|  |  |
| --- | --- |
| LBRC Logo(1).jpg | **LAKIREDDY BALI REDDY COLLEGE OF ENGINEERING (AUTONOMOUS)**  L.B. REDDY NAGAR, MYLAVARAM – 521230. A.P. INDIA  Affiliated to JNTUK Kakinada & Approved by AICTE, New Delhi  Accredited By NAAC, Accredited By NBA Tier-I & Certified by ISO 9001:2015  http://www.lbrce.ac.in, Phone: 08659 – 222933, Fax: 08659 – 222931 Extn:109  **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING** |

**R-Programming Lab Manual**

**Week 1**

a) Installing R and RStudio

b) Basic functionality of R, variable, data types in R

**1. a) Installing R and RStudio**

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.

This programming language was named **R**, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka)

R is often used for statistical computing and graphical presentation to analyze and visualize data.

## **Why Use R?**

* It is a great resource for data analysis, data visualization, data science and machine learning
* It provides many statistical techniques (such as statistical tests, classification, clustering and data reduction)
* It is easy to draw graphs in R, like pie charts, histograms, box plot, scatter plot, etc++
* It works on different platforms (Windows, Mac, Linux)
* It is open-source and free
* It has a large community support
* It has many packages (libraries of functions) that can be used to solve different problems

**To Install R and R Packages**

* 1. Open an internet browser and go to [www.r-project.org](http://www.r-project.org/).
  2. Click the "download R" link in the middle of the page under "Getting Started."
  3. Select a CRAN location (a mirror site) and click the corresponding link.
  4. Click on the "Download R for WINDOWS" link at the top of the page.
  5. Click on the file containing the latest version of R under "Files."
  6. Save the .pkg file, double-click it to open, and follow the installation instructions.
  7. Now that R is installed, you need to download and install RStudio.

**To Install RStudio**

* 1. Go to [www.rstudio.com](http://www.rstudio.com/) and click on the "Download RStudio" button.
  2. Click on "Download RStudio Desktop."
  3. Click on the version recommended for your system, or the latest Mac version, save the .dmg file on your computer, double-click it to open, and then drag and drop it to your applications folder.

**1.b) Basic functionality of R, variable, data types in R**

If you type 5 + 5, and press enter, you will see that R outputs 10.

### Example

5 + 5

**Output:**

[1] 10

# R Syntax

## **Syntax**

To output text in R, use single or double quotes:

### Example

"Hello World!"

To output numbers, just type the number (without quotes):

### Example

5  
10  
25

To do simple calculations, add numbers together:

### Example

5 + 5

# R Print

## **Print**

Unlike many other programming languages, you can output code in R without using a print function:

### Example

"Hello World!"

However, R does have a print() function available if you want to use it. This might be useful if you are familiar with other programming languages, such as Python, which often uses the print() function to output code.

### Example

print("Hello World!")

And there are times you must use the print() function to output code, for example when working with [for](https://www.w3schools.com/r/r_for_loop.asp) loops (which you will learn more about in a later chapter):

### Example

for (x in 1:10) {  
  print(x)  
}

# R Comments

## **Comments**

Comments can be used to explain R code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments starts with a #. When executing the R-code, R will ignore anything that starts with #.

This example uses a comment before a line of code:

### Example

# This is a comment  
"Hello World!"

This example uses a comment at the end of a line of code:

### Example

"Hello World!" # This is a comment

## **Multiline Comments**

Unlike other programming languages, such as [Java](https://www.w3schools.com/java/java_comments.asp), there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:

### Example

# This is a comment  
# written in  
# more than just one line  
"Hello World!"

# R Variables

## **Creating Variables in R**

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <-sign. To output (or print) the variable value, just type the variable name:

### Example

name <- "John"  
age <- 40  
name   # output "John"  
age    # output 40

From the example above, name and age are **variables**, while "John" and 40 are **values**.

In other programming language, it is common to use = as an assignment operator. In R, we can use both = and <- as assignment operators.

However, <- is preferred in most cases because the = operator can be forbidden in some context in R.

## **Print / Output Variables**

Compared to many other programming languages, you do not have to use a function to print/output variables in R. You can just type the name of the variable:

### Example

name <- "John Doe"  
name # auto-print the value of the name variable

However, R does have a print() function available if you want to use it. This might be useful if you are familiar with other programming languages, such as Python, which often use a print() function to output variables.

### Example

name <- "John Doe"  
print(name) # print the value of the name variable

And there are times you must use the print() function to output code, for example when working with for loops (which you will learn more about in a later chapter):

### Example

for (x in 1:10) {  
  print(x)  
}

**Output:**

[1] 1

[1] 2

[1] 3

[1] 4

[1] 5

[1] 6

[1] 7

[1] 8

[1] 9

[1] 10

## **Multiple Variables**

R allows you to assign the same value to multiple variables in one line:

### Example

# Assign the same value to multiple variables in one line  
var1 <- var2 <- var3 <- "Orange"  
  
# Print variable values  
var1  
var2  
var3

## **Variable Names**

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total\_volume).

Rules for R variables are:

* A variable name must start with a letter and can be a combination of letters, digits, period(.)and underscore(\_). If it starts with period(.), it cannot be followed by a digit.
* A variable name cannot start with a number or underscore (\_)
* Variable names are case-sensitive (age, Age and AGE are three different variables)
* Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

# Legal variable names:  
myvar<- "John"  
my\_var<- "John"  
myVar <- "John"  
MYVAR <- "John"  
myvar2 <- "John"  
.myvar<- "John"  
  
# Illegal variable names:  
2myvar <- "John"  
my-var <- "John"  
my var <- "John"  
\_my\_var<- "John"  
my\_v@ar<- "John"  
TRUE <- "John"

Remember that variable names are case-sensitive!

## **Data Types**

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

In R, variables do not need to be declared with any particular type, and can even change type after they have been set:

### Example

my\_var<- 30 # my\_var is type of **numeric**

**my\_var**

**Output:**

[1] 30

my\_var<- "Sally" # my\_var is now of type **character** (aka string)

**my\_var**

**Output:**

[1] "Sally"

**R has a variety of data types and object classes.**

## **Basic Data Types**

Basic data types in R can be divided into the following types:

* numeric - (10.5, 55, 787)
* integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)
* complex - (9 + 3i, where "i" is the imaginary part)
* character (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
* logical (a.k.a. boolean) - (TRUE or FALSE)

Use the **class()** function to check the data type of a variable:

### Example

# numeric  
x <- 10.5  
class(x)  
**Output:**

[1] "numeric"

# integer  
x <- 1000L  
class(x)  
**Output:**

[1] "integer"

# complex  
x <- 9i + 3  
class(x)  
**Output:**

**[1] "complex"**

# character/string  
x <- "R is exciting"  
class(x)  
**Output:**

# logical/boolean  
x <- TRUE  
class(x)

**Output:**

[1] ”logical”

# R Numbers

## **Numbers**

There are three number types in R:

* numeric
* integer
* complex

Variables of number types are created when you assign a value to them:

### Example

x <- 10.5   # numeric  
y <- 10L    # integer  
z <- 1i     # complex

**Output:**

> x

[1] 10.5

> y

[1] 10

> z

[1] 0+1i

## **Numeric**

A numeric data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

### Example

x <- 10.5  
y <- 55  
# Print values of x and y  
x  
y  
**Output:**

> x

[1] 10.5

> y

[1] 55  
# Print the class name of x and y  
class(x)  
class(y)

**Output:**

> class(x)

[1] "numeric"

> class(y)

[1] "numeric"

## **Integer**

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an integer variable, you must use the letter L after the integer value:

### Example

x <- 1000L  
y <- 55L  
# Print values of x and y  
x  
y  
**Output:**

> x

[1] 1000

> y

[1] 55

# Print the class name of x and y  
class(x)  
class(y)

**Output:**

> class(x)

[1] "integer"

> class(y)

[1] "integer"

## **Complex**

A complex number is written with an "i" as the imaginary part:

### Example

x <- 3+5i  
y <- 5i  
# Print values of x and y  
x  
y

**Output:**

> x

[1] 3+5i

> y

[1] 0+5i

# Print the class name of x and y  
class(x)  
class(y)

**Output:**

>class(x)

[1] "complex"

> class(y)

[1] "complex"

## **Type Conversion**

You can convert from one type to another with the following functions:

* as.numeric()
* as.integer()
* as.complex()

### Example

x <- 1L # integer  
y <- 2 # numeric  
# convert from integer to numeric:  
a <- as.numeric(x)

# convert from numeric to integer:  
b <- as.integer(y)  
# print values of x and y  
x  
y  
# print the class name of a and b  
class(a)  
class(b)

**Output:**

print values of x and y

> x

[1] 1

> y

[1] 2

# print the class name of a and b

> class(a)

[1] "numeric"

> class(b)

[1] "integer"

**Week 2**

a) Implement R script to show the usage of various operators available in R language.

b) Implement R script to read person’s age from keyboard and display whether he is

eligible for voting or not.

c) Implement R script to find biggest number between two numbers.

d) Implement R script to check the given year is leap year or not.

**2(a) Implement R script to show the usage of various operators available in R language.**

**R Script:**

a=40

b=20

print("Arthimetic Operators")

print(paste("addition=",(a+b)))

print(paste("subtraction =",a-b))

print(paste("multiplication=",a\*b))

print(paste("division of numbers",a/b))

print(paste("modulo of numbers",a%%b))

print(paste("Quotient of number",a%/%b))

print(paste("power of number=",a^b))

print("Relational Operators")

print(paste("Checks Greater:",a>b))

print(paste("Checks lessthan:",a<b))

print(paste("Checks equal to:",a==b))

print(paste("Checks Greater or equal to:",a>=b))

print(paste("Checks less than or equal to:",a<=b))

print(paste("Checks not equal or not:",a!=b))

print("Logical operators")

print(paste("And operation",a&b))

print(paste("OR operation",a|b))

print(paste("NOT operation of a",!a))

print(paste("NOT operation of b",!b))

print(paste("Logical And operation",a&&b))

print(paste("Logical OR operation",a||b))

print("Miscellaneous Operators")

print("Colon operator")

print(2:8)

**Output:**

[1] "Arthimetic Operators"

[1] "addition= 60"

[1] "subtraction = 20"

[1] "multiplication= 800"

[1] "division of numbers 2"

[1] "modulo of numbers 0"

[1] "Quotient of number 2"

[1] "power of number= 1.099511627776e+32"

[1] "Relational Operators"

[1] "Checks Greater: TRUE"

[1] "Checks lessthan: FALSE"

[1] "Checks equal to: FALSE"

[1] "Checks Greater or equal to: TRUE"

[1] "Checks less than or equal to: FALSE"

[1] "Checks not equal or not: TRUE"

[1] "Logical operators"

[1] "And operation TRUE"

[1] "OR operation TRUE"

[1] "NOT operation of a FALSE"

[1] "NOT operation of b FALSE"

[1] "Logical And operation TRUE"

[1] "Logical OR operation TRUE"

[1] "Miscellaneous Operators"

[1] "Colon operator"

[1] 2 3 4 5 6 7 8

**2( b) Implement R script to read person‘s age from keyboard and display whether he is**

**eligible for voting or not.**

age = readline(prompt="Enter the Age: ")

age = as.integer(age)

if(age>=18)

{

print(paste("Eligible to vote",age))

}else

{

print(paste("Not Eligible to vote",age))

}

**Output:**

Enter the Age: 21

[1] "Eligible to vote 21"

Enter the Age: 17

[1] "Not Eligible to vote 17"

**2(c) Implement R script to find biggest number between two numbers.**

# To Implement R script to find boggest number between two numbers

a = as.integer(readline(prompt = "Enter the Number 1: "))

b = as.integer(readline(prompt = "Enter the Number 2: "))

if(a>b)

{

sprintf("a value %d is big", a)

}else

{

sprintf(" value %d is big", b)

}

**Output:**

Enter the Number 1: 10

Enter the Number 2: 5

[1] "a value 10 is big"

**2(d) Implement R script to check the given year is leap year or not.**

## **ALGORITHM**

**STEP 1**: prompting appropriate messages to the user

**STEP 2**: take user input using readline() into variables **year**

**STEP 3**: check if year is exactly divisible by **4,100,400** gives a remainder

of **0**

**STEP 4**: if remainder is a non-zero print year is not a leap year.

**STEP 5**: if remainder is zero print year is a leap year.

# Program to check if the input year is a leap year or not

year = as.integer(readline(prompt ="Enter a year: "))

if((year %% 4) == 0) {

if((year %% 100) == 0) {

if((year %% 400) == 0) {

print(paste(year," is a Leap Year"))

} else {

print(paste(year," is not a Leap Year"))

}

} else {

print(paste(year," is a Leap Year"))

}

} else {

print(paste(year," is not a Leap Year"))

}

**Output:**

Enter a year: 1900

[1] "1900 is not a leap year"

Enter a year: 2000

[1] "2000 is a leap year"

**Week-3**

a) Implement R Script to create a list.

b) Implement R Script to access elements in the list.

c) Implement R Script to merge two or more lists.

d) Implement R Script to perform matrix operation.

**3(a) Implement R Script to create a list.**

Lists are the R objects which contain elements of different types like − numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

## **Creating a List**

Following is an example to create a list containing strings, numbers, vectors and a logical values.

# Create a list containing strings, numbers, vectors and logical values

list\_data<- list("Red","Green",c(21,32,11), TRUE, 51.23, 119.1)

print(list\_data)

print(class(list\_data))

**Output:**

print(list\_data)

[[1]]

[1] "Red"

[[2]]

[1] "Green"

[[3]]

[1] 21 32 11

[[4]]

[1] TRUE

[[5]]

[1] 51.23

[[6]]

[1] 119.1

**3(b) Implement R Script to access elements in the list.**

## **Giving a name to list elements**

There are only three steps to print the list data corresponding to the name:

1. Creating a list.
2. Assign a name to the list elements with the help of names() function.
3. Print the list data.

**Example: 1**

**# Create a list containing a vector, a matrix and a list.**

list\_data<- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),list("green",12.3))

**# Give names to the elements in the list.**

names(list\_data) <- c("1st Quarter", "A\_Matrix", "A Inner list")

**# Show the list.**

print(list\_data)

**Output:**

print(list\_data)

$`1st Quarter`

[1] "Jan" "Feb" "Mar"

$A\_Matrix

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

[1,] 3 5 -2

[2,] 9 1 8

$`A Inner list`

$`A Inner list`[[1]]

[1] "green"

$`A Inner list`[[2]]

[1] 12.3

## **Accessing List Elements**

* Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

**Example**:2

**# Create a list containing a vector, a matrix and a list.**

list\_data<- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))

**# Give names to the elements in the list.**

names(list\_data) <- c("1st Quarter", "A\_Matrix", "A Inner list")

**# Access the first element of the list.**

print(list\_data[1])

**# Access the thrid element. As it is also a list, all its elements will be printed.**

print(list\_data[3])

**# Access the list element using the name of the element.**

print(list\_data$A\_Matrix)

**Output:**

print(list\_data[1])

$`1st Quarter`

[1] "Jan" "Feb" "Mar"

# Access the third element. As it is also a list, all its elements will be printed.

print(list\_data[3])

$`A Inner list`

$`A Inner list`[[1]]

[1] "green"

$`A Inner list`[[2]]

[1] 12.3

# Access the list element using the name of the element.

print(list\_data$A\_Matrix)

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

[1,] 3 5 -2

[2,] 91 8

**3(c) Implement R Script to merge two or more lists.**

**Implement R Script to perform matrix operation.**

**Implement R Script to merge two or more lists.**

## **Merging Lists**

You can merge many lists into one list by placing all the lists inside one list() function.

