘by’ is similar to sapply except that the results can be grouped by another column. Suppose you need the mean of Sepal length across different species, this is the function you will need.

There are many other functions in this family namely mapply, tapply. You could try exploring these functions too.

```
Work out #11
Now that we have created plots,
    1. Can you write a function that takes dataframe as input and returns names of column that has at le
    2. Find which of the first 4 column in iris data has maximum mean
```