R is a software environment which is used to analyze statistical information and graphical representation. R allows us to do modular programming using functions.

Our R tutorial includes all topics of R such as introduction, features, installation, rstudio ide, variables, data types, operators, if statement, vector, data handling, graphics, statistical modelling, etc.

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

What is R Programming

"R is an interpreted computer programming language which was created by Ross Ihaka and Robert Gentleman at the University Of Auckland, New Zealand."

It is also a software environment used to analyze **statistical information**, **graphical representation**, **reporting**, and **data modeling**.

R not only allows us to do branching and looping but also allows to do modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python, and FORTRAN languages to improve efficiency.

In the present era, R is one of the most important tool which is used by researchers, data analyst, statisticians, and marketers for retrieving, cleaning, analyzing, visualizing, and presenting data.

History of R Programming

- The history of R goes back about 20-30 years ago. R was developed by Ross lhaka and Robert Gentleman in the University of Auckland, New Zealand, and the R Development Core Team currently develops it.

- This programming language name is taken from the name of both the developers. The first project was considered in 1992. The initial version was released in 1995, and in 2000, a stable beta version was released.

Features of R programming

R is a domain-specific programming language which aims to do data analysis. It has some unique features which make it very powerful. The most important arguably being the notation of vectors. These vectors allow us to perform a complex operation on a set of values in a single command. There are the following features of R programming:

1. It is a simple and effective programming language which has been well developed.
2. It is data analysis software.
3. It is a well-designed, easy, and effective language which has the concepts of user-defined, looping, conditional, and various I/O facilities.
4. It has a consistent and incorporated set of tools which are used for data analysis.
5. For different types of calculation on arrays, lists and vectors, R contains a suite of operators.
6. It provides an effective data handling and storage facility.
7. It is an open-source, powerful, and highly extensible software.
8. It provides highly extensible graphical techniques.
9. It allows us to perform multiple calculations using vectors.
10. R is an interpreted language.

Why use R Programming?

- 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

Applications of R

There are several-applications available in real-time. Some of the popular applications are as follows:

- Facebook
- Google
- Twitter
- HRDAG
- Sunlight Foundation
- RealClimate
- NDAA

- XBOX ONE

Prerequisite

R programming is used for statistical information and data representation. So it is required that we should have the knowledge of statistical theory in mathematics. Understanding of different types of graphs for data representation and most important is that we should have prior knowledge of any programming.

R Syntax

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

Ex. “hello” or ‘hello’ otherwise error

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

Ex. 5 or 15 or 5+5.

Print

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

print("Hello World!")

R 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 #.

Ex. # This is a comment "Hello World!".

Multiline Comments

Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments.

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:

Ex:

```
name <- "sandy"

age <- 21

gender <- 'male'

course <- 'bsc'

name   # output "John"

age      # output 40

gender # output male

course # output bsc
```

Note:

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

Ex.
```
name <- "John Doe"
name
```

Concatenate Elements

You can also concatenate, or join, two or more elements, by using the paste() function.

To combine both text and a variable, R uses comma (,).

Ex. text <- "awesome"

paste("R is", text).

Note: You can also use , to add a variable to another variable.

Ex.

text1 <- "R is"

text2 <- " a programming language"

paste(text1, text2)

Note:

- For numbers, the + character works as a mathematical operator.
- If you try to combine a string (text) and a number, R will give you an error.

Ex.

num1 <- 5

num2 <- 10

```
num1 + num2
```

Multiple Variables

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

Ex. Assign the same value to multiple variables in one line

```
var1 <- var2 <- var3 <- "banana"

# 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...).

Note: 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.

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/string ("k", "R is exciting", "FALSE", "11.5")
- logical/booleans (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

Example:

```
# numeric

x <- 10.5

class(x)


# integer

x <- 1000L

class(x)


# complex

x <- 9i + 3

class(x)


# character/string

x <- "R is exciting"

class(x)


# logical

x <- TRUE
```

```
class(x)
```

Numbers

There are three number types in R:

- numeric
- integer
- complex

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

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

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:

Ex:

```
x <- 10L
```

```
y <- 550L

x

y

class(x)

class(y)
```

Complex

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

Example:

```
x <- 7+10i
y <- 6i
z<- 0

x

y
z


class(x)
class(y)
class(z)
```

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)
```

Simple Math

In R, you can use **operators** to perform common mathematical operations on numbers.

The + operator is used to add together two values and the - operator is used for subtraction.

Built-in Math Functions

R also has many built-in math functions that allows you to perform mathematical tasks on numbers.

For example, the min() and max() functions can be used to find the lowest or highest number in a set:

```
max(25, 10, 15)

min(5, 10, 11)
```

sqrt()

The sqrt() function returns the square root of a number.

abs()

The abs() function returns the absolute (positive) value of a number.

ceiling() and floor()

The ceiling() function rounds a number upwards to its nearest integer, and the floor() function rounds a number downwards to its nearest integer, and returns the result.

Example:

```
ceiling(1.4) =2

floor(1.4)=1

abs(-4.7)=4.7

sqrt(16)
```

Multiline Strings

- You can assign a multiline string to a variable like this:

Example:

```
str <- "R is programming language,
, Go for a walk, don’t go outside."
str
```

Note:

1. However, note that R will add a "**\n**" at the end of each line break. This is called an escape character, and the **n** character indicates a **new line**.
2. If you want the line breaks to be inserted at the same position as in the code, use the cat() function:

Example:

```
str <- "R is programming language , Go for a walk, don’t go outside."

Cat(str)
```

String Length

- There are many usesful string functions in R.
- For example, to find the number of characters in a string, use the nchar() function:
- str <- "R is programming language , Go for a walk, don’t go outside."
- nchar(str)

Check a String

Use the grepl() function to check if a character or a sequence of characters are present in a string:

For example:

str <- "R is programming language , Go for a walk, don’t go outside."
grepl("R", str)

Combine Two Strings

Use the paste() function to merge/concatenate two strings:

Ex.

x <-'hello'

y<-"r programming"

paste(x, y)

Booleans (Logical Values)

- In programming you often need to know if an expression is **True** or **False**.
- You can evaluate any expression in R, and get one of two answers, TRUE or FALSE.
- When you compare two values, the expression is evaluated and R returns the logical answer:

Ex:

```
a <- 20

b <- 33

b > a
```

Operators

- Operators are used to perform operations on variables and values.

R divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Miscellaneous operators

Arithmetic operators

Operator	**Name**
+	Addition
-	Subtraction
*	Multiplication
/	Division

^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

Note:

- <<- is a global assigner.
- It is also possible to turn the direction of the assignment operator.
- x <- 3 is equal to 3 -> x.

Example:

```
x <- 4

x <<- 6

4 -> x

6 ->> x

x
```

R Comparison Operators

Comparison operators are used to compare two values:

Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

R Logical Operators

Logical operators are used to combine conditional statements:

Element-wise Logical AND operator.

Operator	Description

&	Element-wise Logical AND operator.
&&	Logical AND operator - Returns TRUE if both statements are TRUE
|	Element wise- Logical OR operator.
||	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

R If ... Else

Conditions and If Statements

- R supports the usual logical conditions from mathematics.

Operator	**Name**
==	Equal
!=	Not equal

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- These conditions can be used in several ways, most commonly in "if statements" and loops.
- An "if statement" is written with the if keyword, and it is used to specify a block of code to be executed if a condition is TRUE

Syntax: if(boolean_expression) {   // If the boolean expression is true, then statement(s) will be executed.

Example:

```
a <- 33

b <- 200


if (b > a) {

  print("b is greater than a")
}
```

Example:

```
    x <-22
    y<-24
    count=0
    if(x<y)

    {
       cat(x, "is a smaller number\n")
       count=1

}

    if(count==1){

       cat("Block is successfully execute")

    }
```

If-else statement

- In the if statement, the inner code is executed when the condition is true. The code which is outside the if block will be executed when the if condition is false.

- R programming treats any non-zero and non-null values as true, and if the value is either zero or null, then it treats them as false.

Syntax:

```
  if(boolean_expression) {

  // statement(s) will be executed if the boolean expression is true.

  } else {

      // statement(s) will be executed if the boolean expression is false.

    }
```

Example:

```
a<- 'u'

if(a=='a'||a=='e'||a=='i'||a=='o'||a=='u'||a=='A'||a=='E'||a=='I'||a=='O'||a=='U'){

   cat ("character is a vowel\n")

}

else{

   cat("character is a constant")

     }

     cat("character is =",a)

     }
```

Else If

This statement is also known as nested if-else statement. The if statement is followed by an optional else if..... else statement. This statement is used to test various condition in a single if......else if statement. Some points to remember:

1. **if** statement can have either zero or one **else** statement and it must come after any **else if's** statement.
2. **if** statement can have many **else if's** statement and they come before the else statement.
3. Once an **else if** statement succeeds, none of the remaining **else if's** or **else's** will be tested.

The basic syntax of else-if statement is as follows:

```
if(boolean_expression 1) {

  // This block executes when the boolean expression 1 is true.

} else if( boolean_expression 2) {

  // This block executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {

   // This block executes when the boolean expression 3 is true.

} else {

   // This block executes when none of the above condition is true.

}
```

Example:

```
x <- 35
y <- 35
if(x > y) {
print("x is greater than y")
} else if (x == y) {
print("x and y are equal")
}
```

Example:

```
    marks =83;

    if(marks>75){

        print("First class")

    }else if(marks>65) {

        print("Second class")

    }else if(marks>55) {

        print("Third class")

    }else{

       print("Fail")

    }
```

R Switch Statement

- A switch statement is a selection control mechanism that allows the value of an expression to change the control flow of program execution.

We have some key points which are as follows:

- If expression type is a character string, the string is matched to the listed cases.
- If there is more than one match, the first match element is used.
- No default case is available.
- If no case is matched, an unnamed case is used.

1) Based on Index

- If the cases are values like a character vector, and the expression is evaluated to a number than the expression's result is used as an index to select the case.

2) Based on Matching Value

- When the cases have both case value and output value like ["case_1"="value1"], then the expression value is matched against case values. If there is a match with the case, the corresponding value is the output.

Syntax:

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

```
x <- switch( 3, "Shubh", "Navjot", "CP",    "Parlu" )
```

```
print(x)
```

Example:

```
y = "18"
a=10
b=2
x = switch(
   y,
   "9"=cat("Addition=",a+b),
   "12"=cat("Subtraction =",a-b),
   "18"=cat("Division= ",a/b),
   "21"=cat("multiplication =",a*b)
)
print (x)
```

R next Statement

The next statement is a statement which skips the current iteration of a loop without terminating it. When the next statement is encountered, the R parser skips further evaluation and starts the next iteration of the loop.

This statement is mostly used with for loop and while loop.

Example:

```
a <- 1
repeat {
  if(a == 10)
    break
  if(a == 5){
      next
  }
  print(a)
  a <- a+1
}
```

R Break Statement

- In the R language, the break statement is used to break the execution and for an immediate exit from the loop.

There are basically two usages of break statement which are as follows:

- When the break statement is inside the loop, the loop terminates immediately and program control resumes on the next statement after the loop.
- It is also used to terminate a case in the switch statement.

Example:

```
a <- 1
repeat {
  print("hello");
  if(a >= 5)
```

```
        break
  a<-a+1
}
```

Data Structures in R Programming

- Data structure are the objects which we will manipulate in our day-to-day basis in R.

Types of data structure in R

Vector

- 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.

- 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:

1. Atomic vectors

2. Lists

They have three common properties, i.e., **function type, function length**, and **attribute function**.

Difference between atomic vectors and lists:

- There is only one difference in atomic vectors and lists.

- In an atomic vector, all the elements are of the same type, but in the list, the elements are of different data types.

How to create a vector 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.

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

Using the colon(:) operator

Example:

```
numbers <- 1:10

numbers
```

Example:

```
numbers1 <- 1.5:6.5

numbers1
```

Example:

```
numbers1 <- 1.5:6.3
```

```
numbers1
```

Using the seq() function

- 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:

```
x<-seq(1,4,by=0.5)

x

class(x)
```

Example:

```
x<-seq(1,4,length.out=6)

x

class(x)
```

Atomic vectors in R

- In R, there are four types of atomic vectors.
- Atomic vectors are created with the help of **c()** function.
- These atomic vectors are as follows:

Numeric vector

- A vector which contains numeric elements is known as a numeric vector.
- The decimal values are known as numeric data types in R.

Ex.

```
x<-45.5

y<-c(10.1, 10.2, 33.2)

x

y

class(x)

class(y
```

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 are two ways to assign an integer value to a variable, i.e., by using **as.integer()** function and appending L to the value.

Example:

```
x<-as.integer(5)

y<-5L

int_x<-c(1,2,3,4,5)

int_x<-as.integer(int_x)

int_y<-c(1L,2L,3L,4L,5L)

x

y


class(x)

class(y)

class(int_x)

class(int_y)
```

character data type

- 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('').

Example:

```
a<-'shubham'

b<-"Arpita"

c<-65

c<-as.character(c)

a

b

c

char_d<-c(1,2,3,4,5)

char_d<-as.character(char_d)

char_e<-c("shubham","arpita","nishka","vaishali")

char_d

class(a)

class(b)

class(c)

class(d)

class(e)
```

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.

Indexing with integer vector

- We perform indexing by specifying an integer value in square braces [] next to our vector.
- Indexing starts with 1 in R.

Example:

```
v<-seq(1,4,length.out=6)

v

v[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.

Example:

```
char_vec<-c("shubham"=22,"arpita"=23,"vaishali"=25)

char_vec

Char_vec["arpita"]
```

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)]
```

Vector Operation

- In R, there are various operations which are 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.

Example:

```
p<-c(1,2,4,5,7,8)

q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")

r<-c(p,q)
```

2) Arithmetic operations

- We can perform all the arithmetic operations on vectors.
- The arithmetic operations are performed member-by-member on vectors. We can add, subtract, multiply, or divide two vectors.