**# Create two lists.**

list1 <- list(1,2,3)

list2 <- list("Sun","Mon","Tue")

**# Merge the two lists.**

merged.list<- c(list1,list2)

**# Print the merged list.**

print(merged.list)

**Output:**

print(merged.list)

[[1]]

[1] 1

[[2]]

[1] 2

[[3]]

[1] 3

[[4]]

[1] "Sun"

[[5]]

[1] "Mon"

[[6]]

[1] "Tue"

**3 (d) Implement R Script to perform matrix operation.**

# R Matrix

In R, a two-dimensional rectangular data set is known as a matrix. A matrix is created with the help of the vector input to the matrix function. On R matrices, we can perform addition, subtraction, multiplication, and division operation.

In the R matrix, elements are arranged in a fixed number of rows and columns. The matrix elements are the real numbers.

A Matrix is created using the **matrix()** function.

### Syntax

matrix(data, nrow, ncol, byrow, dimnames)

Following is the description of the parameters used −

* **data** is the input vector which becomes the data elements of the matrix.
* **nrow** is the number of rows to be created.
* **ncol** is the number of columns to be created.
* **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
* **dimname** is the names assigned to the rows and columns.

**Example**

**#Arranging elements sequentially by row.**

P <- matrix(c(5:16), nrow = 4, byrow = TRUE)

print(P)

**# Arranging elements sequentially by column.**

Q <- matrix(c(3:14), nrow = 4, byrow = FALSE)

print(Q)

**# Defining the column and row names.**

row\_names = c("row1", "row2", "row3", "row4")

col\_names = c("col1", "col2", "col3")

R <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(row\_names, col\_names))

print(R)

**Output:**

print(P)

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

[1,] 5 6 7

[2,] 8 9 10

[3,] 11 12 13

[4,] 14 15 16

print(Q)

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

[1,] 3 7 11

[2,] 4 8 12

[3,] 5 9 13

[4,] 6 10 14

print(R)

col1 col2 col3

row1 3 4 5

row2 6 7 8

row3 9 10 11

row4 12 13 14

**Accessing Elements of a Matrix**

Elements of a matrix can be accessed by using the column and row index of the element.

# Define the column and row names.

rownames = c("row1", "row2", "row3", "row4")

colnames = c("col1", "col2", "col3")

**# Create the matrix. Arranging elements sequentially by row.**

P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))

print(P)

**# Access the element at 3rd column and 1st row.**

print(P[1,3])

**# Access the element at 2nd column and 4th row.**

print(P[4,2])

**# Access only the 2nd row.**

print(P[2,])

**# Access only the 3rd column.**

print(P[,3])

**Output:**

print(P)

col1 col2 col3

row1 3 4 5

row2 6 7 8

row3 9 10 11

row4 12 13 14

print(P[1,3])

[1] 5

print(P[4,2])

[1] 13

print(P[2,])

col1 col2 col3

6 7 8

print(P[,3])

row1 row2 row3 row4

5 8 11 14

## **Matrix operations**

In R, we can perform the mathematical operations on a matrix such as addition, subtraction, multiplication, etc.

R <- matrix(c(5:16), nrow = 4,ncol=3)

S <- matrix(c(1:12), nrow = 4,ncol=3)

**# Display two matrices R and S**

print(R)

print(S)

**#Addition**

sum<-R+S

print(sum)

**#Subtraction**

sub<-R-S

print(sub)

**#Multiplication**

mul<-R\*S

print(mul)

**#Division**

div<-R/S

print(div)

**Output:**

**print(R)**

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

[1,] 5 9 13

[2,] 6 10 14

[3,] 7 11 15

[4,] 8 12 16

**print(S)**

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

[1,] 1 5 9

[2,] 2 6 10

[3,] 3 7 11

[4,] 4 8 12

**sum<-R+S**

**print(sum)**

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

[1,] 6 14 22

[2,] 8 16 24

[3,] 10 18 26

[4,] 12 20 28

**sub<-R-S**

print(sub)

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

[1,] 4 4 4

[2,] 4 4 4

[3,] 4 4 4

[4,] 4 4 4

**mul<-R\*S**

print(mul)

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

[1,] 5 45 117

[2,] 12 60 140

[3,] 21 77 165

[4,] 32 96 192

**div<-R/S**

print(div)

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

[1,] 5.000000 1.800000 1.444444

[2,] 3.000000 1.666667 1.400000

[3,] 2.333333 1.571429 1.363636

[4,] 2.000000 1.500000 1.333333

**Week-4**

**Implement R script to perform following operations:**

a) Various operations on vectors.

b) Finding the sum and average of given numbers using arrays.

c) To display elements of list in reverse order.

d) Finding the minimum and maximum elements in the array.

**4(a) Implement R script to perform various operations on vectors.**

* In R, a sequence of elements which share the same data type is known as vector.
* A vector supports logical, integer, double, character, complex, or raw data type.
* The elements which are contained in vector known as **components** of the vector.
* We can check the type of vector with the help of the **typeof()** function.

The length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.

Vector is classified into two parts, i.e., **Atomic vectors** and **Lists**. They have three common properties, i.e., **function type, function length**, and **attribute function**.

The length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.

Vector is classified into two parts, i.e., **Atomic vectors** and **Lists**. They have three common properties, i.e., **function type, function length**, and **attribute function**.

## **How to create a vector in R?**

* In R, we use c() function to create a vector. This function returns a one-dimensional array or simply vector.
* The c() function is a generic function which combines its argument. All arguments are restricted with a common data type which is the type of the returned value.

**There are various other ways to create a vector in R, which are as follows:**

### 1) Using the colon(:) operator

We can create a vector with the help of the colon operator. There is the following syntax to use colon operator:

1. z**<-x:y**

This operator creates a vector with elements from x to y and assigns it to z.

**Example:**

A **<- 4:-10**

a

**Output**

[1] 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10

### 2) Using the seq() function

In R, we can create a vector with the help of the seq() function. A sequence function creates a sequence of elements as a vector. The seq() function is used in two ways, i.e., by setting step size with ?by' parameter or specifying the length of the vector with the 'length.out' feature.

### Example

numbers <- seq(from = 0, to = 100, by = 20)  
numbers

**Output:**

[1] 0 20 40 60 80 100

**Note:** The seq() function has three parameters: from is where the sequence starts, to is where the sequence stops, and by is the interval of the sequence.

## **Atomic vectors in R**

In R, there are four types of atomic vectors. Atomic vectors play an important role in Data Science. Atomic vectors are created with the help of **c()** function. These atomic vectors are as follows:

### Example

**# Vector of strings**  
fruits <- c("banana", "apple", "orange")  
# Print fruits  
fruits

# Output:

# [1] "banana" "apple" "orange"

In this example, we create a vector that combines numerical values:

### Example

**# Vector of numerical values**  
numbers <- c(1, 2, 3)  
# Print numbers  
numbers

# Output:

# [1] 1 2 3

To create a vector with numerical values in a sequence, use the : operator:

### Example

**# Vector with numerical values in a sequence**  
numbers <- 1:10  
numbers

# Output:

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

## **Atomic vectors in R**

In R, there are four types of atomic vectors. Atomic vectors play an important role in Data Science. Atomic vectors are created with the help of **c()** function. These atomic vectors are as follows:

### 1. Numeric vector

The decimal values are known as numeric data types in R. If we assign a decimal value to any variable d, then this d variable will become a numeric type. A vector which contains numeric elements is known as a numeric vector.

**Example:**

d**<-** 45.5

num\_vec<-c(10.1, 10.2, 33.2)

d

num\_vec

class(d)

class(num\_vec)

**Output:**

[1] 10.1 10.2 33.2

[1] "numeric"

[1] "numeric"

### 2. Integer vector

A non-fraction numeric value is known as integer data. This integer data is represented by "Int." The Int size is 2 bytes and long Int size of 4 bytes. There is two way to assign an integer value to a variable, i.e., by using as.integer() function and appending of L to the value.

A vector which contains integer elements is known as an integer vector.

**Example:**

d<-as.integer(5)

e<-5L

int\_vec<-c(1,2,3,4,5)

int\_vec<-as.integer(int\_vec)

int\_vec1<-c(1L,2L,3L,4L,5L)

class(d)

class(e)

class(int\_vec)

class(int\_vec1)

**Output:**

[1] "integer"

[1] "integer"

[1] "integer"

[1] "integer"

### 3. Character vector

A character is held as a one-byte integer in memory. In R, there are two different ways to create a character data type value, i.e., using as.character() function and by typing string between double quotes("") or single quotes('').

A vector which contains character elements is known as an integer vector.

**Example:**

d**<-**'shubham'

e**<-**"Arpita"

f**<-65**

f**<-as.character**(f)

d

e

f

char\_vec**<-c**(1,2,3,4,5)

char\_vec**<-as.character**(char\_vec)

char\_vec1**<-c**("shubham","arpita","nishka","vaishali")

char\_vec

class(d)

class(e)

class(f)

class(char\_vec)

class(char\_vec1)

**Output:**

> d

[1] "shubham"

> e

[1] "Arpita"

> f

[1] "65"

>char\_vec

[1] "1" "2" "3" "4" "5"

> class(d)

[1] "character"

> class(e)

[1] "character"

> class(f)

[1] "character"

> class(char\_vec)

[1] "character"

> class(char\_vec1)

[1] "character"

## **Accessing elements of vectors**

We can access the elements of a vector with the help of vector indexing. Indexing denotes the position where the value in a vector is stored. Indexing will be performed with the help of integer, character, or logic.

### 1) Indexing with integer vector

On integer vector, indexing is performed in the same way as we have applied in C, C++, and java. There is only one difference, i.e., in C, C++, and java the indexing starts from 0, but in R, the indexing starts from 1. Like other programming languages, we perform indexing by specifying an integer value in square braces [] next to our vector.

**Example:**

seq\_vec**<-seq**(1,4,length.out=6)

seq\_vec

seq\_vec[2]

**Output**

[1] 1.0 1.6 2.2 2.8 3.4 4.0

[1] 1.6

### 2) Indexing with a character vector

In character vector indexing, we assign a unique key to each element of the vector. These keys are uniquely defined as each element and can be accessed very easily. Let's see an example to understand how it is performed.

**Example:**

char\_vec**<-c**("shubham"=22,"arpita"=23,"vaishali"=25)

char\_vec

char\_vec["arpita"]

**Output**

shubhamarpitavaishali

22 23 25

arpita

23

### 3) Indexing with a logical vector

In logical indexing, it returns the values of those positions whose corresponding position has a logical vector TRUE. Let see an example to understand how it is performed on vectors.

**Example:**

a**<-c**(1,2,3,4,5,6)

a[c(TRUE, FALSE,TRUE,TRUE,FALSE,TRUE)]

**Output**

[1] 1 3 4 6

## **Vector Operation**

In R, there are various operation which is performed on the vector. We can add, subtract, multiply or divide two or more vectors from each other.

### 1) Combining vectors

The c() function is not only used to create a vector, but also it is also used to combine two vectors. By combining one or more vectors, it forms a new vector which contains all the elements of each vector. Let see an example to see how c() function combines the vectors.

**Example:**

p <- c(1,2,3,5,7,8)

q <- c("subbu","raju","raju","sankar","rajesh","ramesh")

r <- c(p,q)

r

**Output:**

[1] "1" "2" "3" "5" "7" "8" "subbu" "raju" "raju" "sankar" "rajesh" "ramesh"

### 2) Arithmetic operations

We can perform all the arithmetic operation on vectors. The arithmetic operations are performed member-by-member on vectors. We can add, subtract, multiply, or divide two vectors. Let see an example to understand how arithmetic operations are performed on vectors.

**Example:**

a<-c(1,3,5,7)

b<-c(2,4,6,8)

print("Addition of a+b")

a+b

print("Subtraction of a-b")

a-b

print("Division of a/b")

a/b

print("Modolus of a%%b")

a%%b

**Output:**

[1] "Addition of a+b"

>a+b

[1] 3 7 11 15

[1] "Subtraction of a-b"

> a-b

[1] -1 -1 -1 -1

[1] "Division of a/b"

> a/b

[1] 0.5000000 0.7500000 0.8333333 0.8750000

[1] "Modolus of a%%b"

> a%%b

[1] 1 3 5 7

### Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

v1 <- c(3,8,4,5,0,11)

v2 <- c(4,11)

# V2 becomes c(4,11,4,11,4,11)

add.result<- v1+v2

print(add.result)

sub.result<- v1-v2

print(sub.result)

**Output:**

[1] 7 19 8 16 4 22

[1] -1 -3 0 -6 -4 0

### Vector Element Sorting

Elements in a vector can be sorted using the **sort()** function.

v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.

sort.result<- sort(v)

print(sort.result)

# Sort the elements in the reverse order.

revsort.result<- sort(v, decreasing = TRUE)

print(revsort.result)

# Sorting character vectors.

v <- c("Red","Blue","yellow","violet")

sort.result<- sort(v)

print(sort.result)

# Sorting character vectors in reverse order.

revsort.result<- sort(v, decreasing = TRUE)

print(revsort.result)

**Output:**

print(sort.result)

[1] -9 0 3 4 5 8 11 304

print(revsort.result)

[1] 304 11 8 5 4 3 0 -9

print(sort.result)

[1] "Blue" "Red" "violet" "yellow"

print(revsort.result)

[1] "yellow" "violet" "Red" "Blue"

**4(b) Implement R script for finding the sum and average of given numbers using arrays.**

thisarray <- c(1:24)

multiarray <- array(thisarray,dim = c(4,3,2))

print(multiarray)

print("sum of array elements:")

print(sum(multiarray))

print("Length of the array:")

len<-length(multiarray)

len

print("Average of array elements:")

print(sum(multiarray)/len)

**Output:**

[1] "sum of array elements:"

[1] 300

[1] "Length of the array:"

[1] 24

[1] "Average of array elements:"

[1] 12.5

**4(c) Implement R script to display elements of list in reverse order.**

list1 <-c(1:24)

print(list1)

print(Elements in Reverse Order")

rev.default(list1)

**Output:**

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

[1] "Elements in Reverse Order"

[1] 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

**4(d) Implement R script to find the minimum and maximum elements in the array.**

nums = c(10, 20, 30, 40, 50, 60)

array1 <-array(nums)

print("The elements in the Array:")

array1

print(paste("Maximum value :",max(nums)))

print(paste("Minimum value :",min(nums)))

**Output:**

[1] "The elements in the Array:"

[1] 10 20 30 40 50 60

[1] "Maximum value : 60"

[1] "Minimum value : 10"

**Week-5**

a) Implement R script to perform various operations on matrices.

b) Implement R script to extract the data from dataframes.

c) Write R script to display file contents.

d) Write R script to copy file contents from one to another.