Example:

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

b<-c(2,4,8)

a+b

a-b
```

```
a/b
a%%b
```

3) Logical Index vector

- With the help of the logical index vector in R, we can form a new vector from a given vector. This vector has the same length as the original vector.
- The vector members are TRUE only when the corresponding members of the original vector are included in the slice; otherwise, it will be false.

Example:

```
a<-c("Shubham","Arpita","Nishka","Vaishali","Sumit","Gunjan")
b<-c(TRUE,FALSE,TRUE,TRUE,FALSE,FALSE)
a[b]
```

4) Numeric Index

- In R, we specify the index between square braces [ ] for indexing a numerical value.
- If our index is negative, it will return us all the values except for the index which we have specified.

Example:

```
q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
q[2]
q[-7]
q[-1]
q[-2]
q[15]
```

5) Duplicate Index

An index vector allows duplicate values which means we can access one element twice in one operation. Let's see an example to understand how a duplicate index works.

```
q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
q[c(2,4,4,3)]
```

6) Range Indexes

Range index is used to slice our vector to form a new vector. For slicing, we used colon(:) operator. Range indexes are very helpful for the situation involving a large operator. Let see an example to understand how slicing is done with the help of the colon operator to form a new vector.

```
q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
b<-q[2:5]
b
```

7) Out-of-order Indexes

In R, the index vector can be out-of-order. Below is an example in which a vector slice with the order of first and second values reversed.

```
q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
b<-q[2:5]
q[c(2,1,3,4,5,6)]
```

8). Named vectors members

We first create our vector of characters as:

```
z=c("TensorFlow","PyTorch")
z
```

Once our vector of characters is created, we name the first vector member as "Start" and the second member as "End" as:

```
names(z)=c("Start","End")
z
```

We retrieve the first member by its name as follows:

z["Start"]

We can reverse the order with the help of the character string index vector.

z[c("Second","First")]

R Lists

Lists are the objects of R which contain elements of different types such as number, vectors, string and another list inside it. It can also contain a function or a matrix as its elements. A list is a data structure which has components of mixed data types. We can say, a list is a generic vector which contains other objects.

Example:

```
vec <- c(3,4,5,6)
char_vec<-c("shubham","nishka","gunjan","sumit")
logic_vec<-c(TRUE,FALSE,FALSE,TRUE)
out_list<-list(vec,char_vec,logic_vec)
out_list
```

Lists creation

The process of creating a list is the same as a vector. In R, the vector is created with the help of c() function. Like c() function, there is another function, i.e., list() which is used to create a list in R. A list avoids the drawback of the vector which is data type. We can add the elements in the list of different data types.

Example 1: Creating list with same data type

```
list_1<-list(1,2,3)
list_2<-list("Shubham","Arpita","Vaishali")
```

```
list_3<-list(c(1,2,3))
list_4<-list(TRUE,FALSE,TRUE)
list_1
list_2
list_3
list_4
```

Example 2: Creating the list with different data type:

```
list_data<-list("Shubham","Arpita",c(1,2,3,4,5),TRUE,FALSE,22.5,12L)
print(list_data)
```

Giving a name to list elements

R provides a very easy way for accessing elements, i.e., by giving the name to each element of a list. By assigning names to the elements, we can access the element easily. 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:

Creating a list containing a vector, a matrix and a list.

```
list_data <- list(c("Shubham","Nishka","Gunjan"), matrix(c(40,80,60,70,90,80), nrow = 2),
list("BCA","MCA","B.tech"))
```

```
# Giving names to the elements in the list.
	names(list_data) <- c("Students", "Marks", "Course")
# Show the list.
```

```
        print(list_data)

$Course

$Course[[1]]

$Course[[2]]

$Course[[3]]
```

Accessing List Elements

R provides two ways through which we can access the elements of a list.

- First one is the indexing method performed in the same way as a vector
- In the second one, we can access the elements of a list with the help of names.
- It will be possible only with the named list.; we cannot access the elements of a list using names if the list is normal.

Example: Accessing elements using index

```
# Creating a list containing a vector, a matrix and a list.

        list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow =

        2),

        list("BCA","MCA","B.tech"))

# Accessing the first element of the list.

        print(list_data[1])


# Accessing the third element. The third element is also a list, so all its elements will be printed.

        print(list_data[3])
```

Example:

```
# Creating a list containing a vector, a matrix and a list.
        list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow =
        2),list("BCA","MCA","B.tech"))
# Giving names to the elements in the list.
        names(list_data) <- c("Student", "Marks", "Course")
        # Accessing the first element of the list.
print(list_data["Student"])
print(list_data$Marks)
print(list_data)
```

Example:

```
# Creating a list containing a vector, a matrix and a list.
        list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow =
        2),
        list("BCA","MCA","B.tech"))

# Giving names to the elements in the list.
        names(list_data) <- c("Student", "Marks", "Course")

# Adding an element at the end of the list.
        list_data[4] <- "Moradabad"
        print(list_data[4])

# Removing the last element.
        list_data[4] <- NULL
```

```
# Printing the 4th Element.
        print(list_data[4])
# Updating the 3rd Element.
        list_data[3] <- "Masters of computer applications"
        print(list_data[3])
```

Converting list to vector

- we cannot perform all the arithmetic operations on list elements.
- To remove this, drawback R provides **unlist() function**.
- This function converts the list into vectors.
- The unlist() function takes the list as a parameter and changes it into a vector.

Example: Creating lists

```
        list1 <- list(10:20)
        print(list1)
        list2 <-list(5:14)
        print(list2)

# Converting the lists to vectors.
        v1 <- unlist(list1)
        v2 <- unlist(list2)
        print(v1)
        print(v2)
```

adding the vectors

result <- v1+v2

print(result)

Merging Lists

- Merging is done with the help of the list() function also.
- To merge the lists, we have to pass all the lists into the list function as a parameter, and it returns a list which contains all the elements which are present in the lists.

Example: Creating two lists.

- list1 <- list(2,4,6,8,10)
- list2 <- list(1,3,5,7,9)

# Merging the two lists.

- merged.list <- list(list1, list2)

# Printing the merged list.

- print(merged.list)

R Arrays

- In R, arrays are the data objects which allow us to store data in more than two dimensions.
- array is created with the help of the **array()** function.
- **array() function** takes a vector as an input and to create an array it uses vectors values in the **dim** parameter.

Eg. - if we create an array of dimension (3, 3, 4) then it will create 4 rectangular matrices of 3 rows and 3 columns.

R Array Syntax

array_name <- array(data, dim= (row_size, column_size, matrices, dim_names))

Data

- The data is the first argument in the array() function.
- It is an input vector which is given to the array.

matrices

In R, the array consists of multi-dimensional matrices.

row_size

This parameter defines the number of row elements which an array can store.

column_size

This parameter defines the number of column elements which an array can store.

dim_names

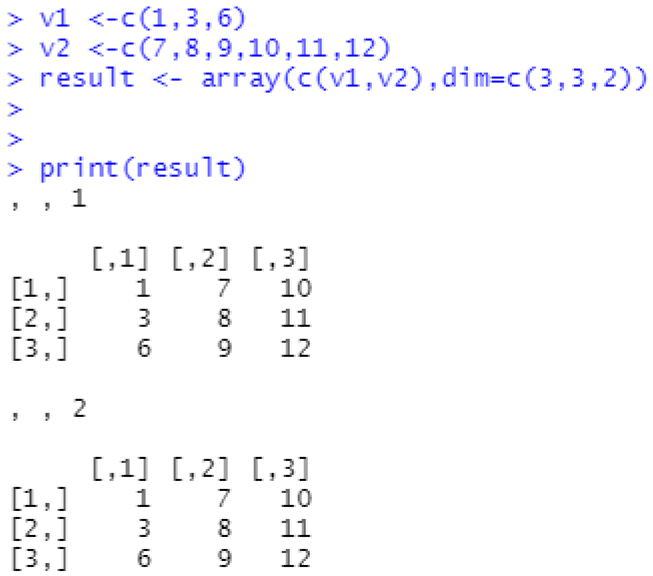
This parameter is used to change the default names of rows and columns.

How to create an array?

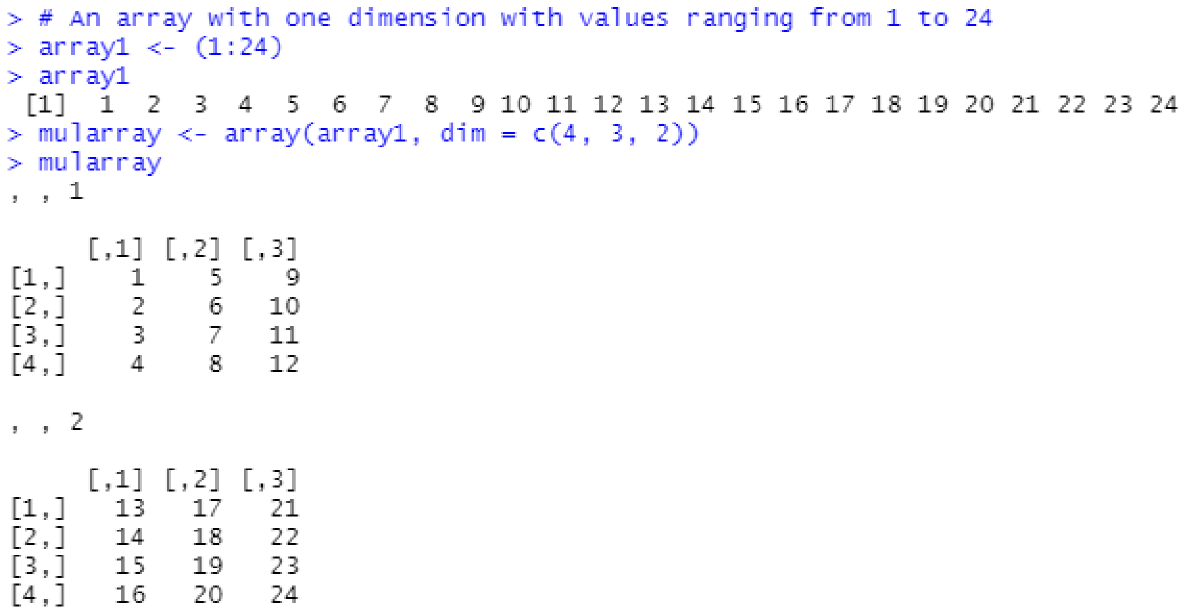
- Create an array using the vector and array() function. In array, data is stored in the form of the matrix.
- There are only two steps to create a matrix which are as follows

1. In the first step, we will create two vectors of different lengths.
2. Once our vectors are created, we take these vectors as inputs to the array.

Example: Creating two vectors of different lengths

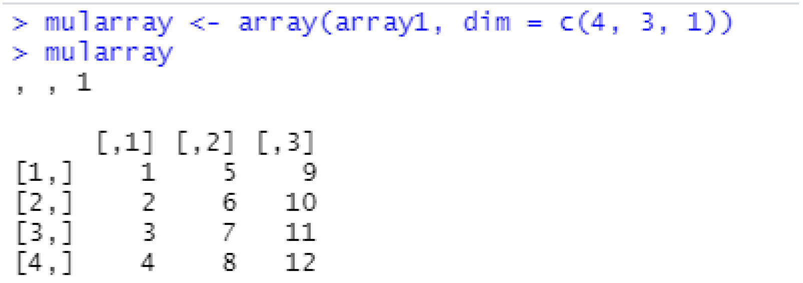


```
> v1 <-c(1,3,6)
> v2 <-c(7,8,9,10,11,12)
> result <- array(c(v1,v2),dim=c(3,3,2))
>
>
> print(result)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    1    7   10
[2,]    3    8   11
[3,]    6    9   12
```

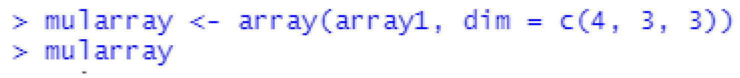


```
> # An array with one dimension with values ranging from 1 to 24
> array1 <- (1:24)
> array1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
> mularray <- array(array1, dim = c(4, 3, 2))
> mularray
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

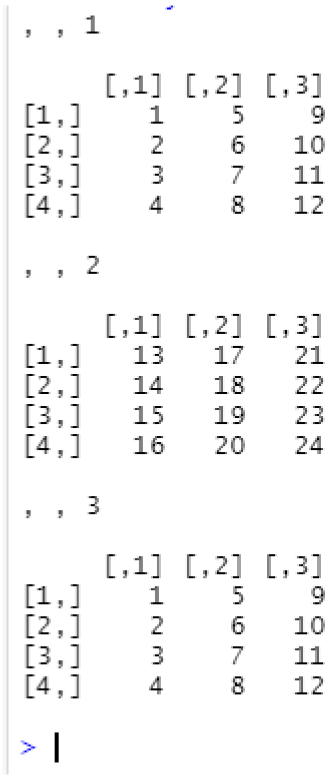
Example:



```
> mularray <- array(array1, dim = c(4, 3, 1))
> mularray
, , 1

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



```
> mularray <- array(array1, dim = c(4, 3, 3))
> mularray
```

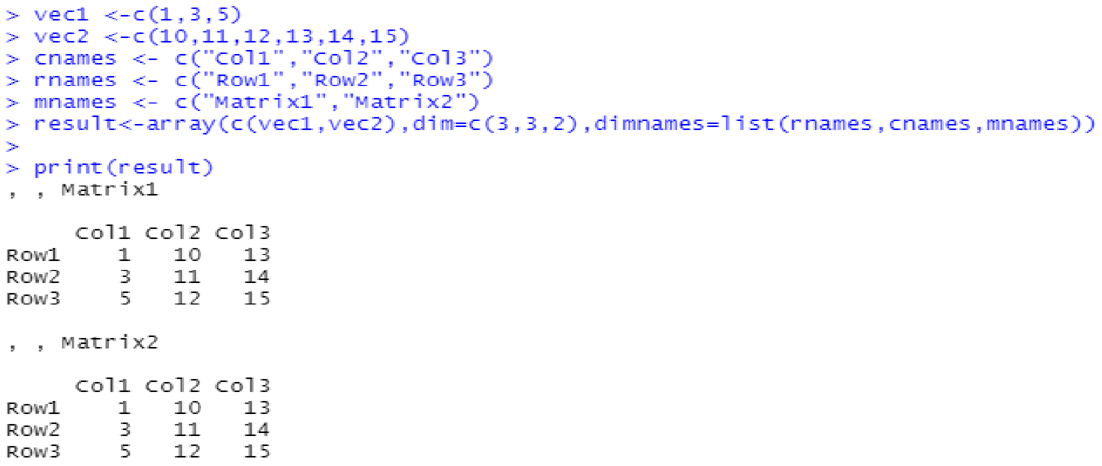