**5(a) a) Implement R script to perform various operations on matrices.**

## **Matrix operations**

In R, we can perform the mathematical operations on a matrix such as addition, subtraction, multiplication, etc.

R <- matrix(c(5:16), nrow = 4,ncol=3)

S <- matrix(c(1:12), nrow = 4,ncol=3)

**# Display two matrices R and S**

print(R)

print(S)

**#Addition**

sum<-R+S

print(sum)

**#Subtraction**

sub<-R-S

print(sub)

**#Multiplication**

mul<-R\*S

print(mul)

**#Division**

div<-R/S

print(div)

**Output:**

**print(R)**

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

[1,] 5 9 13

[2,] 6 10 14

[3,] 7 11 15

[4,] 8 12 16

**print(S)**

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

[1,] 1 5 9

[2,] 2 6 10

[3,] 3 7 11

[4,] 4 8 12

**sum<-R+S**

**print(sum)**

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

[1,] 6 14 22

[2,] 8 16 24

[3,] 10 18 26

[4,] 12 20 28

**sub<-R-S**

print(sub)

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

[1,] 4 4 4

[2,] 4 4 4

[3,] 4 4 4

[4,] 4 4 4

**mul<-R\*S**

print(mul)

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

[1,] 5 45 117

[2,] 12 60 140

[3,] 21 77 165

[4,] 32 96 192

**div<-R/S**

print(div)

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

[1,] 5.000000 1.800000 1.444444

[2,] 3.000000 1.666667 1.400000

[3,] 2.333333 1.571429 1.363636

[4,] 2.000000 1.500000 1.333333

**5(b) Implement R script to extract the data from dataframes.**

exam\_data=data.frame(

name =c('Anastasia','Dima','Katherine','James','Emily','Michael','Matthew','Laura','Kevin','Jonas'),

score =c(12.5,9,16.5,12,9,20,14.5,13.5,8,19),

attempts =c(1,3,2,3,2,3,1,1,2,1),

qualify =c('yes','no','yes','no','no','yes','yes','no','no','yes')

)

print("Original dataframe:")

print(exam\_data)

print("Extract 3rd and 5th rows with 1st and 3rd columns :")

result =exam\_data[c(3,5),c(1,3)]

print(result)

**Output:**

print(exam\_data)

name score attempts qualify

1 Anastasia12.5 1 yes

2 Dima9.0 3 no

3 Katherine16.5 2 yes

4 James12.0 3 no

5 Emily9.0 2 no

6 Michael20.0 3 yes

7 Matthew14.5 1 yes

8 Laura13.5 1 no

9 Kevin8.0 2 no

10Jonas 19.0 1 yes

[1] "Extract 3rd and 5th rows with 1st and 3rd columns:"

name attempts

3 Katherine 2

5 Emily 2

**5(c) Write R script to display file contents.**

#### **Reading a text file**

One of the important formats to store a file is in a text file. R provides various methods that one can read data from a text file.

* **read.delim()**: This method is used for reading “tab-separated value” files (“.txt”). By default, point (“.”) is used as decimal points.

*read.delim(file, header = TRUE, sep = “\t”, dec = “.”, …)*

**file:** the path to the file containing the data to be read into R.  
**header:** a logical value. If TRUE, read.delim() assumes that your file has a header row, so row 1 is the name of each column. If that’s not the case, you can add the argument header = FALSE.  
**sep:** the field separator character. “\t” is used for a tab-delimited file.  
**dec:** the character used in the file for decimal points.

* **read.delim2()**: This method is used for reading “tab-separated value” files (“.txt”). By default, point (“,”) is used as decimal points.

*read.delim2(file, header = TRUE, sep = “\t”, dec = “,”, …)*

* **file.choose()**: In R it’s also possible to choose a file interactively using the function **file.choose()**, and if you’re a beginner in R programming then this method is very useful for you.
* **read\_tsv()**: This method is also used for to read a tab separated (“\t”) values by using the help of **readr** package.

*read\_tsv(file, col\_names = TRUE)*

*file: the path to the file containing the data to be read into R.  
col\_names: Either TRUE, FALSE, or a character vector specifying column names. If TRUE, the first row of the input will be used as the column names.*

**Example:1**

# R program reading a text file

# Read a text file using read.delim()

myData = read.delim("C:/Users/rajen/OneDrive/Documents/sample.txt", header = FALSE)

print(myData)

**Output:**

1 Welcome to R Programming Lab

**Example:2**

# R program reading a text file

# Read a text file using read.delim2

myData = read.delim2("C:/Users/rajen/OneDrive/Documents/sample.txt", header = FALSE)

print(myData)

**Output:**

1 Welcome to R Programming Lab

**Example:3**

# R program reading a text file using file.choose()

myFile = read.delim(file.choose(), header = FALSE)

# If you use the code above in RStudio

# you will be asked to choose a file

print(myFile)

**Output:**

1 Welcome to R Programming Lab

**Example:4**

# R program to read text file

# usingreadr package

# Import the readr library

install.packages("readr")

library(readr)

# Use read\_tsv() to read text file

myData = read\_tsv("C:/Users/rajen/OneDrive/Documents/sample.txt",col\_names = FALSE)

print(myData)

**Output:**

A tibble: 1 x 1

X1

<chr>

1 Welcome to R Programming Lab

**5(d) Write R script to copy file contents from one to another.**

library(readr)

file1=read\_file("C:/Users/rajen/OneDrive/Documents/sample.txt")

print(file1)

write\_file(file1,"file2.txt")

d=read\_file("file2.txt")

print(d)

(OR)

library(readr)

file1=read\_file("C:/Users/Admin/Documents/sample.txt")

print(file1)

file.create("file3.txt")

write\_file(file1,"file3.txt")

d=read\_file("file3.txt")

print(d)

**Output:**

[1] "Welcome to R Programming Lab"

**Week-6**

a) Write an R script to find basic descriptive statistics using summary, str, quartile function

on mtcars datasets.

b) Write an R script to find subset of dataset by using subset(), aggregate() functions on iris dataset.

**6(a) Write an R script to find basic descriptive statistics using summary, str, quartile**

**function on mtcars& cars datasets.**

### What is Descriptive Statistics?

* Descriptive statistics is the branch of statistics that focuses on **describing** and gaining more insight into the data in its present state.
* It deals with what the data in its **current state** means. It makes the data easier to understand and also gives us knowledge about the data which is necessary to perform further analysis.
* Average measures like mean, median, mode, etc. are a good example of descriptive statistics.

## **Descriptive Statistics in R**

R programming language provides us with lots of simple yet effective functions to perform descriptive statistics and gain more knowledge about our data. Summarizing the data, calculating average measures, finding out cumulative measures, summarizing rows/columns of data structures, etc. everything is possible with trivial commands. Let’s start simple with the summarizing functions **str()** and **summary()**.

### Summarizing your Data

R provides two very simple functions that can instantly summarize our data for us. These are the **str()** and the **summary()** functions.

**str() function**

The **str()** function takes a single object as an argument and compactly shows us the **structure** of the input object. It shows us details like length, data type, names and other specifics about the components of the object.

#shows us the **structure** of the input object.

str(mtcars)

Output:

str(mtcars)

'data.frame': 32 obs. of 11 variables:

$ mpg :num 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...

$ cyl :num 6 6 4 6 8 6 8 4 4 6 ...

$ disp: num 160 160 108 258 360 ...

$ hp :num 110 110 93 110 175 105 245 62 95 123 ...

$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...

$ wt :num 2.62 2.88 2.32 3.21 3.44 ...

$ qsec: num 16.5 17 18.6 19.4 17 ...

$ vs :num 0 0 1 1 0 1 0 1 1 1 ...

$ am :num 1 1 1 0 0 0 0 0 0 0 ...

$ gear: num 4 4 4 3 3 3 3 4 4 4 ...

$ carb: num 4 4 1 1 2 1 4 2 2 4 ...

str(cars)

**Output:**

str(cars)

'data.frame': 50 obs. of 2 variables:

$ speed: num 4 4 7 7 8 9 10 10 10 11 ...

$ dist :num 2 10 4 22 16 10 18 26 34 17 ...

**summary() function:**

The **summary()** function also takes a single object as an argument. It then returns the **averages measures** like mean, median, minimum, maximum, 1st quantile, 3rd quantile, etc. for each component or variable in the object. Here is an example of the summary function in action.

summary(mtcars)

**Output:**

**summary(mtcars)**

mpg cyldisp hp drat wtqsec

Min. :10.40 Min. :4.000 Min. : 71.1 Min. : 52.0 Min. :2.760 Min. :1.513 Min. :14.50

1st Qu.:15.43 1st Qu.:4.000 1st Qu.:120.8 1st Qu.: 96.5 1st Qu.:3.080 1st Qu.:2.581 1st Qu.:16.89

Median :19.20 Median :6.000 Median :196.3 Median :123.0 Median :3.695 Median :3.325 Median :17.71

Mean :20.09 Mean :6.188 Mean :230.7 Mean :146.7 Mean :3.597 Mean :3.217 Mean :17.85

3rd Qu.:22.80 3rd Qu.:8.000 3rd Qu.:326.0 3rd Qu.:180.0 3rd Qu.:3.920 3rd Qu.:3.610 3rd Qu.:18.90

Max. :33.90 Max. :8.000 Max. :472.0 Max. :335.0 Max. :4.930 Max. :5.424 Max. :22.90

vs am gear carb

Min. :0.0000 Min. :0.0000 Min. :3.000 Min. :1.000

1st Qu.:0.0000 1st Qu.:0.0000 1st Qu.:3.000 1st Qu.:2.000

Median :0.0000 Median :0.0000 Median :4.000 Median :2.000

Mean :0.4375 Mean :0.4062 Mean :3.688 Mean :2.812

3rd Qu.:1.0000 3rd Qu.:1.0000 3rd Qu.:4.000 3rd Qu.:4.000

Max. :1.0000 Max. :1.0000 Max. :5.000 Max. :8.000

summary(cars)

**Output:**

**summary(cars)**

speed dist

Min. : 4.0 Min. : 2.00

1st Qu.:12.0 1st Qu.: 26.00

Median :15.0 Median : 36.00

Mean :15.4 Mean : 42.98

3rd Qu.:19.0 3rd Qu.: 56.00

Max. :25.0 Max. :120.00

### Getting the Average Measures

R provides a number of functions that give us different average measures for given data. These average measures include:

* **Mean:** The mean of a given set of numeric or logical values(it may be a vector or a row or column of any other data structure) can be easily found using the **mean()** function.
* **Median:** Finding the median of a set of numeric or logical values is also very easy by using the **median()** function.
* **Standard deviation:** The standard deviation of a set of numerical values can be found using the **sd()** function.
* **Variance:** the **var()** function gives us the variance of a set of numeric or logical values.
* **Median Absolute Variance:** The median absolute variance of a set of numeric or logical values can be found by using the **mad()** function.
* **Maximum:** In a given set of numeric or logical values, we can use the **max()** function to find the maximum or the largest value in the set.

**Note:** NA is considered to be the largest by the **max()** function unless its na.rm argument is set to TRUE.

* **Minimum:** The **min()** function is a very handy way to find out the smallest value in a set of numeric values.

**Note:** Like the **max()** function, the min() function considers NA to be the smallest unless na.rm is set to TRUE.

* **Sum:** The sum of a set of numerical values can be found by simply using the **sum()** function.
* **Length:** The length or the number of values in a set is given by the **length()** function.

**Example:**

mean(mtcars$mpg)

median(mtcars$mpg)

sd(mtcars$mpg)

var(mtcars$mpg)

mad(mtcars$mpg)

max(mtcars$mpg, na.rm = TRUE)

min(mtcars$mpg, na.rm = TRUE)

sum(mtcars$mpg)

length(mtcars$mpg)

**Output:**

mean(mtcars$mpg)

[1] 20.09062

> median(mtcars$mpg)

[1] 19.2

>sd(mtcars$mpg)

[1] 6.026948

> var(mtcars$mpg)

[1] 36.3241

> mad(mtcars$mpg)

[1] 5.41149

>max(mtcars$mpg, na.rm = TRUE)

[1] 33.9

>min(mtcars$mpg, na.rm = TRUE)

[1] 10.4

> sum(mtcars$mpg)

[1] 642.9

> length(mtcars$mpg)

[1] 32

**Quantile Function:**

* A quantile is nothing but a sample that is divided into equal groups or sizes. Due to this nature, the quantiles are also called as Fractiles. In the quantiles, the 25th percentile is called as lower quartile, 50th percentile is called as Median and the 75th Percentile is called as the upper quartile.
* This is particularly useful when you’re doing exploratory analysis and reporting, especially if you’re analyzing data which may not be normally distributed.
* We’re going to use the r quantile function; this utility is part of base R (so you don’t need to import any libraries) and can be adapted to generate a variety of “rank based” statistics about your sample.

## **Quantile() function syntax**

The syntax of the Quantile() function in R is,

|  |
| --- |
| quantile(x, probs = , na.rm = **FALSE**) |

Where,

* **X** = the input vector or the values
* **Probs** = probabilities of values between 0 and 1.
* **na.rm** = removes the NA values.

We’re going to use the r quantile function; this utility is part of base R (so you don’t need to import any libraries) and can be adapted to generate a variety of “rank based” statistics about your sample.

**Example:**

# quartile in R example

test = c(9,9,8,9,10,9,3,5,6,8,9,10,11,12,13,11,10)

# get quartile in r code (single line)

quantile(test, prob=c(.25,.5,.75))

**Output:**

quantile(test, prob=c(.25,.5,.75))

25% 50% 75%

8 9 10

You can also use the summary function to generate the same information.

# quartile in R example - summary function

test = c(9,9,8,9,10,9,3,5,6,8,9,10,11,12,13,11,10)

summary(test)

**Output:**

summary(test)

Min. 1st Qu. Median Mean 3rd Qu. Max.

3.000 8.000 9.000 8.941 10.000 13.000

**Quantile function using mtcars and cars dataset.**

quantile(mtcars$wt)

**Output:**

0% 25% 50% 75% 100%

1.51300 2.58125 3.32500 3.61000 5.42400

quantile(mtcars$mpg)

**Output:**

0% 25% 50% 75% 100%

10.400 15.425 19.200 22.800 33.900

quantile(cars$speed)

**Output:**

0% 25% 50% 75% 100%

4 12 15 19 25

quantile(cars$speed,c(.2, .4, .8))

**Output:**

20% 40% 80%

11 14 20

**6(b) Write an R script to find subset of dataset by using subset(), aggregate() functions on iris**

**dataset.**

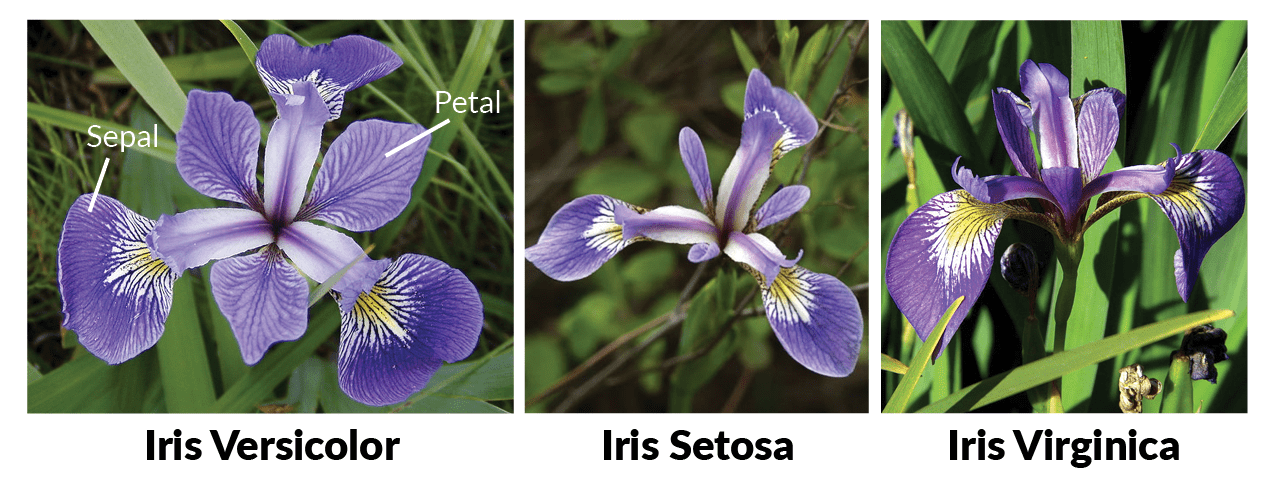
# Sub setting Datasets in R

* R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations.
* Whether you're comparing how different demographics respond to marketing campaigns, zooming in on a specific time frame, or pulling information about a select few products from the inventory, subsetting datasets enables you to extract useful observations in your dataset.
* R is a great tool that makes subsetting data easy and intuitive.
* The **subset( )**function is the easiest way to select variables and observations.
* Subsetting your data does not change the content of your data, but simply selects the portion most relevant to the goal you have in mind. In general, there are three ways to subset the rows and columns of your dataset—by index, by name, and by value.

# Iris dataset

* Iris dataset gives the measurements in centimetres of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

##### In this picture You can see what are we dealing with and how exactly looks the variables (sepal length and width and petal length and width) we are measuring and the object itself:



#### **Format**

iris is a data frame with 150 cases (rows) and 5 variables (columns) named Sepaal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species.

##### Here’s a little summary of what you can basically see in dataset iris:

**summary(iris)**

**Output:**

Sepal.LengthSepal.WidthPetal.LengthPetal.Width Species

Min. :4.300 Min. :2.000 Min. :1.000 Min. :0.100 setosa :50

1st Qu.:5.100 1st Qu.:2.800 1st Qu.:1.600 1st Qu.:0.300 versicolor:50

Median :5.800 Median :3.000 Median :4.350 Median :1.300 virginica :50

Mean :5.843 Mean :3.057 Mean :3.758 Mean :1.199

3rd Qu.:6.400 3rd Qu.:3.300 3rd Qu.:5.100 3rd Qu.:1.800

Max. :7.900 Max. :4.400 Max. :6.900 Max. :2.500

**names(iris)**

[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"

**str(iris)**

'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 ...

**Let's take a look at the data itself. Let's see the first 5 rows of data for each class:**

# Get first 5 rows of each subset

subset(iris, Species == "setosa")[1:5,]

**Output:**

Sepal.LengthSepal.WidthPetal.LengthPetal.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

subset(iris, Species == "versicolor")[1:5,]

**Output:**

Sepal.LengthSepal.WidthPetal.LengthPetal.Width Species

51 7.0 3.2 4.7 1.4 versicolor

52 6.4 3.2 4.5 1.5 versicolor

53 6.9 3.1 4.9 1.5 versicolor

54 5.5 2.3 4.0 1.3 versicolor

55 6.5 2.8 4.6 1.5 versicolor

subset(iris, Species == "virginica")[1:5,]

**Output:**

Sepal.LengthSepal.WidthPetal.LengthPetal.Width Species

101 6.3 3.3 6.0 2.5 virginica

102 5.8 2.7 5.1 1.9 virginica

103 7.1 3.0 5.9 2.1 virginica

104 6.3 2.9 5.6 1.8 virginica

105 6.5 3.0 5.8 2.2 virginica

# AGGREGATE() FUNCTION IN R

### aggregate() Function in R Splits the data into subsets, computes summary statistics for each subsets and returns the result in a group by form.

* aggregate() function is useful in performing all the aggregate operations like sum,count,mean, minimum and Maximum.

### Use aggregate() function to find summary statistics by group.

#### **Syntax for Aggregate() Function in R:**

**aggregate(x, by, FUN, …, simplify = TRUE, drop = TRUE)**

|  |  |
| --- | --- |
| X | an R object, mostly a data frame |
| by | a list of grouping elements, by which the subsets are grouped by |
| FUN | a function to compute the summary statistics |
| simplify | a logical indicating whether results should be simplified to a vector or matrix if possible |
| drop | a logical indicating whether to drop unused combinations of grouping values. |

**Example:**

agg\_mean = aggregate(iris[,1:4], by = list(iris$Species),FUN=mean,na.rm=TRUE)

agg\_mean

the above code takes first 4 columns of iris data set and groups by “species” by computing the mean for each group, so the output will be

**Output:**

Group.1 Sepal.LengthSepal.WidthPetal.LengthPetal.Width

1 setosa 5.006 3.428 1.462 0.246

2 versicolor 5.936 2.770 4.260 1.326

3 virginica 6.588 2.974 5.552 2.026

#### **Example for aggregate() function in R with sum:**

Let’s use the aggregate() function in R to create the sum of all the metrics across species and group by species.

# Aggregate function in R with sum summary statistics

agg\_sum = aggregate(iris[,1:4],by=list(iris$Species),FUN=sum, na.rm=TRUE)

agg\_sum

**Output:**

Group.1 Sepal.LengthSepal.WidthPetal.LengthPetal.Width

1 setosa 250.3 171.4 73.1 12.3

2 versicolor 296.8 138.5 213.0 66.3

3 virginica 329.4 148.7 277.6 101.3

#### **Example for aggregate() function in R with sum:**

Let’s use the aggregate() function in R to create the sum of all the metrics across species and group by species.

# Aggregate function in R with sum summary statistics

agg\_sum = aggregate(iris[,1:4],by=list(iris$Species),FUN=sum, na.rm=TRUE)

agg\_sum

**Output:**

Group.1 Sepal.LengthSepal.WidthPetal.LengthPetal.Width

1 setosa 250.3 171.4 73.1 12.3

2 versicolor 296.8 138.5 213.0 66.3

3 virginica 329.4 148.7 277.6 101.3

#### **Example for aggregate() function in R with count:**

# Aggregate function in R with count

agg\_count = aggregate(iris[,1:4],by=list(iris$Species),FUN=length)

agg\_count

the above code takes first 4 columns of iris data set and groups by “species” by computing the count for each group, so the output will be

**Output:**

Group.1 Sepal.LengthSepal.WidthPetal.LengthPetal.Width

1 setosa 50 50 50 50

2 versicolor 50 50 50 50

3 virginica 50 50 50 50

#### **Example for aggregate() function in R with maximum:**

Let’s use the aggregate() function to create the maximum of all the metrics across species and group by species.

# Aggregate function in R with maximum

agg\_max = aggregate(iris[,1:4],by=list(iris$Species),FUN=max, na.rm=TRUE)

agg\_max

the above code takes first 4 columns of iris data set and groups by “species” by computing the max for each group, so the output will be

**Output:**

Group.1 Sepal.LengthSepal.WidthPetal.LengthPetal.Width

1 setosa5.8 4.4 1.9 0.6

2 versicolor 7.0 3.4 5.1 1.8

3 virginica 7.9 3.8 6.9 2.5

**Week-7**

(a) Reading different types of data sets (.txt, .csv) from Web or disk and writing in file in specific

disk location.

(b) Reading Excel data sheet in R.

(c) Reading XML dataset in R.

**7(a) Reading different types of data sets (.txt, .csv) from Web or disk and writing in file in**

**specific disk location.**

**Reading data from txt or csv files**

The R base function read.table() is a general function that can be used to read a file in table format. The data will be imported as a data frame.

Note that, depending on the format of your file, several variants of read.table() are available, including read.csv, read.csv2(), read.delim and read.delim2().

* **read.csv**(): for reading **“comma separated value”** files (“.csv”).
* **read.csv2**(): variant used in countries that use a comma “,” as decimal point and a semicolon “;” as field separators.
* **read.delim**(): for reading “tab-separated value” files (“.txt”). By default, point (“.”) is used as decimal points.
* **read.delim2**(): for reading “tab-separated value” files (“.txt”). By default, comma (“,”) is used as decimal points.

The simplified formats of these functions are, as follows:

**# Read tabular data into R**

read.table(file, header=FALSE, sep="", dec=".")

**# Read "comma separated value" files (".csv")**

read.csv(file, header=TRUE, sep=",", dec=".", **...**)

**# Or use read.csv2: variant used in countries that # use a comma as decimal point and a semicolon as field separator.**

read.csv2(file, header=TRUE, sep=";", dec=",", **...**)

**# Read TAB delimited files**

read.delim(file, header=TRUE, sep="\t", dec=".", **...**)read.delim2(file, header=TRUE, sep="\t", dec=",", **...**)

* **file**: the path to the file containing the data to be imported into R.
* **sep**: the field separator character. “\t” is used for tab-delimited file.
* **header**: logical value. If TRUE, **read.table()** assumes that your file has a header row, so row 1 is the name of each column. If that’s not the case, you can add the argument **header = FALSE**.
* **dec**: the character used in the file for decimal points.

# Reading a local file

To import a local .txt or a .csv file, the syntax would be:

# Read a txt file, named "mtcars.txt"

my\_data<-read.delim("mtcars.txt")

# Read a csv file, named "mtcars.csv"

my\_data<-read.csv("mtcars.csv")

**Note:**

The above R code, assumes that the file “mtcars.txt” or “mtcars.csv” is in your current working directory. To know your current working directory, type the function **getwd**() in R console.

* It’s also possible to choose a file interactively using the function **file.choose**(), which I recommend if you’re a beginner in R programming:

**# Read a txt file**

my\_data<-read.delim(file.choose())

**# Read a csv file**

my\_data<-read.csv(file.choose())

If you use the R code above in RStudio, you will be asked to choose a file.

# Reading a file from internet

It’s possible to use the functions **read.delim**(), **read.csv**() and **read.table**() to import files from the web.

my\_data<-read.delim("http://www.sthda.com/upload/boxplot\_format.txt")

**head**(my\_data)

🡪Here I am using weather data.

**Example-1: R program reading a .text file**

# Read a text file using read.delim()

Data1 = read.delim("weather.txt", header = TRUE)

print(Data1)

**Output:**

my\_data

outlook temperature humidity windy play

1 overcast hot high FALSE yes

2 overcast cool normal TRUE yes

3 overcast mild high TRUE yes

4 overcast hot normal FALSE yes

5 rainy mild high FALSE yes

6 rainy cool normal FALSE yes

7 rainy cool normal TRUE no

8 rainy mild normal FALSE yes

9 rainy mild high TRUE no

10 sunny hot high FALSE no

11 sunny hot high TRUE no

12 sunny mild high FALSE no

13 sunny cool normal FALSE yes

14 sunny mild normal TRUE yes

Data2<-read.table(“weather.txt”,header=TRUE, sep="\t")

Data2

**Output:**

Data2

outlook temperature humidity windy play

1 overcast hot FALSE yes

2 overcast cool normal TRUE yes

3 overcast mild high TRUE yes

4 overcast hot normal FALSE yes

5 rainy mild high FALSE yes

6 rainy cool normal FALSE yes

7 rainy cool normal TRUE no

8 rainy mild normal FALSE yes

9 rainy mild high TRUE no

10 sunny hot high FALSE no

11 sunny hot high TRUE no

12 sunny mild high FALSE no

13 sunny cool normal FALSE yes

14 sunny mild normal TRUE yes

**Example-2: R program reading a .csv file**

Data3<- read.csv(“weather.csv”, header=TRUE)

Data3

outlook temperature humidity windy play

1 overcast hot high FALSE yes

2 overcast cool normal TRUE yes

3 overcast mild high TRUE yes

4 overcast hot normal FALSE yes

5 rainy mild high FALSE yes

6 rainy cool normal FALSE yes

7 rainy cool normal TRUE no

8 rainy mild normal FALSE yes

9 rainy mild high TRUE no

10 sunny hot high FALSE no

11 sunny hot high TRUE no

12 sunny mild high FALSE no

13 sunny cool normal FALSE yes

14 sunny mild normal TRUE yes

Data4<-read.table(“weather.csv”, header=TRUE,sep=",")

Data4

outlook temperature humidity windy play

1 overcast hot high FALSE yes

2 overcast cool normal TRUE yes

3 overcast mild high TRUE yes

4 overcast hot normal FALSE yes

5 rainy mild high FALSE yes

6 rainy cool normal FALSE yes

7 rainy cool normal TRUE no

8 rainy mild normal FALSE yes

9 rainy mild high TRUE no

10 sunny hot high FALSE no

11 sunny hot high TRUE no

12 sunny mild high FALSE no

13 sunny cool normal FALSE yes

14 sunny mild normal TRUE yes

It’s also possible to choose a file interactively using the function **file.choose**()