```
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24

, , 3

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

> 
```

Naming rows and columns

- In R, we can give the names to the rows, columns, and matrices of the array.
- This is done with the help of the **dim** name parameter of the **array()** function.

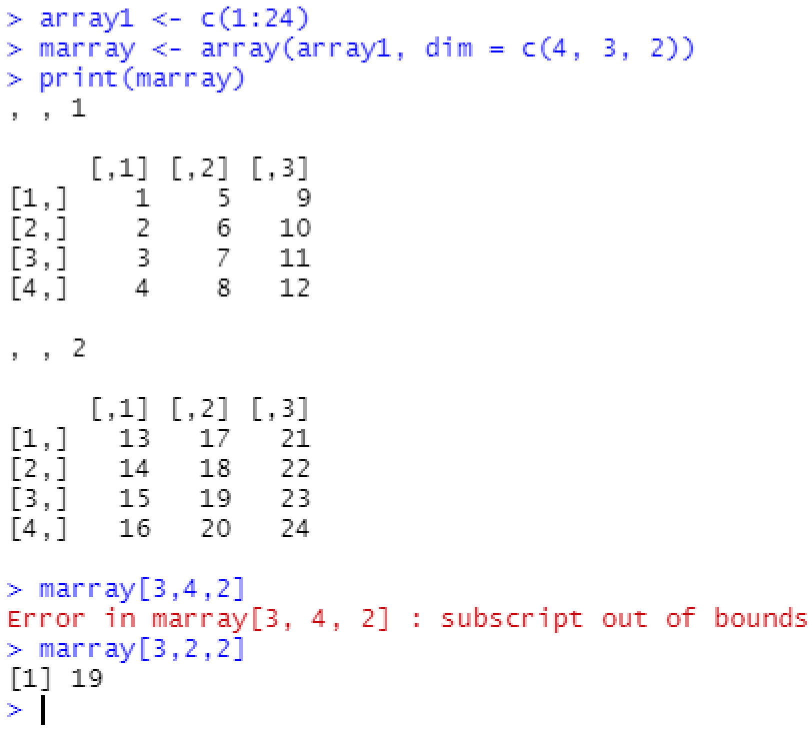
Example: Creating two vectors



```
> vec1 <-c(1,3,5)
> vec2 <-c(10,11,12,13,14,15)
> cnames <- c("Col1","Col2","Col3")
> rnames <- c("Row1","Row2","Row3")
> mnames <- c("Matrix1","Matrix2")
> result<-array(c(vec1,vec2),dim=c(3,3,2),dimnames=list(rnames,cnames,mnames))
>
> print(result)
, , Matrix1

     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15

, , Matrix2

     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15
```

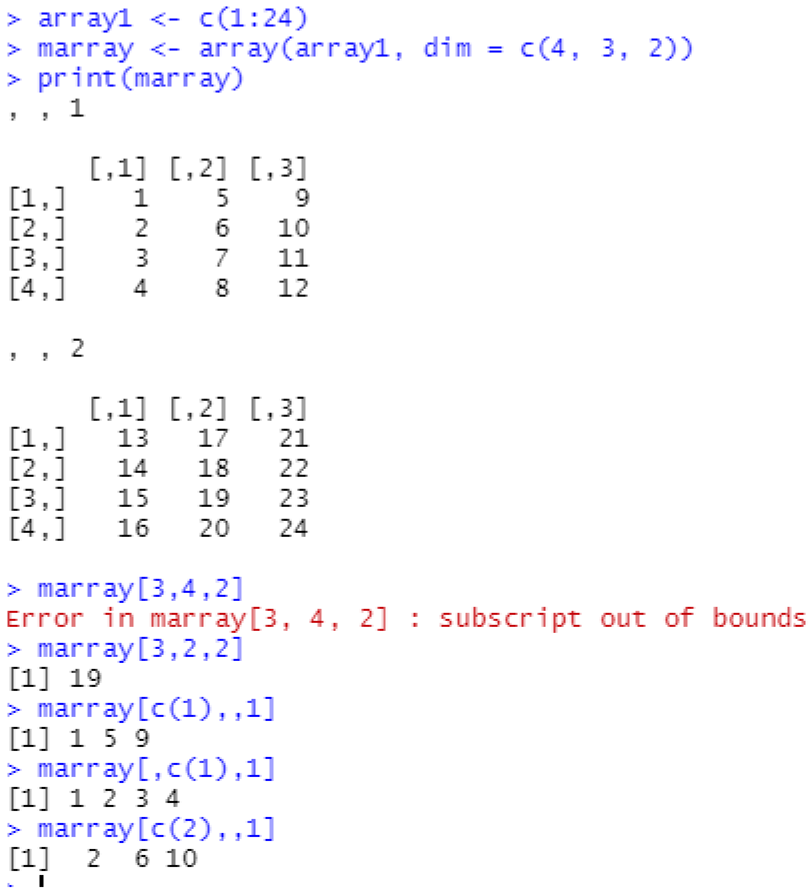
Accessing array elements

- we can access the elements of the array with the help of the indexing method.



```
> array1 <- c(1:24)
> marray <- array(array1, dim = c(4, 3, 2))
> print(marray)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24

> marray[3,4,2]
Error in marray[3, 4, 2] : subscript out of bounds
> marray[3,2,2]
[1] 19
> 
```

Example:



```
> array1 <- c(1:24)
> marray <- array(array1, dim = c(4, 3, 2))
> print(marray)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24

> marray[3,4,2]
Error in marray[3, 4, 2] : subscript out of bounds
> marray[3,2,2]
[1] 19
> marray[c(1),,1]
[1] 1 5 9
> marray[,c(1),1]
[1] 1 2 3 4
> marray[c(2),,1]
[1]  2  6 10
```

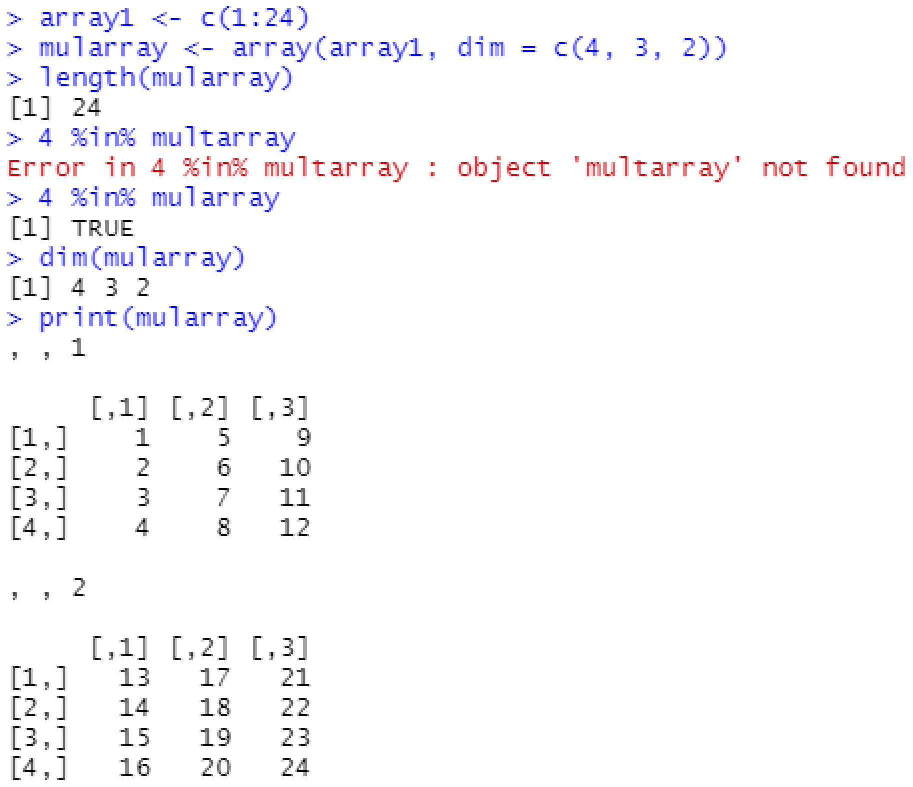
Note:

- we can access the row or column of the matrics.
- A comma (,) before c() means that we want to access the column.
- A comma (,) after c() means that we want to access the row.

Operations over the array

1. To check if an item is present in an array, use the **%in%** operator.
2. **dim()** function to find the amount of rows and columns in an array.
3. **length()** function to find the dimension .

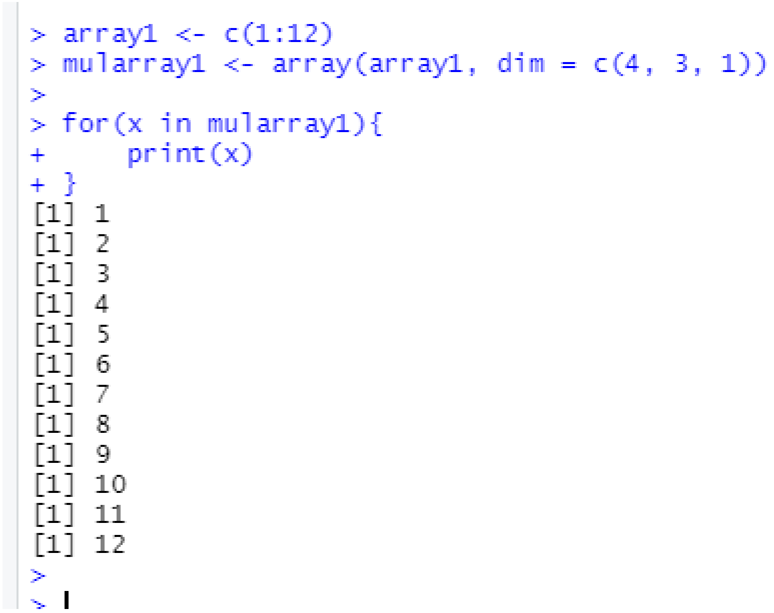
Example:



```
> array1 <- c(1:24)
> mularray <- array(array1, dim = c(4, 3, 2))
> length(mularray)
[1] 24
> 4 %in% multarray
Error in 4 %in% multarray : object 'multarray' not found
> 4 %in% mularray
[1] TRUE
> dim(mularray)
[1] 4 3 2
> print(mularray)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

Loop through an array

Example:



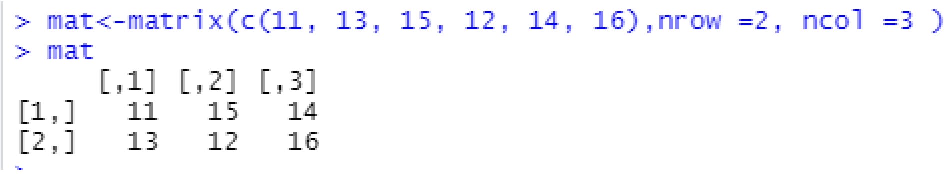
```
> array1 <- c(1:12)
> mularray1 <- array(array1, dim = c(4, 3, 1))
>
> for(x in mularray1){
+     print(x)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
[1] 11
[1] 12
>
> |
```

R Matrices

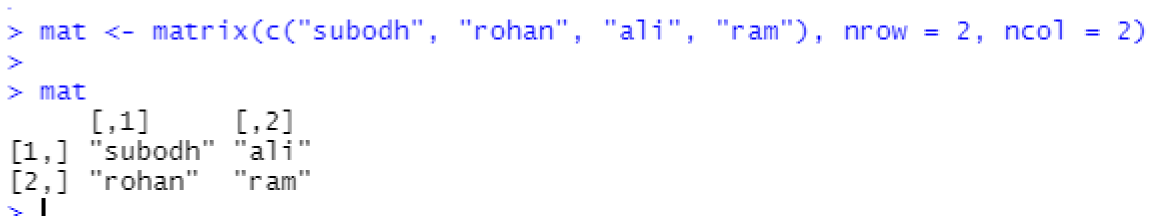
- A matrix is a two dimensional data set with columns and rows.

- A matrix can be created with the **matrix() function.** Specify the **nrow** and **ncol** parameters to get the amount of rows and columns.
- c() function is used to concatenate items together.