**🡪 To read .txt file**

data3 <-read.delim(file.choose(), header=TRUE)

data3

data4 <-read.table(file.choose(),header=TRUE, sep="\t")

data4

**🡪 To read .csv file**

data1 <- read.csv(file.choose(), header=TRUE)

data1

data2 <-read.table(file.choose(), header=TRUE,sep=",")

data2

# Reading a file from internet

It’s possible to use the functions **read.delim**(), **read.csv**() and **read.table**() to import files from the web.

my\_data<-read.delim("http://www.sthda.com/upload/boxplot\_format.txt")

**head**(my\_data)

**Output:**

Nom variable Group

1 IND1 10 A

2 IND2 7 A

3 IND3 20 A

4 IND4 14 A

5 IND5 14 A

6 IND6 12 A

7 IND7 10 A

8 IND8 23 A

9 IND9 17 A

10 IND10 20 A

11 IND11 14 A

12 IND12 13 A

13 IND13 11 B

14 IND14 17 B

15 IND15 21 B

16 IND16 11 B

17 IND17 16 B

18 IND18 14 B

19 IND19 17 B

20 IND20 17 B

21 IND21 19 B

22 IND22 21 B

23 IND23 7 B

24 IND24 13 B

25 IND25 0 C

26 IND26 1 C

27 IND27 7 C

28 IND28 2 C

29 IND29 3 C

30 IND30 1 C

31 IND31 2 C

32 IND32 1 C

33 IND33 3 C

34 IND34 0 C

35 IND35 1 C

36 IND36 4 C

37 IND37 3 D

38 IND38 5 D

39 IND39 12 D

40 IND40 6 D

41 IND41 4 D

42 IND42 3 D

43 IND43 5 D

44 IND44 5 D

45 IND45 5 D

46 IND46 5 D

47 IND47 2 D

48 IND48 4 D

49 IND49 3 E

50 IND50 5 E

51 IND51 3 E

52 IND52 5 E

53 IND53 3 E

54 IND54 6 E

55 IND55 1 E

56 IND56 1 E

57 IND57 3 E

58 IND58 2 E

59 IND59 6 E

60 IND60 4 E

61 IND61 11 F

62 IND62 9 F

63 IND63 15 F

64 IND64 22 F

65 IND65 15 F

66 IND66 16 F

67 IND67 13 F

68 IND68 10 F

69 IND69 26 F

70 IND70 26 F

71 IND71 24 F

72 IND72 13 F

**Import dataset in R programming**

R is a programming language designed for data analysis. Therefore, loading data is one of the core features of R.

R contains a set of functions that can be used to load data sets into memory. You can also load data into memory using R Studio - via the menu items and toolbars.

## Data Formats

R can load data in two different formats:

* CSV files
* Text files

CSV means Comma Separated Values. You can export CSV files from many data carrying applications. For instance, you can export CSV files from data in an Excel spreadsheet. Here is an example of how a CSV file looks like inside:

name,id,salary

"John Doe",1,99999.00

"Joe Blocks",2,120000.00

"Cindy Loo",3,150000.00

As you can see, the values on each line are separated by commas. The first line contains a list of column names. These column names tell what the data in the following lines mean. These names only make sense to you. R does not care about these names. R just uses these name to identify data from the different columns.

A text file is typically similar to a CSV file, but instead of using commas as separators between values, text files often use other characters, like e.g. a Tab character. Here is an example of how a text file could look inside:

name id salary

"John Doe" 1 99999.00

"Joe Blocks" 2 120000.00

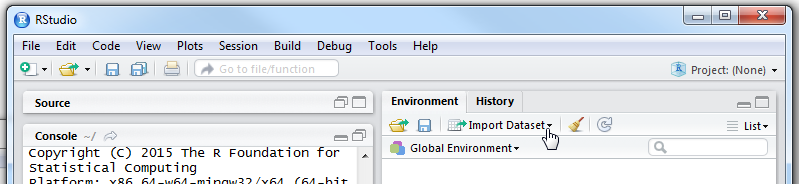
"Cindy Loo" 3 150000.00

As you can see, the data might be easier to read in text format - if you look at the data directly in the data file that is. Once the data is loaded into R / R studio, there is no difference. You can look at the data in R Studio's tabular data set viewer, and then you cannot see the difference between CSV files and text files.

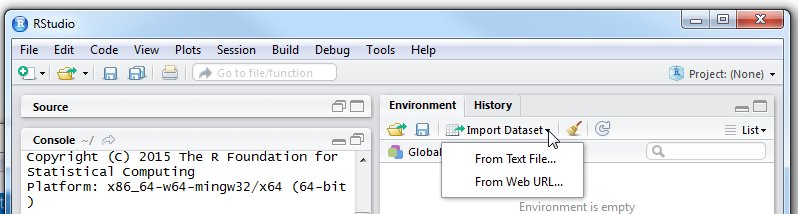
Actually, the name "text files" is a bit confusing. Both CSV files and text files contains data in textual form (as characters). One just uses commas as separator between the values, whereas the others use a tab character.

## Load Data Via R Studio Menu Items

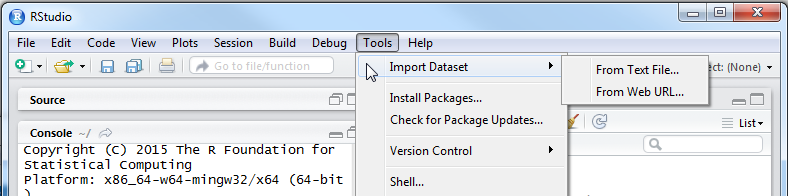
The easiest way to load data into memory in R is by using the R Studio menu items. R Studio has menu items for loading data in two different places. The first is in the toolbar of the upper right section of R Studio. This screenshot shows where the "Import Dataset" button is (look for the little mouse pointer "hand") :



When you click the button you get this little menu:



You can also import data from the top menu of R Studio. The next screenshot shows where the "Import Dataset" menu item is located in R Studio's top menu:



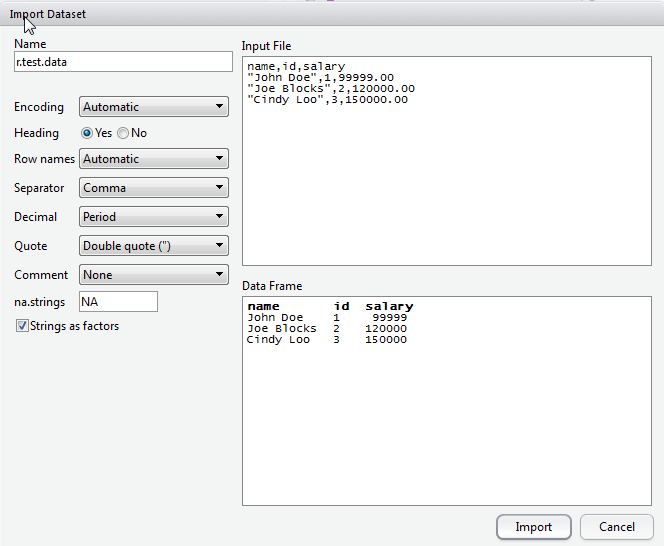
### Text File or Web URL

As you can see in both the "Import Dataset" menu items, you can import a data set "From Text File" or "From Web URL". These two options refer to where you load the data from. "From Text File" means from a text file on your local computer. "From Web URL" means that you load the data from a web server somewhere on the internet.

Regardless of whether you choose "From Text File" or "From Web URL", R can load the file as either a CSV or text file. The location of the file has nothing to do with the data format used inside the file. Don't get confused by that. The menu item "From Text file" does not mean "text file format" (tab characters as separators). It just means "a file on your local computer". "From Local File" would probably have been a more informative text for this menu item.

### Selecting Data Format

After you have chosen the location to load the file from, you will be shown a dialog like this:



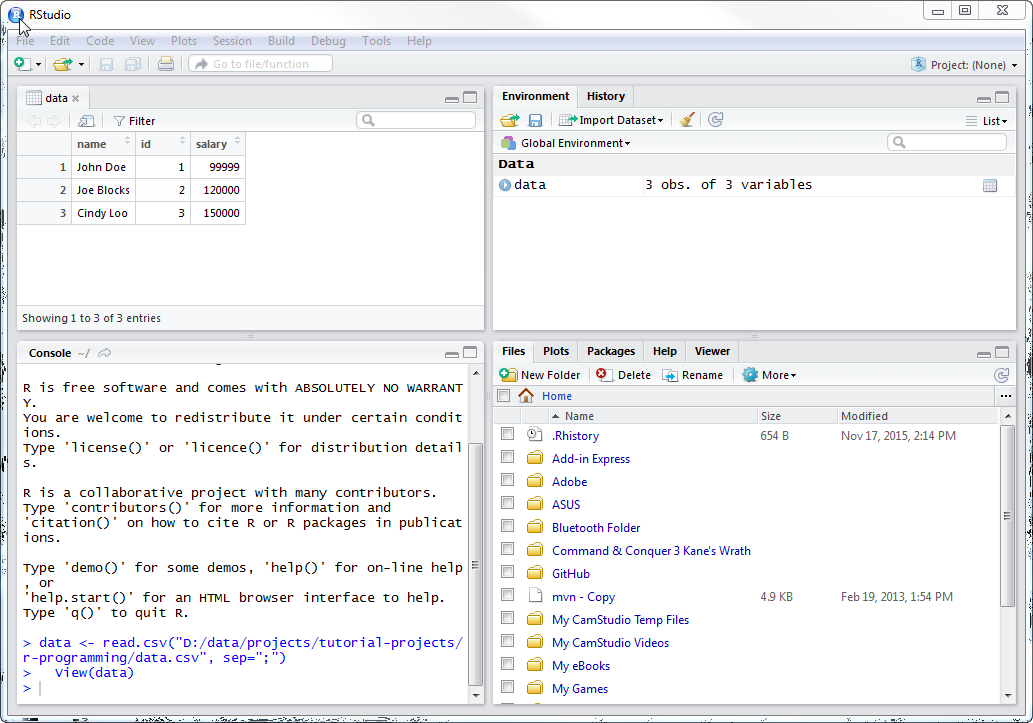
The select boxes (drop down boxes) allows you to specify different configurations about the data format of the file you are about to import. In the boxes on the right you can see two boxes. The top box shows you what the data file looks like. The bottom box shows you how R Studio interprets the data in the file based on the configurations chosen in the select boxes in the left side of the dialog. If you change the choices in the select boxes you will see that the bottom right box changes.

When you have selected all the configurations you need in the select boxes on the left, click the "Import" button. The data will now be loaded into R Studio.

Note that R Studio prints the R commands needed to load the data into the R console in the left side of R studio. You can copy these functions and use them to load data into R via R code.

### After the Data is Loaded

After you have loaded the data into R Studio it will look similar to the screenshot below:



**7(b) Reading Excel data sheet in R**.

## Steps to Import an Excel file into R

### Step 1: Install the readxl package

In the R Console, type the following command to [install](https://datatofish.com/install-package-r/) the **readxl** package:

Install.packages(“readxl”)

### Step 2: Prepare your Excel File

Let’s suppose that you have an Excel file with some data about products:

|  |  |
| --- | --- |
| **Product** | **Price** |
| Refrigerator | 1200 |
| Oven | 750 |
| Dishwasher | 900 |
| Coffee Maker | 300 |

And let’s say that the Excel file name is **product\_list**, and your goal is to import that file into R.

### Step 3: Import the Excel file into R

### In order to import your file, you’ll need to apply the following template in the R Editor:

library(“readxl”)

read.excel(“Path where your Excel file is stored\\FileName.xlsx”)

**Example:**

my\_data<- read\_excel("product\_list.xlsx")

my\_data

**(OR)**

my\_data<-read\_excel(file.choose())

my\_data

**Note:**

If you use the R code above in RStudio, you will be asked to choose a file.

**Output:**

# A tibble: 4 x 2

Product Price

<chr><dbl>

1 Refrigerator 1200

2 Oven 750

3 Dishwasher 900

4 Coffee Maker 300

# Importing Excel files using xlsx package

The **xlsx** package, a java-based solution, is one of the powerful R packages to **read**, **write** and **format** **Excel files**.

## Installing and loading xlsx package

* Install

install.packages(“xlsx”)

* Load

library(“xlsx)

**Using xlsx package**

There are two main functions in **xlsx** package for reading both xls and xlsx Excel files: **read.xlsx**() and **read.xlsx2**() [faster on big files compared to read.xlsx function].

The simplified formats are:

read.xlsx(file, sheetIndex, header=TRUE)

read.xlsx2(file, sheetIndex, header=TRUE)

* file: file path
* sheetIndex: the index of the sheet to be read
* header: a logical value. If TRUE, the first row is used as column names.

**Example:**

library(“xlsx”)

my\_data1 <- read.xlsv(file.choose(), 1) # read first sheet

**7(c) Reading XML dataset in R.**

In R, we can read the xml files by installing "XML" package into the R environment. This package will be installed with the help of the familiar command i.e., install. packages.

install.packages(“XML”)

**Creating XML File**

Save the following data with the .xml file extension to create an xml file. XML tags describe the meaning of data, so that data contained in such tags can easily tell or explain about the data.

Example: xml\_data.xml

**Example: xml\_data.xml**

<records>

<employee\_info>

<id>1</id>

<name>Shubham</name>

<salary>623</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>2</id>

<name>Nishka</name>

<salary>552</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Gunjan</name>

<salary>669</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Sumit</name>

<salary>825</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Arpita</name>

<salary>762</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Vaishali</name>

<salary>882</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Anisha</name>

<salary>783</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

<employee\_info>

<id>1</id>

<name>Ginni</name>

<salary>964</salary>

<date>1/1/2012</date>

<dept>IT</dept>

</employee\_info>

</records>

## **Reading XML File**

In R, we can easily read an xml file with the help of xmlParse() function. This function is stored as a list in R. To use this function, we first need to load the xml package with the help of the library() function. Apart from the xml package, we also need to load one additional package named methods.