Example:



```
> mat<-matrix(c(11, 13, 15, 12, 14, 16),nrow =2, ncol =3 )
> mat
     [,1] [,2] [,3]
[1,]   11   15   14
[2,]   13   12   16
```

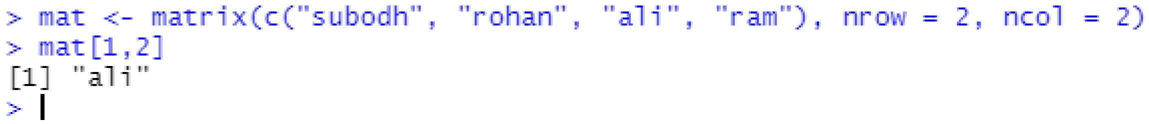


```
> mat <- matrix(c("subodh", "rohan", "ali", "ram"), nrow = 2, ncol = 2)
>
> mat
     [,1]     [,2] 
[1,] "subodh" "ali"
[2,] "rohan"  "ram"
> 
```

Access Matrix Items

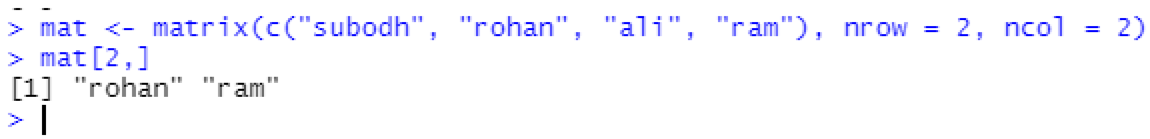
You can access the items by using [ ] brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

Example:



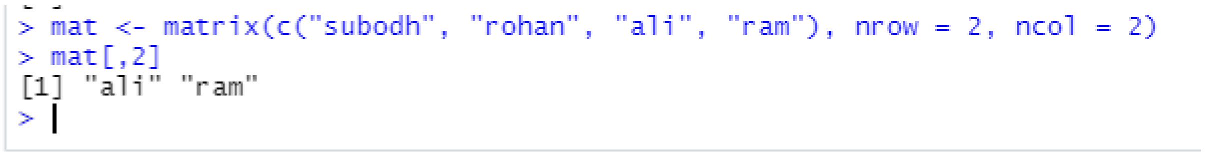
```
> mat <- matrix(c("subodh", "rohan", "ali", "ram"), nrow = 2, ncol = 2)
> mat[1,2]
[1] "ali"
> 
```

Example:



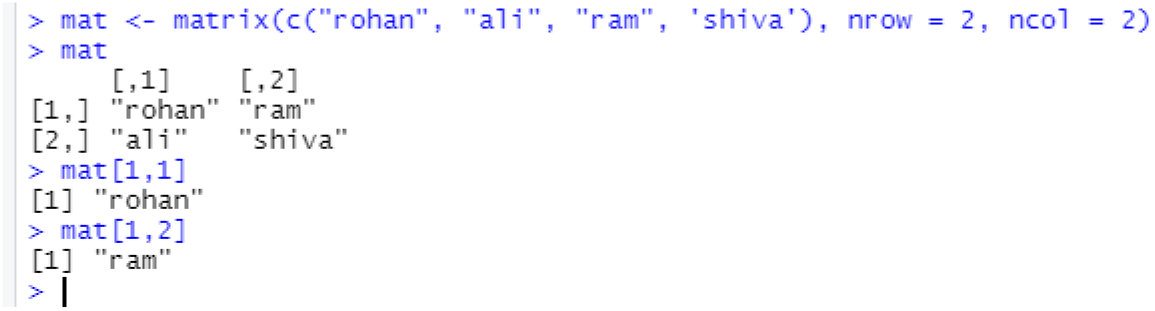
```
> mat <- matrix(c("subodh", "rohan", "ali", "ram"), nrow = 2, ncol = 2)
> mat[2,]
[1] "rohan" "ram"
> 
```

Example:



```
> mat <- matrix(c("subodh", "rohan", "ali", "ram"), nrow = 2, ncol = 2)
> mat[,2]
[1] "ali" "ram"
> 
```

Example:

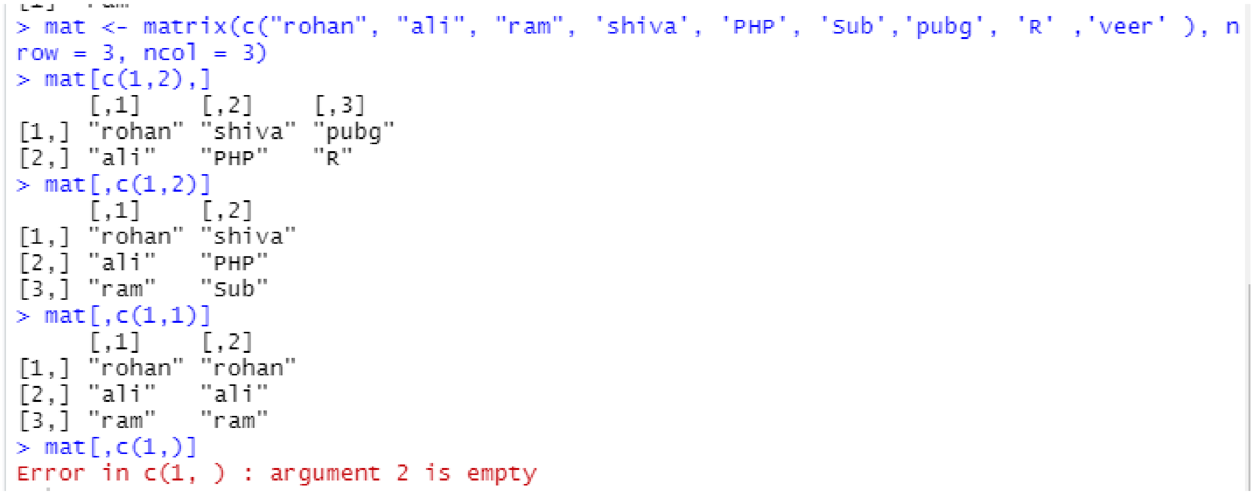


```
> mat <- matrix(c("rohan", "ali", "ram", 'shiva'), nrow = 2, ncol = 2)
> mat
      [,1]    [,2]
[1,] "rohan" "ram"
[2,] "ali"   "shiva"
> mat[1,1]
[1] "rohan"
> mat[1,2]
[1] "ram"
> 
```

Access More than One Row and more than one column

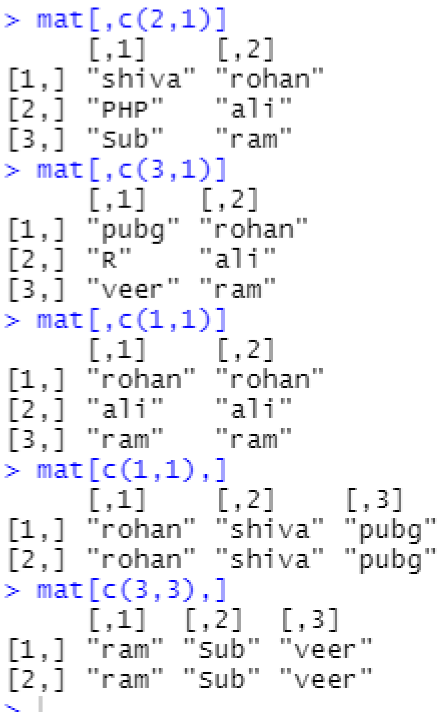
More than one row and column can be accessed by the c() function.

Example:



```
> mat <- matrix(c("rohan", "ali", "ram", 'shiva', 'PHP', 'Sub','pubg', 'R' ,'veer' ), n
row = 3, ncol = 3)
> mat[c(1,2),]
     [,1]    [,2]    [,3]
[1,] "rohan" "shiva" "pubg"
[2,] "ali"   "PHP"   "R"
> mat[,c(1,2)]
     [,1]    [,2]
[1,] "rohan" "shiva"
[2,] "ali"   "PHP"
[3,] "ram"   "Sub"
> mat[,c(1,1)]
     [,1]    [,2]
[1,] "rohan" "rohan"
[2,] "ali"   "ali"
[3,] "ram"   "ram"
> mat[,c(1,)]
Error in c(1, ) : argument 2 is empty
```

Example:

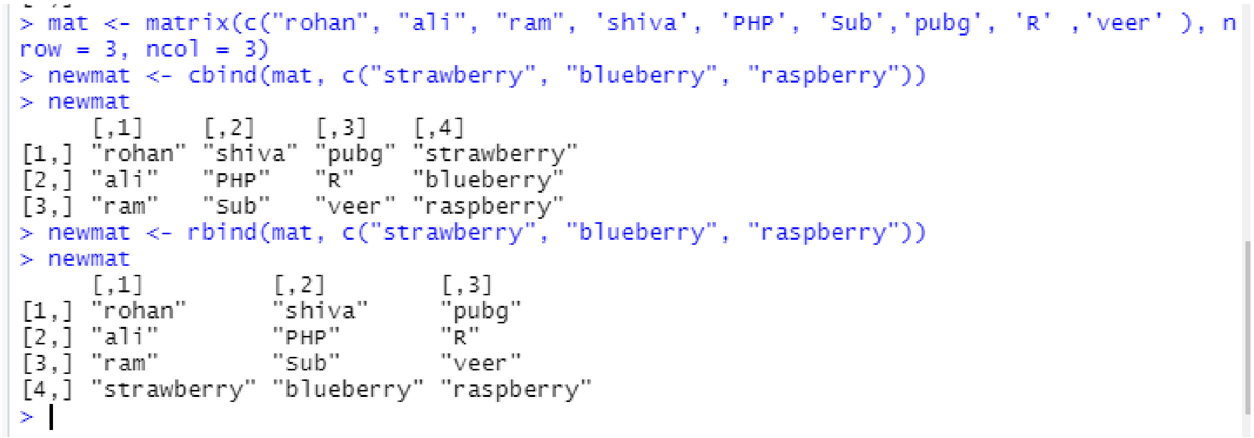


```
> mat[,c(2,1)]
     [,1]    [,2]   
[1,] "shiva" "rohan"
[2,] "PHP"   "ali"  
[3,] "Sub"   "ram"  
> mat[,c(3,1)]
     [,1]   [,2]   
[1,] "pubg" "rohan"
[2,] "R"    "ali"  
[3,] "veer" "ram"  
> mat[,c(1,1)]
     [,1]    [,2]   
[1,] "rohan" "rohan"
[2,] "ali"   "ali"  
[3,] "ram"   "ram"  
> mat[c(1,1),]
     [,1]    [,2]    [,3]  
[1,] "rohan" "shiva" "pubg"
[2,] "rohan" "shiva" "pubg"
> mat[c(3,3),]
     [,1]  [,2]  [,3]  
[1,] "ram" "Sub" "veer"
[2,] "ram" "Sub" "veer"
>
```

Add Rows and Columns

Use the **cbind() function** to add additional columns and **rbind() function** to add additional rows in a Matrix.

Example:

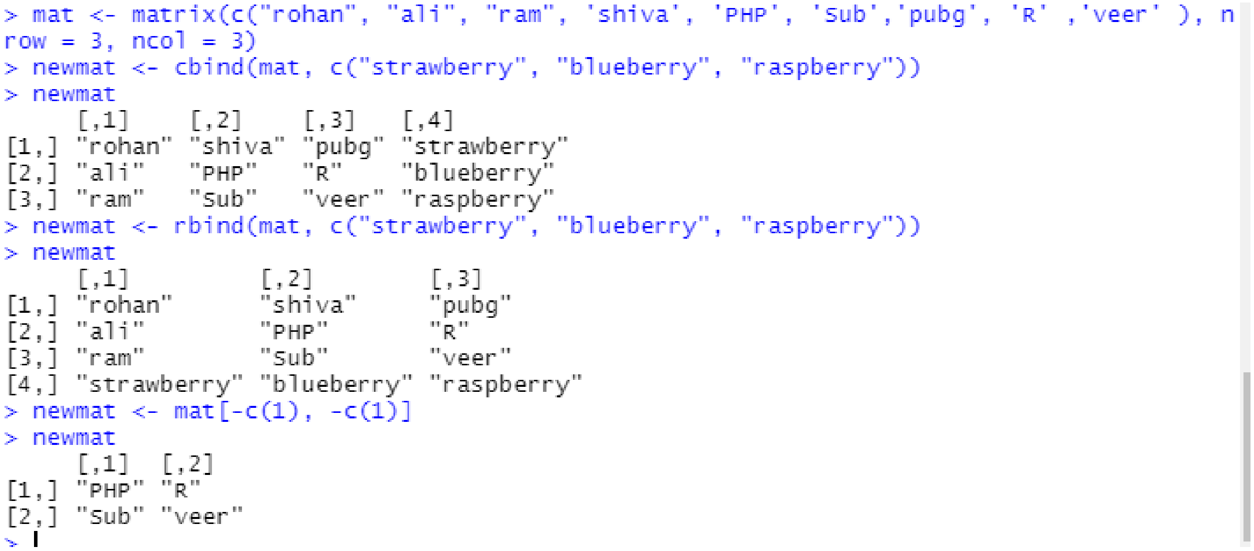


```
> mat <- matrix(c("rohan", "ali", "ram", 'shiva', 'PHP', 'Sub','pubg', 'R' ,'veer' ), n
row = 3, ncol = 3)
> newmat <- cbind(mat, c("strawberry", "blueberry", "raspberry"))
> newmat
     [,1]    [,2]    [,3]   [,4]        
[1,] "rohan" "shiva" "pubg" "strawberry"
[2,] "ali"   "PHP"   "R"    "blueberry" 
[3,] "ram"   "Sub"   "veer" "raspberry" 
> newmat <- rbind(mat, c("strawberry", "blueberry", "raspberry"))
> newmat
     [,1]         [,2]        [,3]       
[1,] "rohan"      "shiva"     "pubg"     
[2,] "ali"        "PHP"       "R"        
[3,] "ram"        "Sub"       "veer"     
[4,] "strawberry" "blueberry" "raspberry"
> 
```

Remove Rows and Columns

Use the c() function to remove rows and columns in a Matrix:

Example:

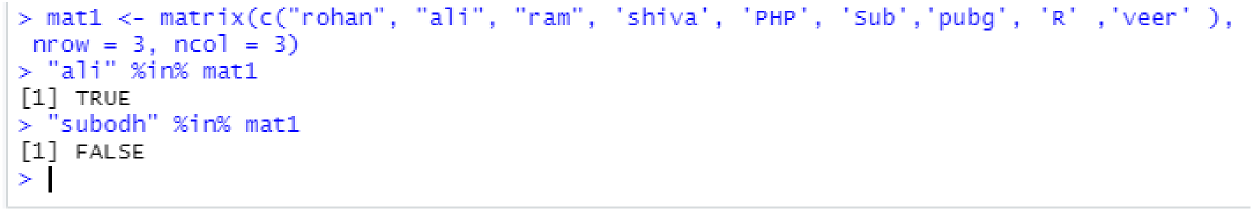


```
> mat <- matrix(c("rohan", "ali", "ram", 'shiva', 'PHP', 'Sub','pubg', 'R' ,'veer' ), n
row = 3, ncol = 3)
> newmat <- cbind(mat, c("strawberry", "blueberry", "raspberry"))
> newmat
     [,1]    [,2]    [,3]   [,4]        
[1,] "rohan" "shiva" "pubg" "strawberry"
[2,] "ali"   "PHP"   "R"    "blueberry" 
[3,] "ram"   "Sub"   "veer" "raspberry" 
> newmat <- rbind(mat, c("strawberry", "blueberry", "raspberry"))
> newmat
     [,1]         [,2]        [,3]       
[1,] "rohan"      "shiva"     "pubg"     
[2,] "ali"        "PHP"       "R"        
[3,] "ram"        "Sub"       "veer"     
[4,] "strawberry" "blueberry" "raspberry"
> newmat <- mat[-c(1), -c(1)]
> newmat
     [,1]  [,2]  
[1,] "PHP" "R"   
[2,] "Sub" "veer"
> 
```

Operations over the matrix

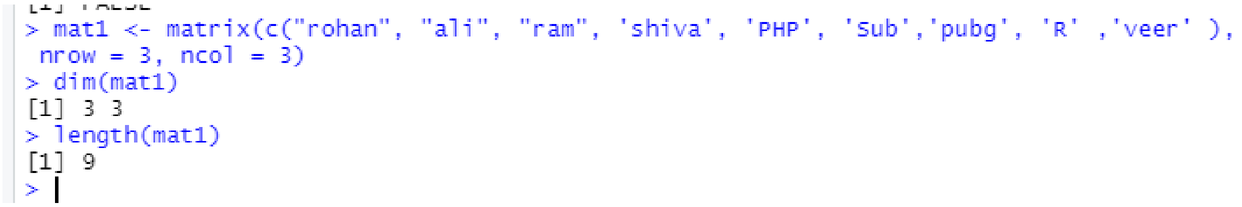
4. To check if an item is present in a matrix, use the **%in%** operator.
5. **dim()** function to find the amount of rows and columns in a Matrix:
6. **length()** function to find the dimension of a Matrix.

Example:



```
> mat1 <- matrix(c("rohan", "ali", "ram", 'shiva', 'PHP', 'Sub','pubg', 'R' ,'veer' ),
 nrow = 3, ncol = 3)
> "ali" %in% mat1
[1] TRUE
> "subodh" %in% mat1
[1] FALSE
> 
```

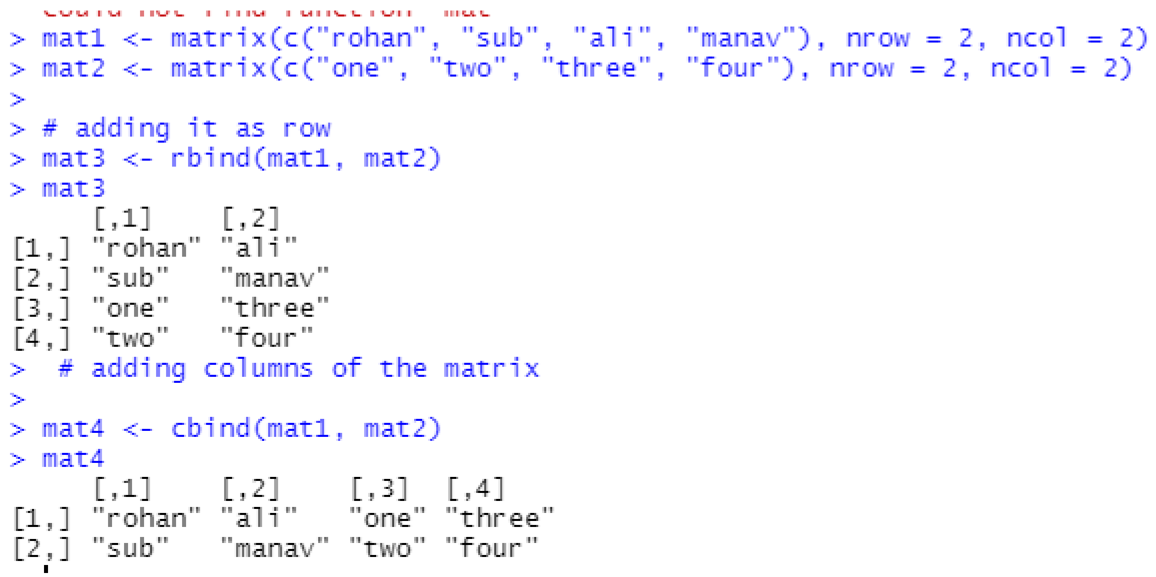
Example:



```
> mat1 <- matrix(c("rohan", "ali", "ram", 'shiva', 'PHP', 'Sub','pubg', 'R' ,'veer' ),
 nrow = 3, ncol = 3)
> dim(mat1)
[1] 3 3
> length(mat1)
[1] 9
> 
```

Combine two Matrices

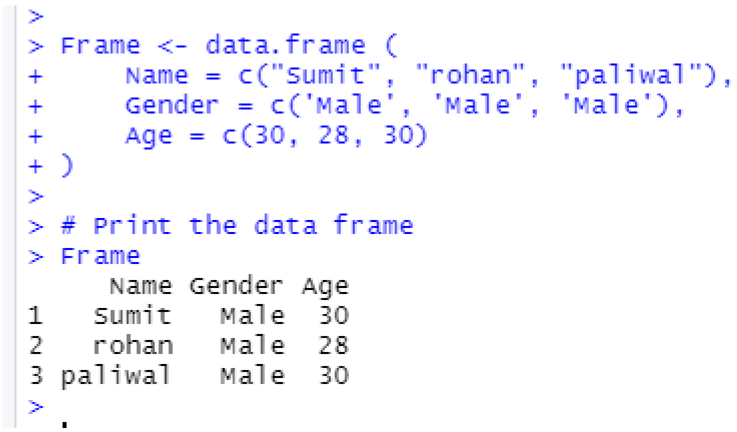
- **rbind()** or **cbind()** function to combine two or more matrices together.



```
could not find function "mat"
> mat1 <- matrix(c("rohan", "sub", "ali", "manav"), nrow = 2, ncol = 2)
> mat2 <- matrix(c("one", "two", "three", "four"), nrow = 2, ncol = 2)
>
> # adding it as row
> mat3 <- rbind(mat1, mat2)
> mat3
     [,1]    [,2]
[1,] "rohan" "ali"
[2,] "sub"   "manav"
[3,] "one"   "three"
[4,] "two"   "four"
>  # adding columns of the matrix
>
> mat4 <- cbind(mat1, mat2)
> mat4
     [,1]    [,2]    [,3]  [,4]
[1,] "rohan" "ali"   "one" "three"
[2,] "sub"   "manav" "two" "four"
```

Data Frames

- Data Frames are data displayed in a format as a table.
- Data Frames can have different types of data inside it.
- However, each column should have the same type of data.
- **data.frame()** function to create a data frame.
- While the first column can be **character,** the second and third can be **numeric** or **logical.**



```
>
> Frame <- data.frame (
+     Name = c("Sumit", "rohan", "paliwal"),
+     Gender = c('Male', 'Male', 'Male'),
+     Age = c(30, 28, 30)
+ )
>
> # Print the data frame
> Frame
     Name Gender Age
1   Sumit   Male  30
2   rohan   Male  28
3 paliwal   Male  30
>
```

Summarize the Data

- **summary()** function used to summarize the data from a Data Frame.

The **summary()** function returns six statistical numbers for each variable:

- Min
- First quantile (percentile)
- Median

- Mean
- Third quantile (percentile)
- Max

Mean, Median, and Mode

In statistics, there are often three values that interests us:

- Mean - The average value
- Median - The middle value
- Mode - The most common value

Percentiles

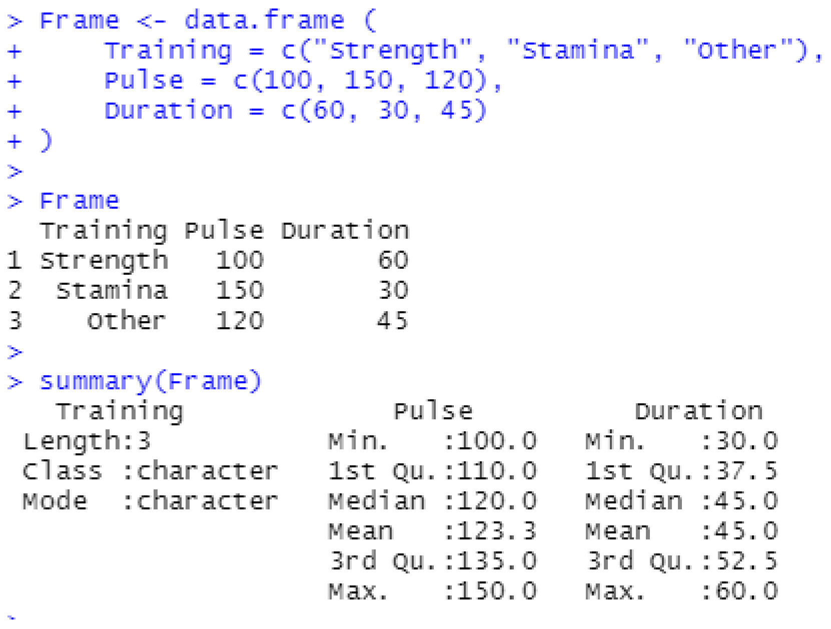
- Percentiles are used in statistics to give you a number that describes the value that a given percent of the values are lower than

Quartiles

Quartiles are data divided into four parts, when sorted in an ascending order:

1. The value of the first quartile cuts off the first 25% of the data
2. The value of the second quartile cuts off the first 50% of the data
3. The value of the third quartile cuts off the first 75% of the data
4. The value of the fourth quartile cuts off the 100% of the data

Note: Use the **quantile()** function to get the quartile value.

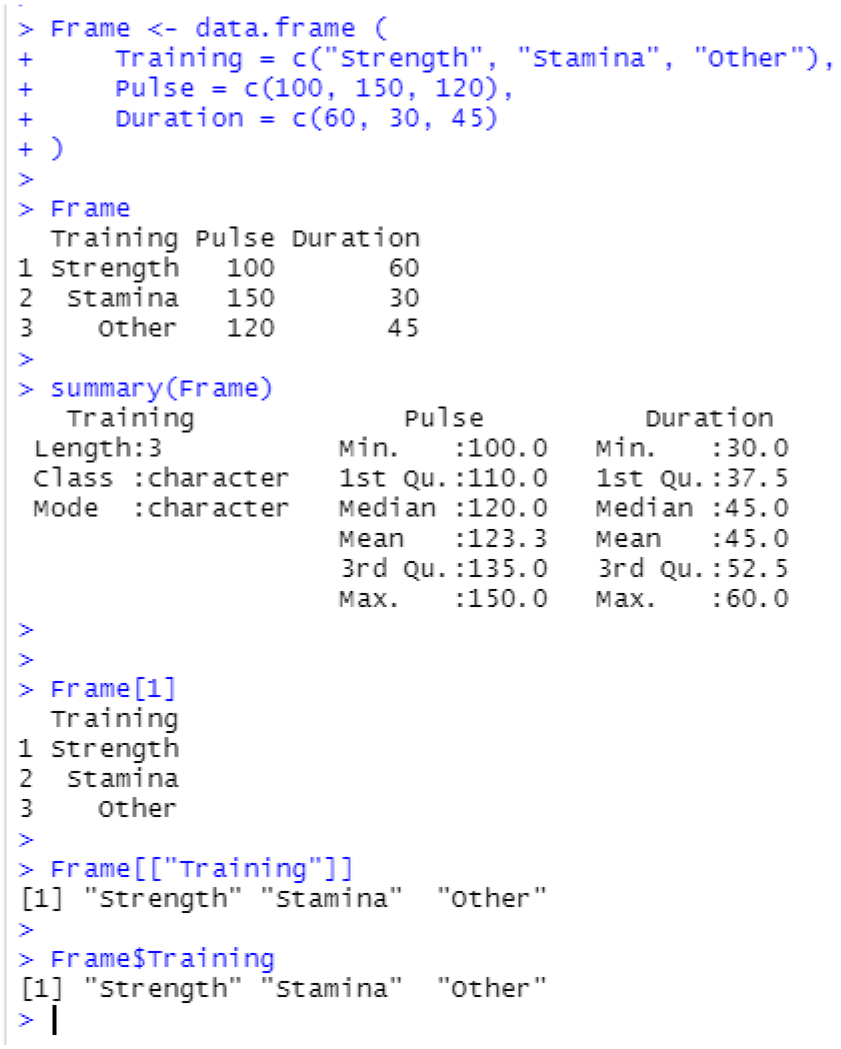


```
> Frame <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
>
> summary(Frame)
   Training             Pulse          Duration   
 Length:3           Min.   :100.0   Min.   :30.0  
 Class :character   1st Qu.:110.0   1st Qu.:37.5  
 Mode  :character   Median :120.0   Median :45.0  
                    Mean   :123.3   Mean   :45.0  
                    3rd Qu.:135.0   3rd Qu.:52.5  
                    Max.   :150.0   Max.   :60.0  
```

Access Items

- We can use **single brackets [ ]**, double brackets **[[ ]]** or **$** to access columns from a data frame.

Example:

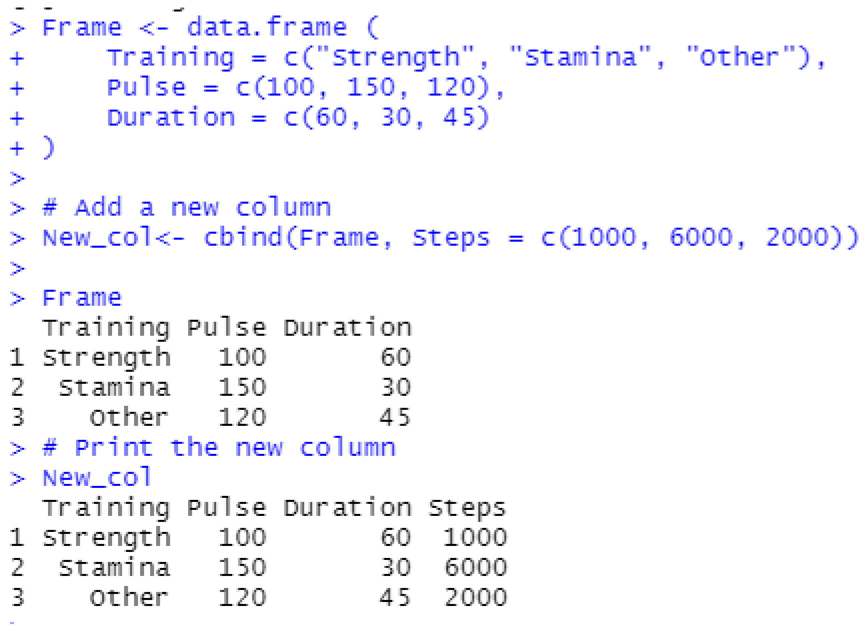


```
> Frame <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
>
> summary(Frame)
   Training             Pulse          Duration  
 Length:3          Min.   :100.0   Min.   :30.0  
 Class :character  1st Qu.:110.0   1st Qu.:37.5  
 Mode  :character  Median :120.0   Median :45.0  
                   Mean   :123.3   Mean   :45.0  
                   3rd Qu.:135.0   3rd Qu.:52.5  
                   Max.   :150.0   Max.   :60.0  
>
>
> Frame[1]
  Training
1 Strength
2  Stamina
3    Other
>
> Frame[["Training"]]
[1] "Strength" "Stamina"  "Other"
>
> Frame$Training
[1] "Strength" "Stamina"  "Other"
> 
```

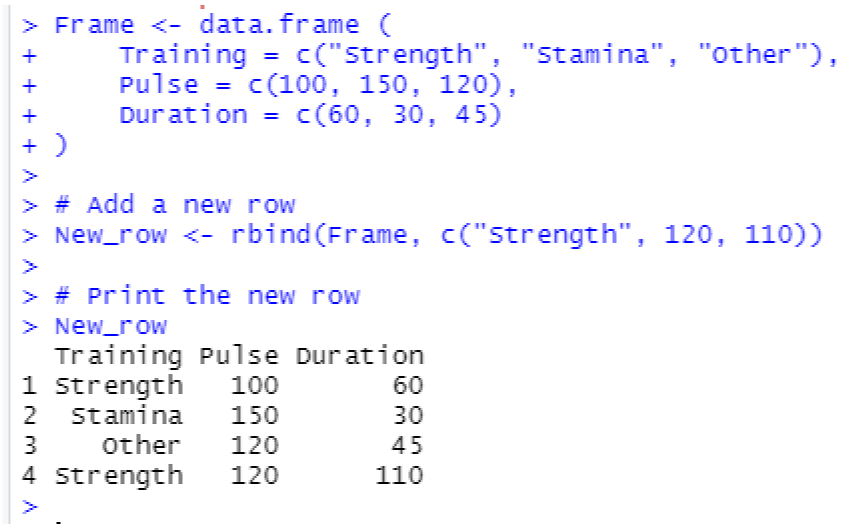
Adding Rows and columns

- Use the **rbind()** function to add new rows in a Data Frame.
- Use the **cbind()** function to add new columns in a Data Frame.

Example:



```
> Frame <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> # Add a new column
> New_col<- cbind(Frame, Steps = c(1000, 6000, 2000))
>
> Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
> # Print the new column
> New_col
  Training Pulse Duration Steps
1 Strength   100       60  1000
2  Stamina   150       30  6000
3    Other   120       45  2000
```

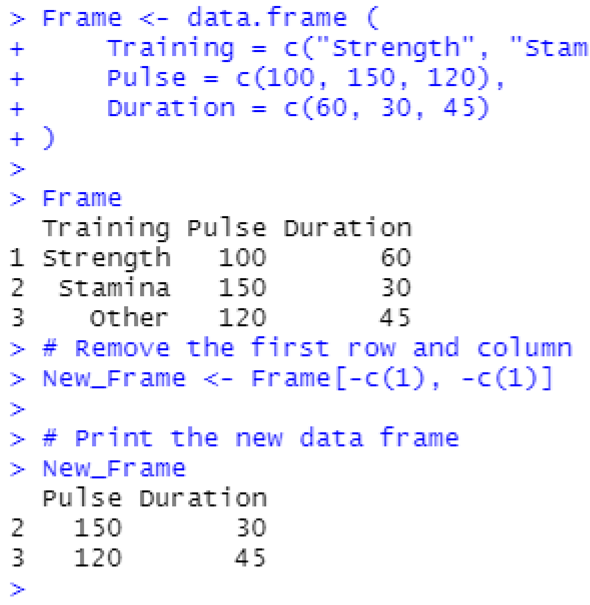


```
> Frame <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> # Add a new row
> New_row <- rbind(Frame, c("Strength", 120, 110))
>
> # Print the new row
> New_row
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
4 Strength   120      110
>
```

Removing Rows and Columns

- Use the c() function to remove rows and columns in a Data Frame.

Example:

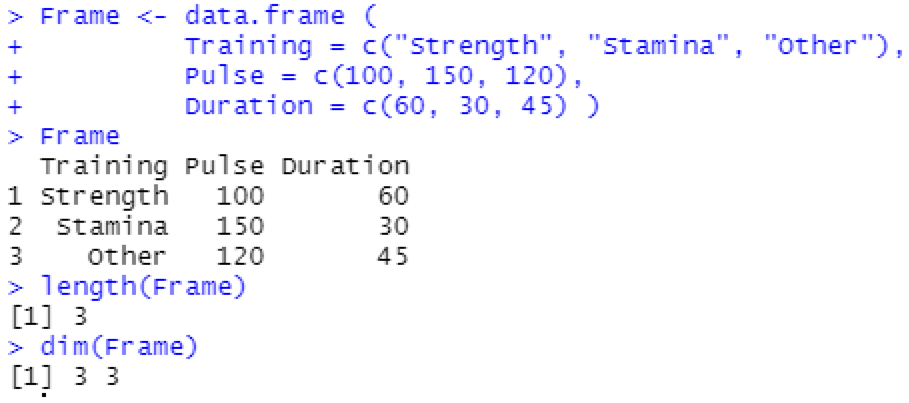


```
> Frame <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
> # Remove the first row and column
> New_Frame <- Frame[-c(1), -c(1)]
>
> # Print the new data frame
> New_Frame
  Pulse Duration
2   150       30
3   120       45
>
```

Some other operations over Data frame

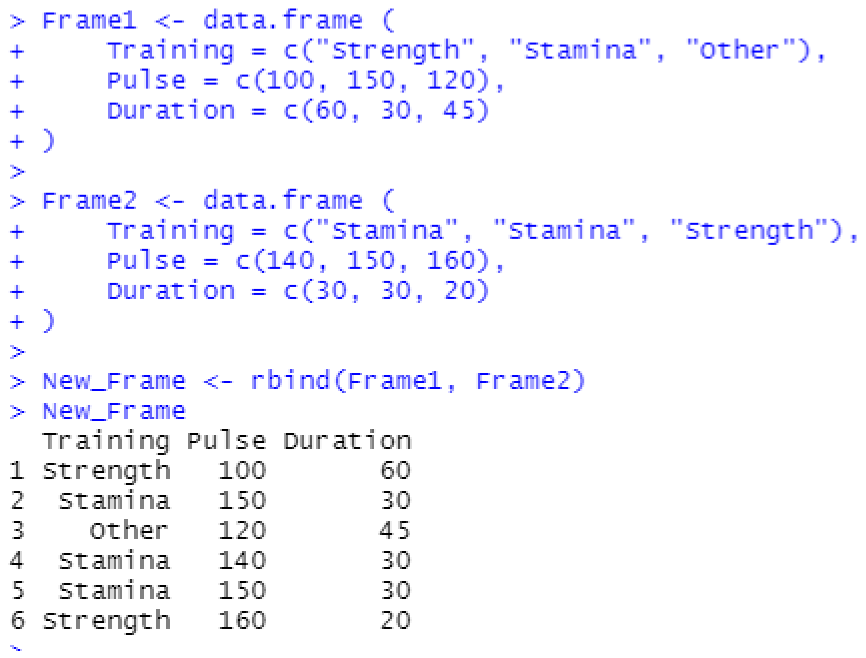
- dim() function to find the amount of rows and columns in a Data Frame.
- **length()** function to find the number of columns in a Data Frame
- **cbind()** function to combine two or more data frames in R horizontally.
- **rbind()** function to combine two or more data frames in R vertically.

Example:



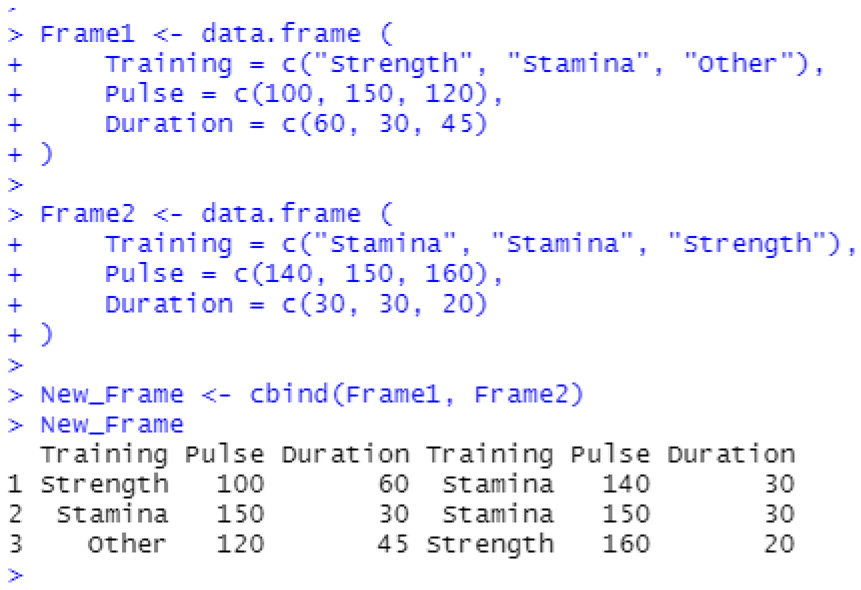
```
> Frame <- data.frame (
+          Training = c("Strength", "Stamina", "Other"),
+          Pulse = c(100, 150, 120),
+          Duration = c(60, 30, 45) )
> Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
> length(Frame)
[1] 3
> dim(Frame)
[1] 3 3
```

Example:



```
> Frame1 <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> Frame2 <- data.frame (
+     Training = c("Stamina", "Stamina", "Strength"),
+     Pulse = c(140, 150, 160),
+     Duration = c(30, 30, 20)
+ )
>
> New_Frame <- rbind(Frame1, Frame2)
> New_Frame
  Training Pulse Duration
1 Strength   100       60
2  Stamina   150       30
3    Other   120       45
4  Stamina   140       30
5  Stamina   150       30
6 Strength   160       20
>
```

Example:



```
>
> Frame1 <- data.frame (
+     Training = c("Strength", "Stamina", "Other"),
+     Pulse = c(100, 150, 120),
+     Duration = c(60, 30, 45)
+ )
>
> Frame2 <- data.frame (
+     Training = c("Stamina", "Stamina", "Strength"),
+     Pulse = c(140, 150, 160),
+     Duration = c(30, 30, 20)
+ )
>
> New_Frame <- cbind(Frame1, Frame2)
> New_Frame
  Training Pulse Duration Training Pulse Duration
1 Strength   100       60  Stamina   140       30
2  Stamina   150       30  Stamina   150       30
3    Other   120       45 Strength   160       20
>
```

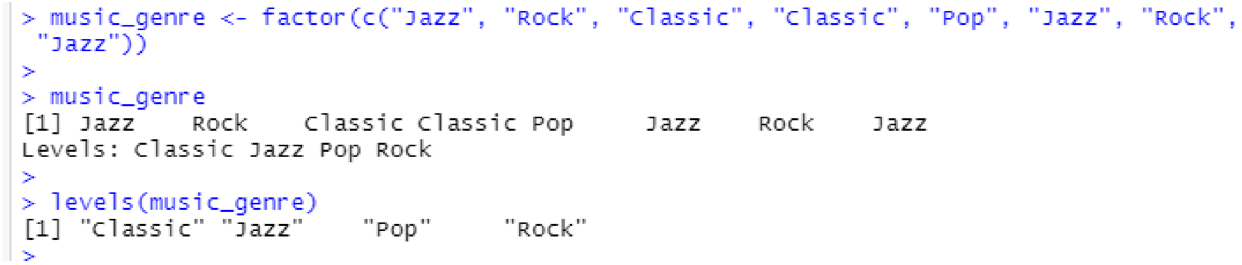
Factors

- These are the data objects which are used to categorize the data and to store it on multiple levels.
- Simply we can say that , Factors are used to categorize data.

How to create a Factor:

- use the **factor()** function and add a vector as an argument.