**Example: Reading xml data in the form of a list.**

# Loading the package required to read XML files.

library("XML")

# Also loading the other required package.

library("methods")

# Giving the input file name to the function.

result **<-** xmlParse(file = "xml\_data.xml")

xml\_data **<-** xmlToList(result)

print(xml\_data)

**# To download file to the current working directory**

download.file("https://www.w3schools.com/xml/simple.xml", "breakfast.xml")

**# Install XML package**

install.packages("XML")

**# To load library**

library(XML)

**# Giving the input file name to the function.**

doc <- xmlParse("breakfast.xml")

print(doc)

**#Converting the data into list**

xml\_data<-xmlToList(doc)

print(xml\_data)

xmldataframe<- xmlToDataFrame("breakfast.xml")

xmldataframe

wd<- getwd()

wd

**Output:**

library(XML)

> doc <- xmlParse("breakfast.xml")

> print(doc)

<?xml version="1.0" encoding="UTF-8"?>

<breakfast\_menu>

<food>

<name>Belgian Waffles</name>

<price>$5.95</price>

<description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>

<calories>650</calories>

</food>

<food>

<name>Strawberry Belgian Waffles</name>

<price>$7.95</price>

<description>Light Belgian waffles covered with strawberries and whipped cream</description>

<calories>900</calories>

</food>

<food>

<name>Berry-Berry Belgian Waffles</name>

<price>$8.95</price>

<description>Light Belgian waffles covered with an assortment of fresh berries and whipped cream</description>

<calories>900</calories>

</food>

<food>

<name>French Toast</name>

<price>$4.50</price>

<description>Thick slices made from our homemade sourdough bread</description>

<calories>600</calories>

</food>

<food>

<name>Homestyle Breakfast</name>

<price>$6.95</price>

<description>Two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>

<calories>950</calories>

</food>

</breakfast\_menu>

**>xml\_data<-xmlToList(doc)**

**> print(xml\_data**)

$food

$food$name

[1] "Belgian Waffles"

$food$price

[1] "$5.95"

$food$description

[1] "Two of our famous Belgian Waffles with plenty of real maple syrup"

$food$calories

[1] "650"

$food

$food$name

[1] "Strawberry Belgian Waffles"

$food$price

[1] "$7.95"

$food$description

[1] "Light Belgian waffles covered with strawberries and whipped cream"

$food$calories

[1] "900"

$food

$food$name

[1] "Berry-Berry Belgian Waffles"

$food$price

[1] "$8.95"

$food$description

[1] "Light Belgian waffles covered with an assortment of fresh berries and whipped cream"

$food$calories

[1] "900"

$food

$food$name

[1] "French Toast"

$food$price

[1] "$4.50"

$food$description

[1] "Thick slices made from our homemade sourdough bread"

$food$calories

[1] "600"

$food

$food$name

[1] "Homestyle Breakfast"

$food$price

[1] "$6.95"

$food$description

[1] "Two eggs, bacon or sausage, toast, and our ever-popular hash browns"

$food$calories

[1] "950"

**Week-8**

(a) Implement R script to create a Pie chart, Bar chart, scatter plot and Histogram.

(Introduction to ggplot2 graphics)

(b) Implement R script to perform mean, median, mode, range, summary, variance, standard

deviation operations.

**8(a) Implement R script to create a Pie chart, Bar chart, scatter plot and Histogram.**

**(Introduction to ggplot2 graphics)**

# R Data Visualization

Data visualization is an efficient technique for gaining insight about data through a visual medium. With the help of visualization techniques, a human can easily obtain information about hidden patterns in data that might be neglected.

By using the data visualization technique, we can work with large datasets to efficiently obtain key insights about it.

### Standard Graphics

R standard graphics are available through package graphics, include several functions which provide statistical plots, like:

* Scatterplots
* Piecharts
* Boxplots
* Barplots etc.

**Scatterplots**

# R Scatterplots

The scatter plots are used to compare variables. A comparison between variables is required when we need to define how much one variable is affected by another variable.

In a scatterplot, the data is represented as a collection of points. Each point on the scatterplot defines the values of the two variables. One variable is selected for the vertical axis and other for the horizontal axis.

In R, there are **two ways of creating scatterplot**, i.e., using **plot()** function and using the **ggplot2** package's functions.

**There is the following syntax for creating scatterplot in R:**

plot(x, y, main, xlab, ylab, xlim, ylim, axes)

Here,

|  |  |  |
| --- | --- | --- |
| **S.No** | **Parameters** | **Description** |
| 1. | x | It is the dataset whose values are the horizontal coordinates. |
| 2. | y | It is the dataset whose values are the vertical coordinates. |
| 3. | main | It is the title of the graph. |
| 4. | xlab | It is the label on the horizontal axis. |
| 5. | ylab | It is the label on the vertical axis. |
| 6. | xlim | It is the limits of the x values which is used for plotting. |
| 7. | ylim | It is the limits of the values of y, which is used for plotting. |
| 8. | axes | It indicates whether both axes should be drawn on the plot. |

### Example

**Program:**

#Fetching two columns from mtcars

data **<-mtcars**[, c('wt','mpg')]

# Giving a name to the chart file.

png(file = "scatterplot.png")

# Plotting the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.

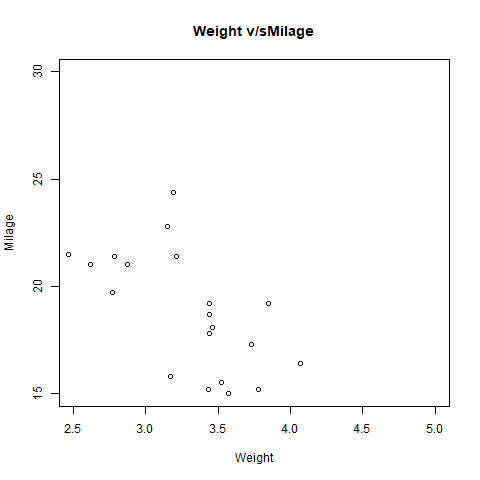
plot(x = data$wt,y = data$mpg, xlab = "Weight", ylab = "Milage", xlim = c(2.5,5),

ylim = c(15,30), main = "Weight v/sMilage")

# Saving the file.

dev.off()

**Output:**



## **Scatterplot using ggplot2**

In R, there is another way for creating scatterplot i.e. with the help of ggplot2 package.

The ggplot2 package provides ggplot() and geom\_point() function for creating a scatterplot. The ggplot() function takes a series of the input item. The first parameter is an input vector, and the second is the aes() function in which we add the x-axis and y-axis.

**Program:**

#Loading ggplot2 package

library(ggplot2)

# Giving a name to the chart file.

png(file = "scatterplot\_ggplot.png")

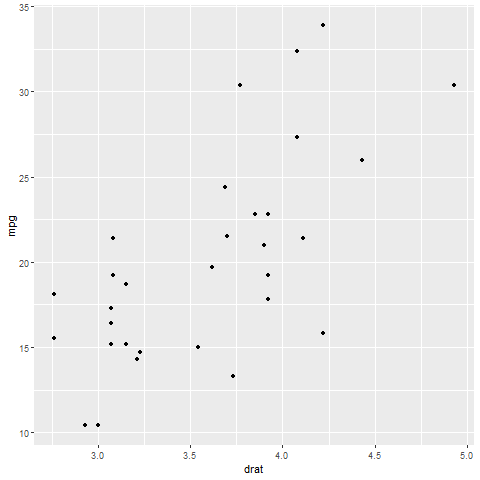
# Plotting the chart using ggplot() and geom\_point() functions.

ggplot(mtcars,aes(x = drat, y = mpg)) +geom\_point()

# Saving the file.

dev.off()

**Output:**



# R Pie Charts

R programming language has several libraries for creating charts and graphs. A pie-chart is a representation of values in the form of slices of a circle with different colors. Slices are labeled with a description, and the numbers corresponding to each slice are also shown in the chart. However, pie charts are not recommended in the R documentation, and their characteristics are limited.

The Pie charts are created with the help of pie () function, which takes positive numbers as vector input. Additional parameters are used to control labels, colors, titles, etc.

There is the following syntax of the pie() function:

pie(X, Labels, Radius, Main, Col, Clockwise)

Here,

1. **X** is a vector that contains the numeric values used in the pie chart.
2. **Labels** are used to give the description to the slices.
3. **Radius** describes the radius of the pie chart.
4. **Main** describes the title of the chart.
5. **Col** defines the color palette.
6. **Clockwise** is a logical value that indicates the clockwise or anti-clockwise direction in which slices are drawn.

**Program:**

**# Creating data for the graph.**

x <- c(20, 65, 15, 50)

print(x)

labels <- c("India", "America", "Shri Lanka", "Nepal")

print(labels)

**# Giving the chart file a name.**

png(file = "Country.jpg")

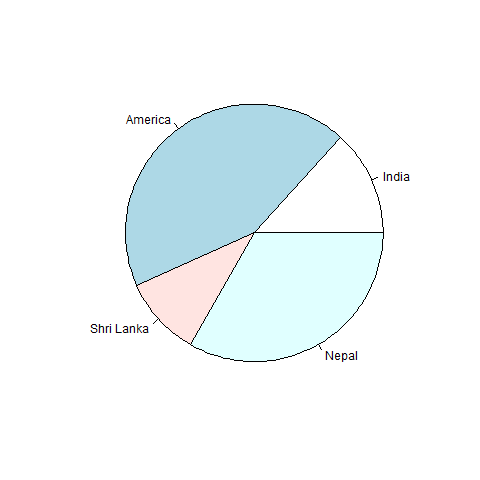
**# Plotting the chart**.

pie(x,labels)

**# Saving the file.**

dev.off()

**Output:**



## **Title and color**

A pie chart has several more features that we can use by adding more parameters to the pie() function. We can give a title to our pie chart by passing the main parameter. It tells the title of the pie chart to the pie() function. Apart from this, we can use a rainbow colour pallet while drawing the chart by passing the col parameter.

**Program:**

# Creating data for the graph.

x <- c(20, 65, 15, 50)

labels <- c("India", "America", "Shri Lanka", "Nepal")

# Giving the chart file a name.

png(file = "title\_color.jpg")

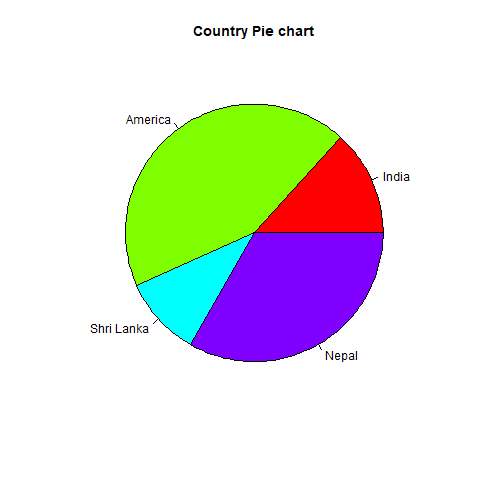
# Plotting the chart.

pie(x,labels,main="Country Pie chart",col=rainbow(length(x)))

# Saving the file.

dev.off()

**Output:**



## **Slice Percentage & Chart Legend**

There are two additional properties of the pie chart, i.e., slice percentage and chart legend. We can show the data in the form of percentage as well as we can add legends to plots in R by using the legend() function. There is the following syntax of the legend() function.

legend(x,y=NULL,legend,fill,col,bg)

Here,

* x and y are the coordinates to be used to position the legend.
* legend is the text of legend
* fill is the color to use for filling the boxes beside the legend text.
* col defines the color of line and points besides the legend text.
* bg is the background color for the legend box.

**Program:**

# Creating data for the graph.

x <- c(20, 65, 15, 50)

labels <- c("India", "America", "Shri Lanka", "Nepal")

pie\_percent<- round(100\*x/sum(x), 1)

# Giving the chart file a name.

png(file = "per\_pie.jpg")

# Plotting the chart.

pie(x, labels = pie\_percent, main = "Country Pie Chart",col = rainbow(length(x)))

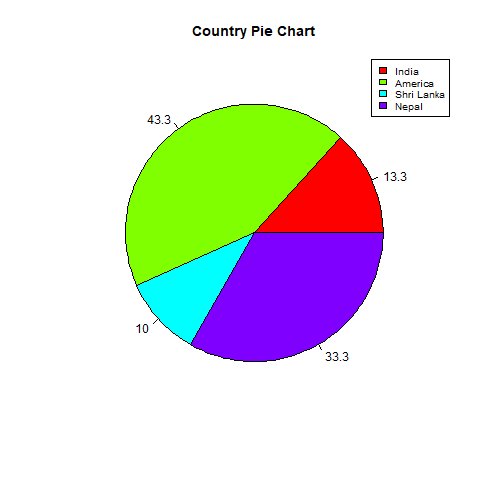
legend("topright", c("India", "America", "Shri Lanka", "Nepal"), cex = 0.8,

fill = rainbow(length(x)))

#Saving the file.

dev.off()

**Output:**



## **Dimensional Pie Chart**

In R, we can also create a three-dimensional pie chart. For this purpose, R provides a plotrix package whose pie3D() function is used to create an attractive 3D pie chart. The parameters of pie3D() function remain same as pie() function.

**Program:**

**# Getting the library.**

library(plotrix)

# Creating data for the graph.

x <- c(20, 65, 15, 50,45)

labels <- c("India", "America", "Shri Lanka", "Nepal","Bhutan")

pie\_percent<- round(100\*x/sum(x), 1)

# Giving the chart file a name.

png(file = "three\_D\_pie.jpg")

# Plotting the chart.

pie3D(x, labels = pie\_percent, main = "Country Pie Chart",col = rainbow(length(x)))

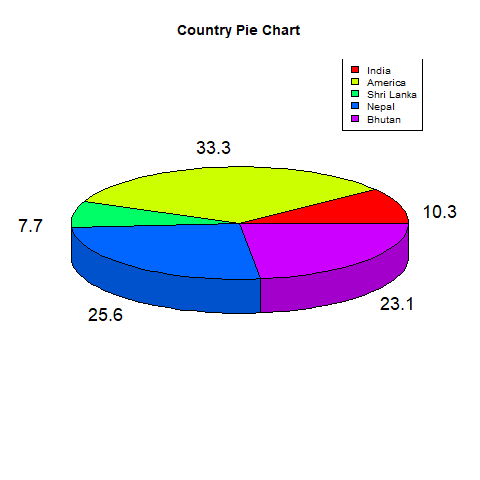
legend("topright", c("India", "America", "Shri Lanka", "Nepal","Bhutan"), cex = 0.8,

fill = rainbow(length(x)))

#Saving the file.

dev.off()

**Output:**



# R Boxplot

Boxplots are a measure of how well data is distributed across a data set. This divides the data set into three quartiles. This graph represents the minimum, maximum, average, first quartile, and the third quartile in the data set. Boxplot is also useful in comparing the distribution of data in a data set by drawing a boxplot for each of them.

R provides a boxplot() function to create a boxplot. There is the following syntax of boxplot() function:

boxplot(x, data, notch, varwidth, names, main)

Here,

|  |  |  |
| --- | --- | --- |
| **S.No** | **Parameter** | **Description** |
| 1. | x | It is a vector or a formula. |
| 2. | data | It is the data frame. |
| 3. | notch | It is a logical value set as true to draw a notch. |
| 4. | varwidth | It is also a logical value set as true to draw the width of the box same as the sample size. |
| 5. | names | It is the group of labels that will be printed under each boxplot. |
| 6. | main | It is used to give a title to the graph. |

In the below example, we will use the "mtcars" dataset present in the R environment. We will use its two columns only, i.e., "mpg" and "cyl". The below example will create a boxplot graph for the relation between mpg and cyl, i.e., miles per gallon and number of cylinders, respectively.