Example:

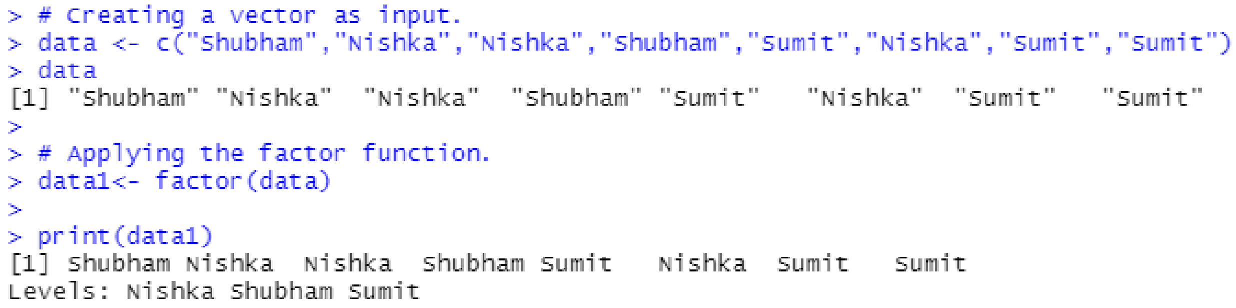


```
> music_genre <- factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock",
 "Jazz"))
>
> music_genre
[1] Jazz    Rock    Classic Classic Pop     Jazz    Rock    Jazz
Levels: Classic Jazz Pop Rock
>
> levels(music_genre)
[1] "Classic" "Jazz"    "Pop"     "Rock"
>
```

Or we can create Factor

1. In the first step, we create a vector.
2. Next step is to convert the vector into a factor.

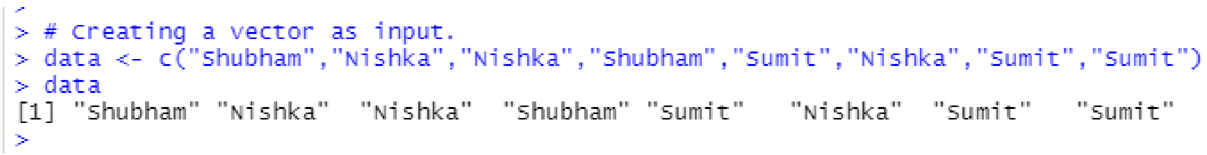
Example:



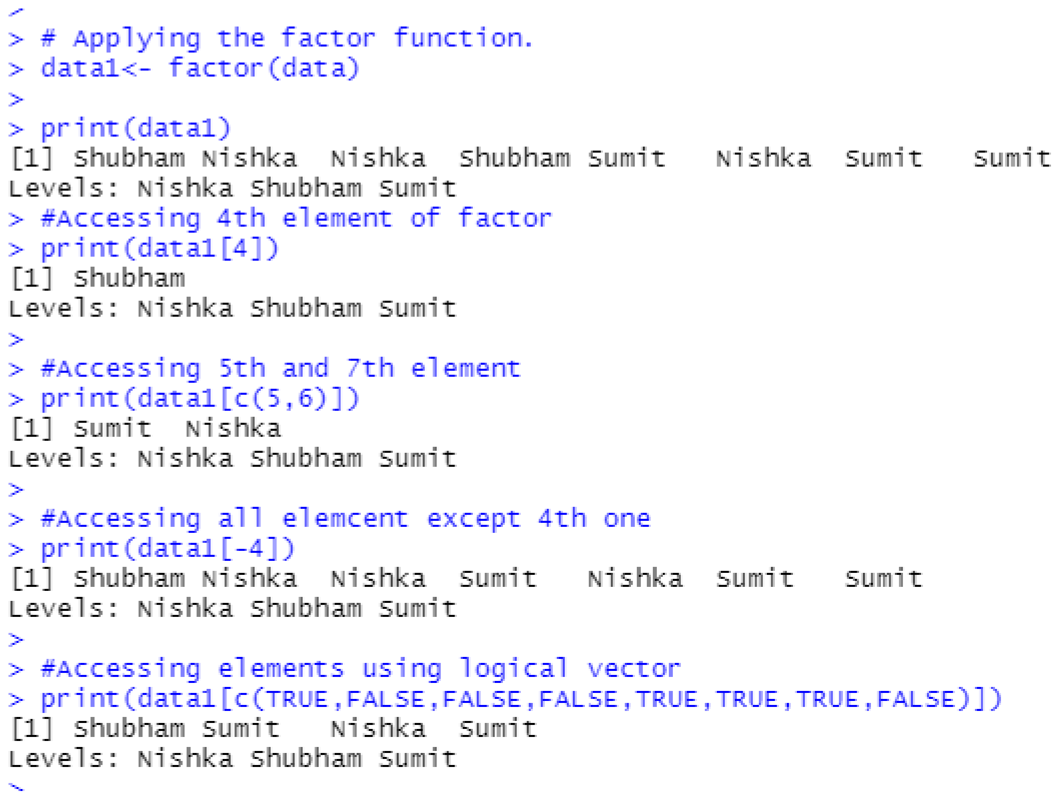
```
> # Creating a vector as input.
> data <- c("Shubham","Nishka","Nishka","Shubham","Sumit","Nishka","Sumit","Sumit")
> data
[1] "Shubham" "Nishka"  "Nishka"  "Shubham" "Sumit"   "Nishka"  "Sumit"   "Sumit"
>
> # Applying the factor function.
> data1<- factor(data)
>
> print(data1)
[1] Shubham Nishka  Nishka  Shubham Sumit   Nishka  Sumit   Sumit
Levels: Nishka Shubham Sumit
```

Accessing the elements of the factors

Example:



```
> # Creating a vector as input.
> data <- c("Shubham","Nishka","Nishka","Shubham","Sumit","Nishka","Sumit","Sumit")
> data
[1] "Shubham" "Nishka"  "Nishka"  "Shubham" "Sumit"   "Nishka"  "Sumit"   "Sumit"
>
```

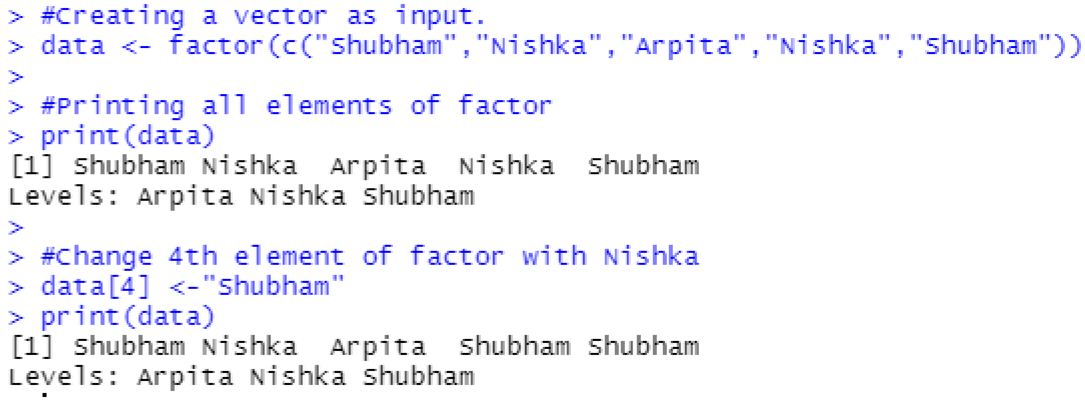


```
> # Applying the factor function.
> data1<- factor(data)
>
> print(data1)
[1] Shubham Nishka  Nishka  Shubham Sumit   Nishka  Sumit   Sumit
Levels: Nishka Shubham Sumit
> #Accessing 4th element of factor
> print(data1[4])
[1] Shubham
Levels: Nishka Shubham Sumit
>
> #Accessing 5th and 7th element
> print(data1[c(5,6)])
[1] Sumit  Nishka
Levels: Nishka Shubham Sumit
>
> #Accessing all elemcent except 4th one
> print(data1[-4])
[1] Shubham Nishka  Nishka  Sumit   Nishka  Sumit   Sumit
Levels: Nishka Shubham Sumit
>
> #Accessing elements using logical vector
> print(data1[c(TRUE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,FALSE)])
[1] Shubham Sumit   Nishka  Sumit
Levels: Nishka Shubham Sumit
```

Modification of factor

- We can modify the value of a factor by simply re-assigning it.
-

Example:

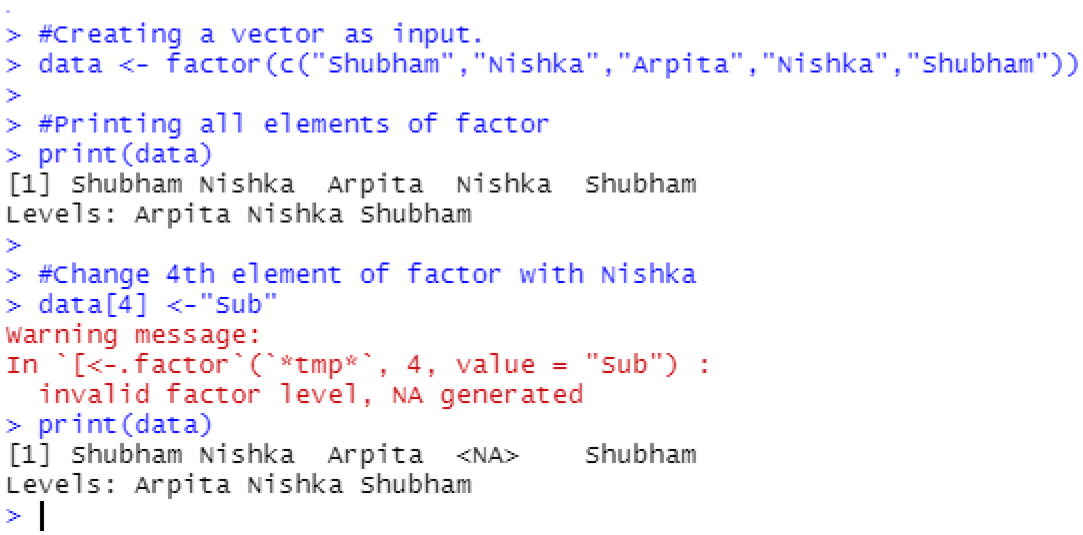


```
> #Creating a vector as input.
> data <- factor(c("Shubham","Nishka","Arpita","Nishka","Shubham"))
>
> #Printing all elements of factor
> print(data)
[1] Shubham Nishka  Arpita  Nishka  Shubham
Levels: Arpita Nishka Shubham
>
> #Change 4th element of factor with Nishka
> data[4] <-"Shubham"
> print(data)
[1] Shubham Nishka  Arpita  Shubham Shubham
Levels: Arpita Nishka Shubham
```

- we cannot choose values outside of its predefined levels means we cannot insert value if it's level is not present on it.

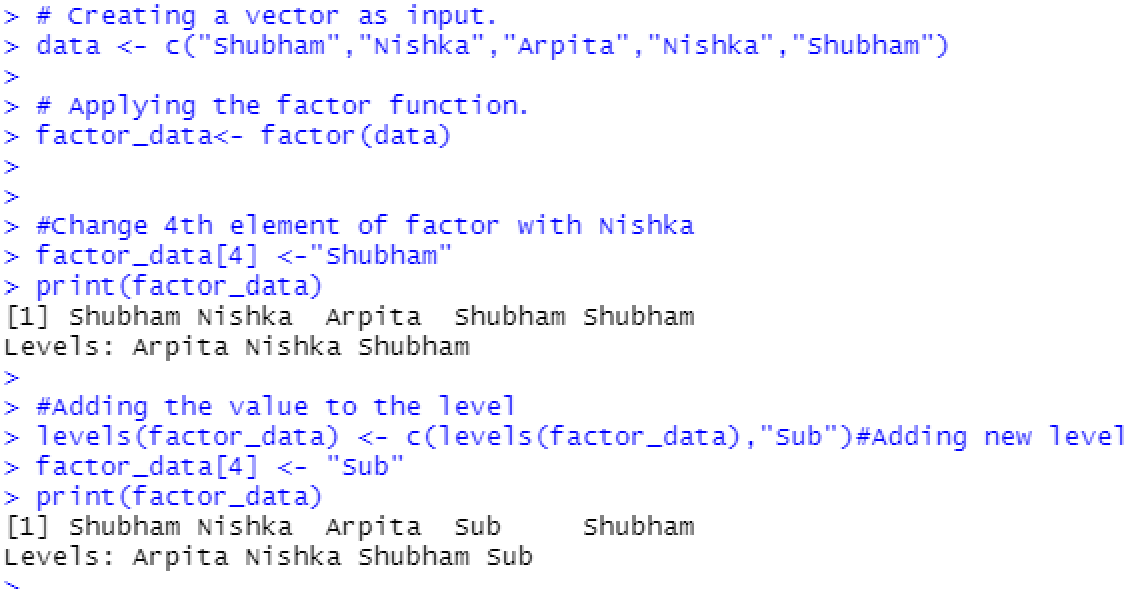
- For this purpose, we have to create a level of that value, and then we can add it to our factor.