**Program:**

# Giving a name to the chart file.

png(file = "boxplot.png")

# Plotting the chart.

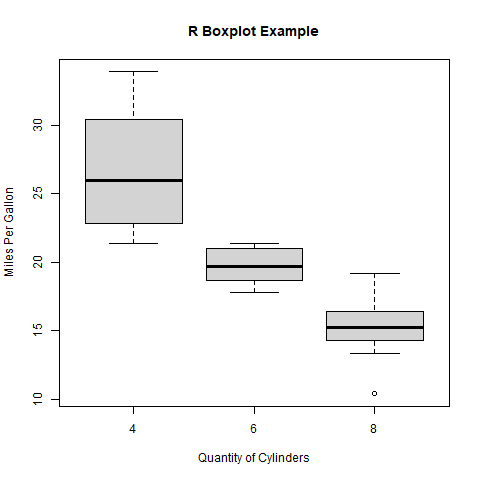
boxplot(mpg ~ cyl, data = mtcars, xlab = "Quantity of Cylinders",

ylab = "Miles Per Gallon", main = "R Boxplot Example")

# Save the file.

dev.off()

**Output:**



## **Boxplot using notch**

In R, we can draw a boxplot using a notch. It helps us to find out how the medians of different data groups match with each other. Let's see an example to understand how a boxplot graph is created using notch for each of the groups

**Program:**

# Giving a name to our chart.

png(file = "boxplot\_using\_notch.png")

# Plotting the chart.

boxplot(mpg ~ cyl, data = mtcars,

        xlab = "Quantity of Cylinders",

        ylab = "Miles Per Gallon",

        main = "Boxplot Example",

        notch = TRUE,

        varwidth = TRUE,

        ccol = c("green","yellow","red"),

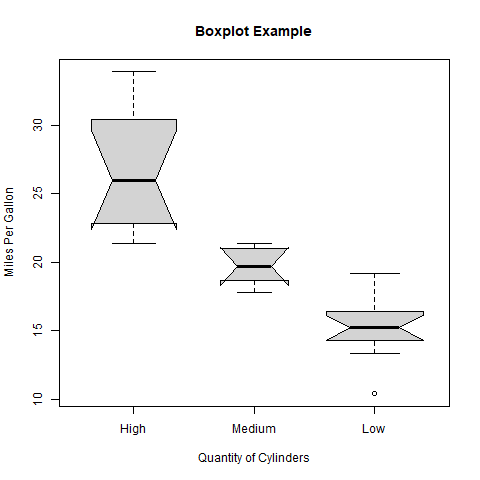
        names = c("High","Medium","Low")

)

# Saving the file.

dev.off()

**Output:**



# R Bar Charts

A bar chart is a pictorial representation in which numerical values of variables are represented by length or height of lines or rectangles of equal width. A bar chart is used for summarizing a set of categorical data. In bar chart, the data is shown through rectangular bars having the length of the bar proportional to the value of the variable.

In R, we can create a bar chart to visualize the data in an efficient manner. For this purpose, R provides the barplot() function, which has the following syntax:

barplot(h,x,y,main, names.arg,col)

Here,

|  |  |  |
| --- | --- | --- |
| **S.No** | **Parameter** | **Description** |
| 1. | H | A vector or matrix which contains numeric values used in the bar chart. |
| 2. | xlab | A label for the x-axis. |
| 3. | ylab | A label for the y-axis. |
| 4. | main | A title of the bar chart. |
| 5. | names.arg | A vector of names that appear under each bar. |
| 6. | col | It is used to give colors to the bars in the graph. |

Program:

# Creating the data for Bar chart

H<- c(12,35,54,3,41)

# Giving the chart file a name

png(file = "bar\_chart.png")

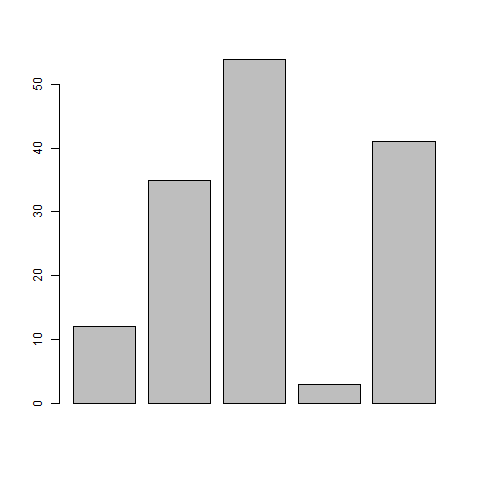
# Plotting the bar chart

barplot(H)

# Saving the file

dev.off()

**Output:**



## **Labels, Title &Colors**

Like pie charts, we can also add more functionalities in the bar chart by-passing more arguments in the barplot() functions. We can add a title in our bar chart or can add colors to the bar by adding the main and col parameters, respectively. We can add another parameter i.e., args.name, which is a vector that has the same number of values, which are fed as the input vector to describe the meaning of each bar.

**Program:**

# Creating the data for Bar chart

H <- c(12,35,54,3,41)

M<- c("Feb","Mar","Apr","May","Jun")

# Giving the chart file a name

png(file = "bar\_properties.png")

# Plotting the bar chart

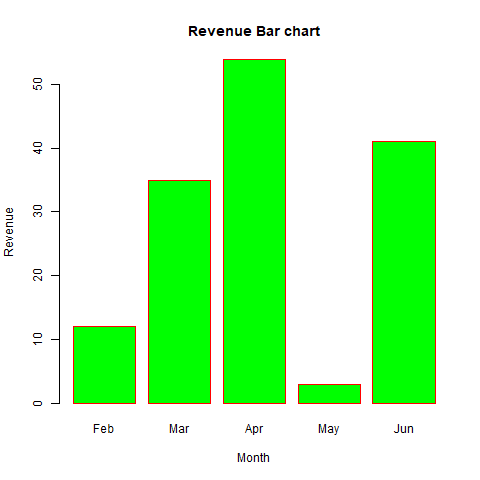
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="Green",

main="Revenue Bar chart",border="red")

# Saving the file

dev.off()

**Output:**



**Additional Lab Experiments:**

# R Histogram

A histogram is a type of bar chart which shows the frequency of the number of values which are compared with a set of values ranges. The histogram is used for the distribution, whereas a bar chart is used for comparing different entities. In the histogram, each bar represents the height of the number of values present in the given range.

For creating a histogram, R provides hist() function, which takes a vector as an input and uses more parameters to add more functionality. There is the following syntax of hist() function:

hist(v,main,xlab,ylab,xlim,ylim,breaks,col,border)

Here,

|  |  |  |
| --- | --- | --- |
| **S.No** | **Parameter** | **Description** |
| 1. | v | It is a vector that contains numeric values. |
| 2. | main | It indicates the title of the chart. |
| 3. | col | It is used to set the color of the bars. |
| 4. | border | It is used to set the border color of each bar. |
| 5. | xlab | It is used to describe the x-axis. |
| 6. | ylab | It is used to describe the y-axis. |
| 7. | xlim | It is used to specify the range of values on the x-axis. |
| 8. | ylim | It is used to specify the range of values on the y-axis. |
| 9. | breaks | It is used to mention the width of each bar. |

**Program:**

# Creating data for the graph.

v <- c(12,24,16,38,21,13,55,17,39,10,60)

# Giving a name to the chart file.

png(file = "histogram\_chart.png")

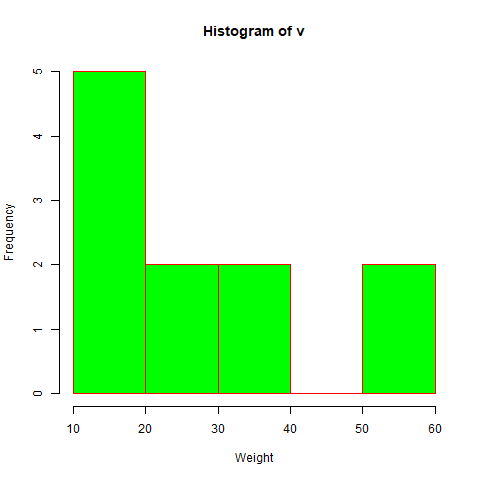
# Creating the histogram.

hist(v,xlab = "Weight",ylab="Frequency",col = "green",border = "red")

# Saving the file.

dev.off()

**Output:**



# R Line Graphs

A line graph is a pictorial representation of information which changes continuously over time. A line graph can also be referred to as a line chart. Within a line graph, there are points connecting the data to show the continuous change. The lines in a line graph can move up and down based on the data. We can use a line graph to compare different events, information, and situations.

A line chart is used to connect a series of points by drawing line segments between them. Line charts are used in identifying the trends in data. For line graph construction, R provides plot() function, which has the following syntax:

plot(v,type,col,xlab,ylab)

Here,

|  |  |  |
| --- | --- | --- |
| **S.No** | **Parameter** | **Description** |
| 1. | v | It is a vector which contains the numeric values. |
| 2. | type | This parameter takes the value? I? to draw only the lines or? p? to draw only the points and "o" to draw both lines and points. |
| 3. | xlab | It is the label for the x-axis. |
| 4. | ylab | It is the label for the y-axis. |
| 5. | main | It is the title of the chart. |
| 6. | col | It is used to give the color for both the points and lines |

**Program:**

# Creating the data for the chart.

v <- c(13,22,28,7,31)

# Giving a name to the chart file.

png(file = "line\_graph.jpg")

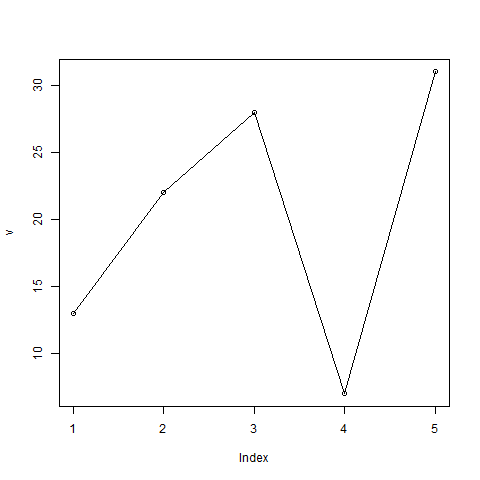
# Plotting the bar chart.

plot(v,type = "o")

# Saving the file.

dev.off()

**Output:**



## **Line Chart Title, Color, and Labels**

Like other graphs and charts, in line chart, we can add more features by adding more parameters. We can add the colors to the lines and points, add labels to the axis, and can give a title to the chart.

**Program:**

# Creating the data for the chart.

v <- c(13,22,28,7,31)

# Giving a name to the chart file.

png(file = "line\_graph\_feature.jpg")

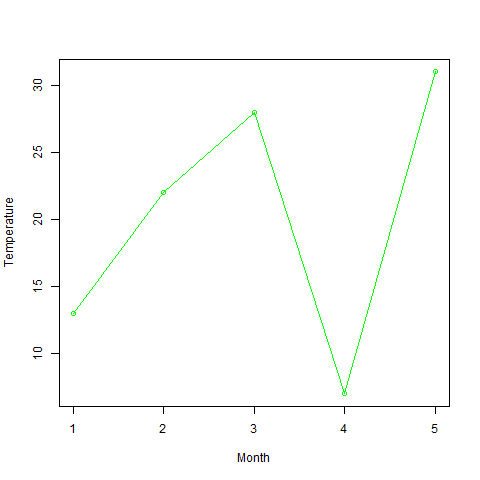
# Plotting the bar chart.

plot(v,type = "o",col="green",xlab="Month",ylab="Temperature")

# Saving the file.

dev.off()

**Output:**



**8(b) Implement R Script to perform mean, median, mode, range, summary, variance,**

**standard deviation operations.**

#### Measure of central tendency

It represents the whole set of data by single value.It gives us the location of central points. There are three main measures of central tendency:

* Mean
* Mode
* Median

**Mean:** Calculate sum of all the values and divide it with the total number of values in the data set.

**>** x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6) # our data set

> mean.result = mean(x) # calculate mean

> print (mean.result)

[1] 2.8

**Median:** The middle value of the data set.

> x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6) # our data set

> median.result = median(x) # calculate median

> print (median.result)

[1] 2.5

**Mode:** The most occurring number in the data set. For calculating mode, there is no default function in R. So, we have to create our own custom function.

> mode <- function(x) {

+ ux <- unique(x)

+ ux[which.max(tabulate(match(x, ux)))]

+ }

> x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6) # our data set

> mode.result = mode(x) # calculate mode (with our custom function named ‘mode’)

> print (mode.result)

[1] 1

**Variance:** How far a set of data values are spread out from their mean.

> variance.result = var(x) # calculate variance

> print (variance.result)

[1] 2.484211

**Standard Deviation:** A measure that is used to quantify the amount of variation or dispersion of a set of data values.

> sd.result = sqrt(var(x)) # calculate standard deviation

> print (sd.result)

[1] 1.576138

**Week 9**

a) Implement R Script to perform Normal, Binomial distributions.

b) Implement R Script to perform correlation, Linear and multiple regression.

**9 a) Implement R Script to perform Normal, Binomial distributions.**

**Normal Distribution**: x <- seq(-10, 10, by = .1)

y <- dnorm(x, mean = 2.5, sd = 0.5)

png(file = "dnorm.png")

plot(x,y)

dev.off()

**Binomial Distribution**

x <- seq(0,50,by = 1)

y <- dbinom(x,50,0.5)

png(file = "dbinom.png")

plot(x,y)

dev.off()

**9 b) Implement R Script to perform Linear and multiple regression.**

**Linear Regression:**

x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

relation <- lm(y~x)

png(file = "linearregression.png")

plot(y,x,col = "blue",main = "Height & Weight Regression",

abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")

dev.off()

**Multiple Regression:**

input <- mtcars[,c("mpg","disp","hp","wt")]

model <- lm(mpg~disp+hp+wt, data = input)

print(model)

cat("# # # # The Coefficient Values # # # ","\n")

a <- coef(model)[1]

print(a)

Xdisp <- coef(model)[2]

Xhp <- coef(model)[3]

Xwt <- coef(model)[4]

print(Xdisp)

print(Xhp)

print(Xwt)

**Week 10**

Introduction to Non-Tabular Data Types: Time series, spatial data, Network data.

Data Transformations: Converting Numeric Variables into Factors, Date Operations,

String Parsing, Geocoding.

**Time series:**

Time series is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day.

The basic syntax for **ts()** function in time series analysis is −

timeseries.object.name <- ts(data, start, end, frequency)

Following is the description of the parameters used −

* **data** is a vector or matrix containing the values used in the time series.
* **start** specifies the start time for the first observation in time series.
* **end** specifies the end time for the last observation in time series.
* **frequency** specifies the number of observations per unit time.

# Get the data points in form of a R vector.

rainfall <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)

# Convert it to a time series object.

rainfall.timeseries <- ts(rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.

print(rainfall.timeseries)

# Give the chart file a name.

png(file = "rainfall.png")

# Plot a graph of the time series.

plot(rainfall.timeseries)

# Save the file.

dev.off()

**Spatial Data**

library(rjson)

library(ggmap)

library(RgoogleMaps)

library(png)

library(sp)

library(RColorBrewer)

library(rgdal)

gadm <- readRDS("F:/IND\_adm1.rds", refhook = NULL)

ind1 = gadm

spplot(ind1, "NAME\_1", scales=list(draw=T), colorkey=F, main="India")

ind1$NAME\_1 = as.factor(ind1$NAME\_1)

ind1$fake.data = runif(length(ind1$NAME\_1))

spplot(ind1,"NAME\_1", col.regions=rgb(0,ind1$fake.data,0), colorkey=T, main="Indian States")

TN=ind1[ind1$NAME\_1=="Tamil Nadu",]

spplot(TN,"NAME\_1", col.regions=rgb(0,0,1), main = "Tamil Nadu, India",scales=list(draw=T), colorkey =F)

TS=ind1[ind1$NAME\_1=="Telangana",]

spplot(TS,"NAME\_1", col.regions=rgb(0,0,1), main = "Telangana, India",scales=list(draw=T), colorkey =F)

AP=ind1[ind1$NAME\_1=="Andhra Pradesh",]

spplot(AP,"NAME\_1", col.regions=rgb(0,0,1), main = "Andhra Pradesh,India",scales=list(draw=T), colorkey =F)

ind2=readRDS("F:/IND\_adm2.rds")

TN\_districts = (ind2[ind2$NAME\_1=="Tamil Nadu",])

TN\_districts$NAME\_2=as.factor(TN\_districts$NAME\_2)

col = rainbow(length(levels(TN\_districts$NAME\_2)))

spplot(TN\_districts,"NAME\_2", main="The Districts of TamilNadu", col.regions=col, colorkey=T)

AP\_districts = (ind2[ind2$NAME\_1=="Andhra Pradesh",])

AP\_districts$NAME\_2=as.factor(AP\_districts$NAME\_2)

col = rainbow(length(levels(AP\_districts$NAME\_2)))

spplot(AP\_districts,"NAME\_2", main="The Districts of Andhra Pradesh", col.regions=col, colorkey=T)

ind3=readRDS("F:/IND\_adm3.rds")

TN\_TALUKS=ind3[ind3$NAME\_1=='Tamil Nadu',]

TN\_TALUKS$NAME\_3<-as.factor(TN\_TALUKS$NAME\_3)

col1=rainbow(length(levels(TN\_TALUKS$NAME\_3)))

spplot(TN\_TALUKS,"NAME\_3",main = "Taluk, District - TN", colorkey=T,col.regions=col,scales=list(draw=T))

AP\_TALUKS=ind3[ind3$NAME\_1=='Andhra Pradesh',]

AP\_TALUKS$NAME\_3<-as.factor(AP\_TALUKS$NAME\_3)

col1=rainbow(length(levels(AP\_TALUKS$NAME\_3)))

spplot(AP\_TALUKS,"NAME\_3",main = "Taluk, District - AP", colorkey=T,col.regions=col,scales=list(draw=T)) #Now this is too cluttery

**Network Data**

install.packages("igraph")

## Load package

library(igraph)

## Create undirected graphs

g <- graph\_from\_literal(1-2, 1-3, 1-7, 3-4, 2-3, 2-4, 3-5, 4-5, 4-6, 4-7, 5-6, 5-8, 6-7, 7-8)

## Create directed graphs using addition or substraction operators

dg <- graph\_from\_literal(JFK-+PEK, JFK-+CDG, PEK++CDG)

## check out the vertices and the order

V(g)

vcount(g)

## check out the edges and the size

E(g)

ecount(g)

## summary

g$name <- "undirected graph"

print\_all(g)

plot(g)

dg$name <- "directed graph"

print\_all(dg)

plot(dg)

V(g)$name <-c("Adam","Judy","Bobby","Sam","Frank","Jay","Tom","Jerry")

plot(g)

**Converting Numeric Variables into Factors**

# Data Vector 'V'

V = c("North", "South", "East", "East")

# Convert vector 'V' into a factor

drn <- factor(V)

# Converting a factor into a numeric vector

as.numeric(drn)

# Creating a Factor

soap\_cost <- factor(c(29, 28, 210, 28, 29))

# Converting Factor to Numeric

as.numeric(soap\_cost)

**Date Operations**

xd <- as.Date("2012-07-27")

> xd

[1] "2012-07-27"

> str(xd)

Date[1:1], format: "2012-07-27"

> weekdays(xd)

[1] "Friday"

> xd + 7

[1] "2012-08-03"

> xd + 0:6

[1] "2012-07-27" "2012-07-28" "2012-07-29" "2012-07-30"

[5] "2012-07-31" "2012-08-01" "2012-08-02"

weekdays(xd + 0:6)

[1] "Friday" "Saturday" "Sunday" "Monday"

[5] "Tuesday" "Wednesday" "Thursday"

startDate <- as.Date("2012-01-01")

> xm <- seq(startDate, by="2 months", length.out=6)

> xm

[1] "2012-01-01" "2012-03-01" "2012-05-01" "2012-07-01"

[5] "2012-09-01" "2012-11-01"

months(xm)

[1] "January" "March" "May" "July"

[5] "September" "November"

> quarters(xm)

[1] "Q1" "Q1" "Q2" "Q3" "Q3" "Q4"

**String Parsing**

fruits <- c(

"apples and oranges and pears and bananas",

"pineapples and mangos and guavas"

)

str\_split(fruits, " and ")

#> [[1]]

#> [1] "apples" "oranges" "pears" "bananas"

#>

#> [[2]]

#> [1] "pineapples" "mangos" "guavas"

#>

str\_split(fruits, " and ", simplify = TRUE)

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

#> [1,] "apples" "oranges" "pears" "bananas"

#> [2,] "pineapples" "mangos" "guavas" ""

# Specify n to restrict the number of possible matches

str\_split(fruits, " and ", n = 3)

#> [[1]]

#> [1] "apples" "oranges" "pears and bananas"

#>

#> [[2]]

#> [1] "pineapples" "mangos" "guavas"

#>

str\_split(fruits, " and ", n = 2)

#> [[1]]

#> [1] "apples" "oranges and pears and bananas"

#>

#> [[2]]

#> [1] "pineapples" "mangos and guavas"

#>

# If n greater than number of pieces, no padding occurs

str\_split(fruits, " and ", n = 5)

#> [[1]]

#> [1] "apples" "oranges" "pears" "bananas"

#>

#> [[2]]

#> [1] "pineapples" "mangos" "guavas"

#>

# Use fixed to return a character matrix

str\_split\_fixed(fruits, " and ", 3)

#> [,1] [,2] [,3]

#> [1,] "apples" "oranges" "pears and bananas"

#> [2,] "pineapples" "mangos" "guavas"

str\_split\_fixed(fruits, " and ", 4)

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

#> [1,] "apples" "oranges" "pears" "bananas"

#> [2,] "pineapples" "mangos" "guavas" NA

# str\_split\_n extracts only a single piece from a string

str\_split\_n(fruits, " and ", 1)

#> [1] "apples" "pineapples"

str\_split\_n(fruits, " and ", 3)

#> [1] "pears" "guavas"

**Week-11**

**a) Mislabeled variables**: View all variable labels with the [names()](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/names) function. Our example dataset has long labels that will be difficult to call in the code to come. We can modify them with [dplyr’s](https://dplyr.tidyverse.org/) [rename()](https://dplyr.tidyverse.org/reference/rename.html).

**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*

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)*

*#code*

*install.packages("deplyr")*

*library(deplyr)*

*data<-read.csv("mtcars.csv")*

*dataf<-data.frame(data)*

*dataf*

*rename(dataf,miles=mpg)*

**b) Faulty data types:**These can be determined by either the [str()](https://www.rdocumentation.org/packages/utils/versions/3.6.2/topics/str) function or the more explicit [typeof()](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/typeof) function.

typeof(dataf$mpg)

typeof(dataf$cyl)

typeof(dataf$disp)

typeof(dataf$hp)

typeof(dataf$drat)

typeof(dataf$wt)

typeof(dataf$qsec)

typeof(dataf$vs)

typeof(dataf$am)

typeof(dataf$gear)

typeof(dataf$carb)

**c) Non-unique ID numbers**: we can remove duplicates with the [duplicated()](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/duplicated) function. Or we can use distinct() to identify distinct values.

#with duplicated()  
dataf <- dataf[!duplicated(dataf$disp), ]#with distinct()  
df <- df %>% distinct(disp, .keep\_all = TRUE)

String inconsistencies:- Also, beware of missing values erroneously represented by character “NA” values rather than NA data types. Fix instances with the following code:

df <- df %>% na\_if(gender, "NA")

**Step 3: Check for data irregularities**

Next, we’ll evaluate the dataset for irregularities, which consist of accuracy concerns like **invalid values** and **outliers**. Again, these are two common pitfalls in messy data frames, but be aware of irregularities specific to your own data.

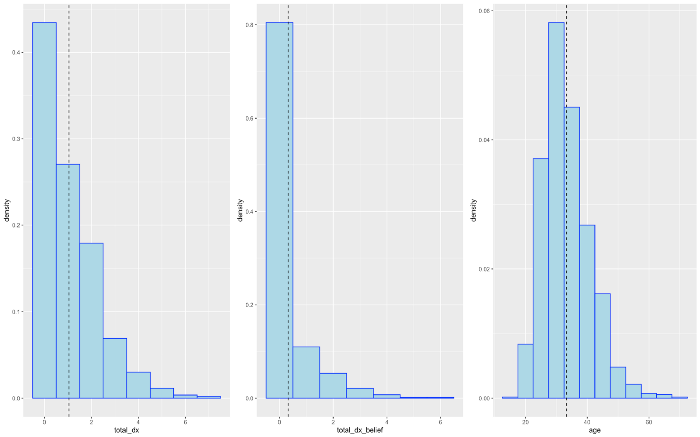
**a) Invalid values**: These are responses that don’t make logical sense.

One common example is age. Our dataset consists of responses from tech employees, meaning anyone reporting an age older than 80 or younger than 15 is likely to be an entry error. Let’s take a look:

It is safe to say that a 3-yr-old and 323-yr-old did not complete an employee survey. To remove the invalid entries, we can use the following code:

df <- df[-c(which(df$age > 80 | df$age < 15)), ]

**b) Outliers**: This is a topic with much debate. After a little feature engineering (check out the full data cleaning script [here](https://github.com/emiburns/mental-health-in-tech-medium-project) for reference), our dataset has 3 continuous variables: age, the number of diagnosed mental illnesses each respondent has, and the number of believed mental illnesses each respondent has. To get a feeling for how the data is distributed, we can plot histograms for each variable:



Both “total\_dx” and “total\_dx\_belief” are heavily skewed. If we wanted to mitigate the impact of extreme outliers, there are 3 common ways to do so: delete the outlier, replace the value (aka Winsorize), or do nothing.

**Delete the observation**: Locate and remove the observation with the extreme value. This is common when dealing with extreme values that are clearly the result of human entry error (like the 323-year value previously entered in our “age” column). However, be careful when this is not the case as deleting observations can lead to a loss of important information.

**Winsorize**: When an outlier is negatively impacting your model assumptions or results, you may want to replace it with a less extreme maximum value. In Winsorizing, values outside a predetermined percentile of the data are identified and set to said percentile. The following is an example of 95% Winsorization with our dataset:

#looking at number of values above 95th percentile   
sum(df$total\_dx > quantile(df$total\_dx, .95))

df <- df %>% mutate(wins\_total\_dx = Winsorize(total\_dx))

## Step 4: Decide how to deal with missing values

**a) Remove the variable**. Delete the column with the NA value(s). In projects with large amounts of data and few missing values, this may be a valid approach. However, if you’re dealing with a smaller dataset and/or a multitude of NA values, keep in mind removing variables can result in a significant loss of information.

**b) Remove the observation**. Delete the row with the NA value. Again, this may be an acceptable approach in large projects but beware of the potential loss of valuable information. To remove observations with missing values, we can easily employ the dplyr library again:

#identifying the rows with NAs  
rownames(df)[apply(df, 2, anyNA)]#removing all observations with NAs  
df\_clean <- df %>% na.omit()

**c) Impute the missing value**. Substitute NA values with inferred replacement values. We can do so using the mean, median, or mode of a given variable like so:

for(i in 1:ncol(df)){  
 df[is.na(df[,i]), i] <- mean(df[,i], na.rm = TRUE)  
}

We can additionally impute continuous values using predictive methods such as **linear regression**, or impute categorical values using methods like **logistic regression** or **ANOVA**.

[KNN imputation](https://www.rdocumentation.org/packages/impute/versions/1.46.0/topics/impute.knn) offers yet another probable alternative to imputing either continuous or categorical missing values, but keep in mind it can be time-consuming and highly dependent on the chosen k-value.

#imputing missing values with the caret package's knn method  
df\_preprocess <- preProcess(df %>% dplyr::select(primary\_role),  
 method = c("knnImpute"),  
 k = 10,  
 knnSummary = mean)df\_impute <- predict(df\_preprocess, df, na.action = na.pass)

**Week-12**

Data sources: SQLite examples for relational databases, Loading SPSS and SAS files, Reading from Google Spreadsheets, API and web scraping examples

# SQLite examples

install.packages("RSQLite")

#code

library(DBI)

# Create an ephemeral in-memory RSQLite database

con <- dbConnect(RSQLite::SQLite(), ":memory:")

dbListTables(con)

dbWriteTable(con, "mtcars", mtcars)

dbListTables(con)

dbListFields(con, "mtcars")

dbReadTable(con, "mtcars")

# You can fetch all results:

res <- dbSendQuery(con, "SELECT \* FROM mtcars WHERE cyl = 4")

dbFetch(res)

dbClearResult(res)

# Or a chunk at a time

res <- dbSendQuery(con, "SELECT \* FROM mtcars WHERE cyl = 4")

while(!dbHasCompleted(res)){

chunk <- dbFetch(res, n = 5)

print(nrow(chunk))

}

# Clear the result

dbClearResult(res)

# Disconnect from the database

dbDisconnect(con)