Example:



```
> #Creating a vector as input.
> data <- factor(c("Shubham","Nishka","Arpita","Nishka","Shubham"))
>
> #Printing all elements of factor
> print(data)
[1] Shubham Nishka  Arpita  Nishka  Shubham
Levels: Arpita Nishka Shubham
>
> #Change 4th element of factor with Nishka
> data[4] <-"Sub"
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "Sub") :
  invalid factor level, NA generated
> print(data)
[1] Shubham Nishka  Arpita  <NA>    Shubham
Levels: Arpita Nishka Shubham
> 
```

Example:

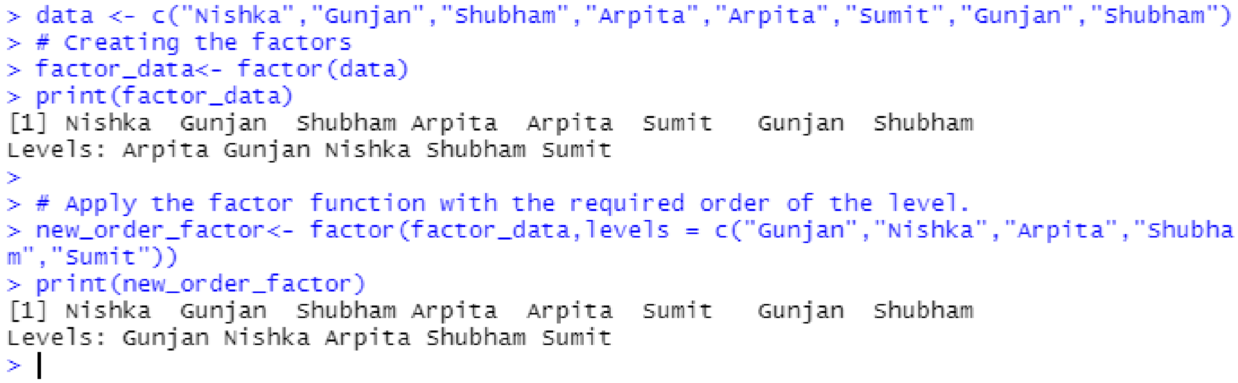


```
> # Creating a vector as input.
> data <- c("Shubham","Nishka","Arpita","Nishka","Shubham")
>
> # Applying the factor function.
> factor_data<- factor(data)
>
>
> #Change 4th element of factor with Nishka
> factor_data[4] <-"Shubham"
> print(factor_data)
[1] Shubham Nishka  Arpita  Shubham Shubham
Levels: Arpita Nishka Shubham
>
> #Adding the value to the level
> levels(factor_data) <- c(levels(factor_data),"Sub")#Adding new level
> factor_data[4] <- "Sub"
> print(factor_data)
[1] Shubham Nishka  Arpita  Sub     Shubham
Levels: Arpita Nishka Shubham Sub
>
```

Changing order of the levels

We can change the order of the levels in the factor with the help of the factor function.

Example:



```
> data <- c("Nishka","Gunjan","Shubham","Arpita","Arpita","Sumit","Gunjan","Shubham")
> # Creating the factors
> factor_data<- factor(data)
> print(factor_data)
[1] Nishka  Gunjan  Shubham Arpita  Arpita  Sumit   Gunjan  Shubham
Levels: Arpita Gunjan Nishka Shubham Sumit
> 
> # Apply the factor function with the required order of the level.
> new_order_factor<- factor(factor_data,levels = c("Gunjan","Nishka","Arpita","Shubha
m","Sumit"))
> print(new_order_factor)
[1] Nishka  Gunjan  Shubham Arpita  Arpita  Sumit   Gunjan  Shubham
Levels: Gunjan Nishka Arpita Shubham Sumit
> |
```

Loops

Loops can execute a block of code as long as a specified condition is reached.

R has two loop commands:

- while loops
- for loops

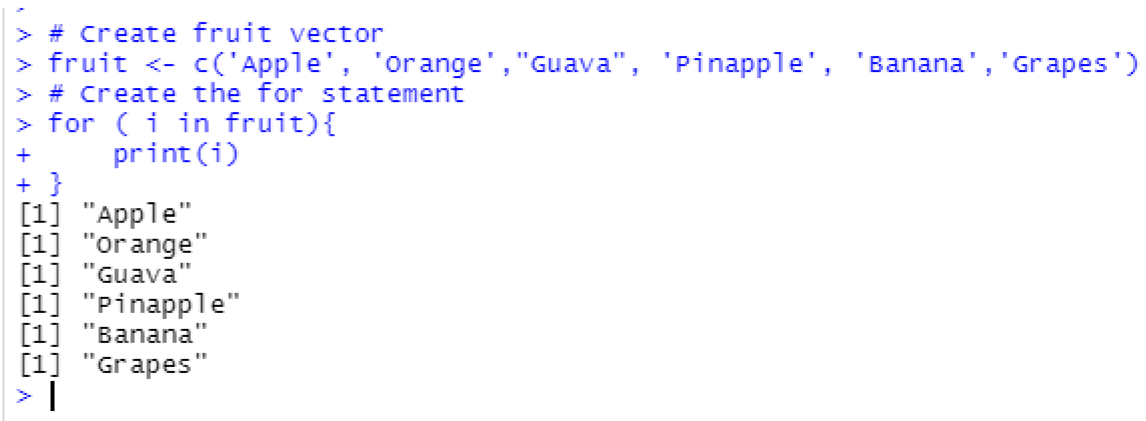
for loop

A for loop is used for iterating over a sequence.

syntax:

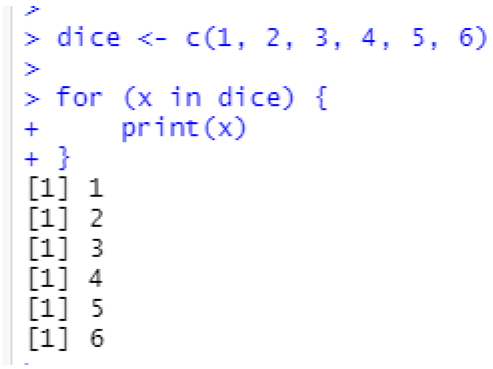
for (value **in** vector) {
  statements
}

Example:

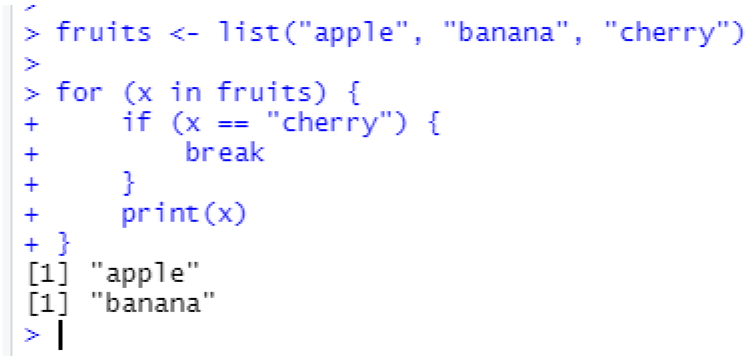


```
> # Create fruit vector
> fruit <- c('Apple', 'Orange',"Guava", 'Pinapple', 'Banana','Grapes')
> # Create the for statement
> for ( i in fruit){
+     print(i)
+ }
[1] "Apple"
[1] "Orange"
[1] "Guava"
[1] "Pinapple"
[1] "Banana"
[1] "Grapes"
> |
```

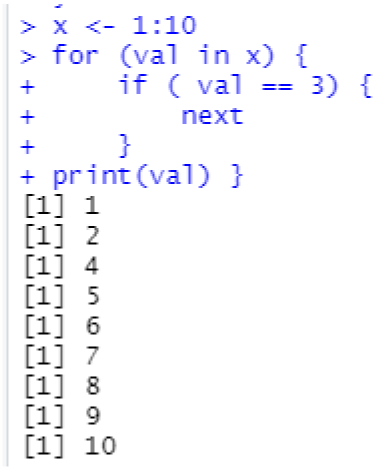
Example:



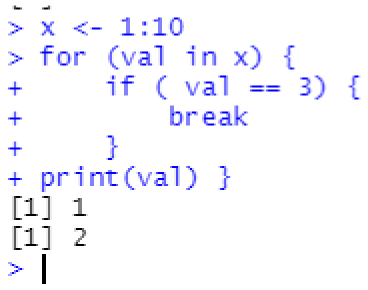
```
> dice <- c(1, 2, 3, 4, 5, 6)
>
> for (x in dice) {
+     print(x)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```



```
> fruits <- list("apple", "banana", "cherry")
>
> for (x in fruits) {
+     if (x == "cherry") {
+         break
+     }
+     print(x)
+ }
[1] "apple"
[1] "banana"
> 
```



```
> x <- 1:10
> for (val in x) {
+     if ( val == 3) {
+         next
+     }
+ print(val) }
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```



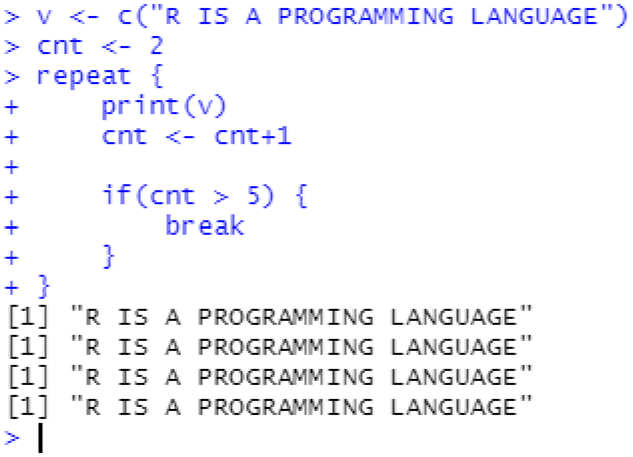
```
> x <- 1:10
> for (val in x) {
+     if ( val == 3) {
+         break
+     }
+ print(val) }
[1] 1
[1] 2
> 
```

repeat loop

- It is a special type of loop in which there is no condition to exit from the loop.
- For exiting, we include a break statement with a user-defined condition.
- This property of the loop makes it different from the other loops.

Syntax:

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



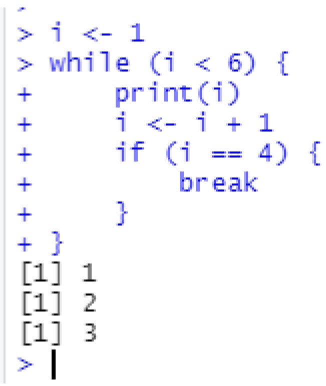
```
> v <- c("R IS A PROGRAMMING LANGUAGE")
> cnt <- 2
> repeat {
+     print(v)
+     cnt <- cnt+1
+
+     if(cnt > 5) {
+         break
+     }
+ }
[1] "R IS A PROGRAMMING LANGUAGE"
[1] "R IS A PROGRAMMING LANGUAGE"
[1] "R IS A PROGRAMMING LANGUAGE"
[1] "R IS A PROGRAMMING LANGUAGE"
> 
```

while loop

- A while loop is a type of control flow statement which is used to iterate a block of code several numbers of times.
- The while loop terminates when the value of the Boolean expression will be false.

syntax:

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



```
> i <- 1
> while (i < 6) {
+     print(i)
+     i <- i + 1
+     if (i == 4) {
+         break
+     }
+ }
[1] 1
[1] 2
[1] 3
> 
```

R Function

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- R provides a series of in-built functions, and it allows the user to create their own functions.

Syntax:

```
func_name <- function(arg_1, arg_2, ...) {

  Function body

}
```

Components of a function

Function Name

The function name is the actual name of the function. In R, the function is stored as an object with its name.

Arguments

- In function, arguments are optional means a function may or may not contain arguments, and these arguments can have default values also.
- We pass a value to the argument when a function is invoked.

Function Body

The function body contains a set of statements which defines what the function does.

Return value

It is the last expression in the function body which is to be evaluated.

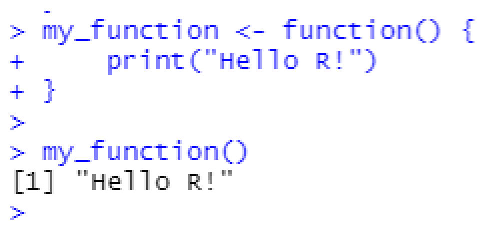
R also has two types of function

- **Built-in Function**
- **User-defined Function.**

Example: Creating a function

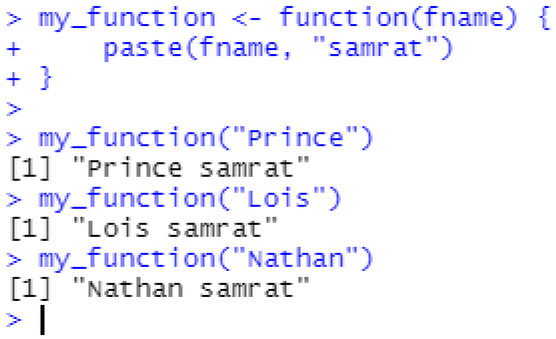
```
my_function <- function()
 {
# create a function with the name my_function
  print("Hello R!")
}
```

Example: Calling a function



```
> my_function <- function() {
+     print("Hello R!")
+ }
>
> my_function()
[1] "Hello R!"
>
```

Example:



```
> my_function <- function(fname) {
+     paste(fname, "samrat")
+ }
>
> my_function("Prince")
[1] "Prince samrat"
> my_function("Lois")
[1] "Lois samrat"
> my_function("Nathan")
[1] "Nathan samrat"
> 
```

Note:

Parameters or Arguments?

The terms "parameter" and "argument" can be used for the same thing: information that is passed into a function.

From a function's perspective:

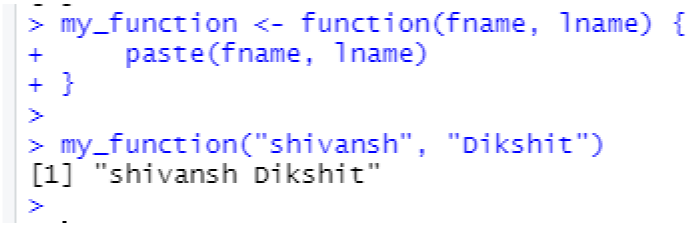
A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

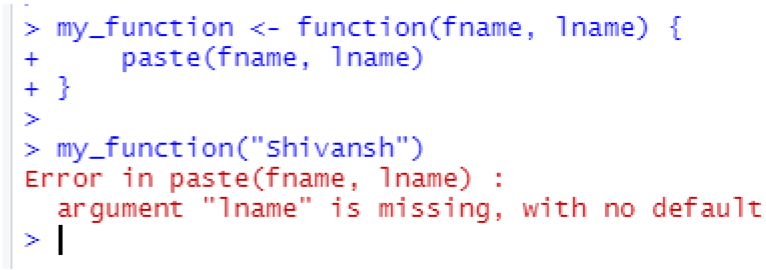
- By default, a function must be called with the correct number of arguments.
- Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less:

Example:



```
> my_function <- function(fname, lname) {
+     paste(fname, lname)
+ }
>
> my_function("shivansh", "Dikshit")
[1] "shivansh Dikshit"
>
```

Example:

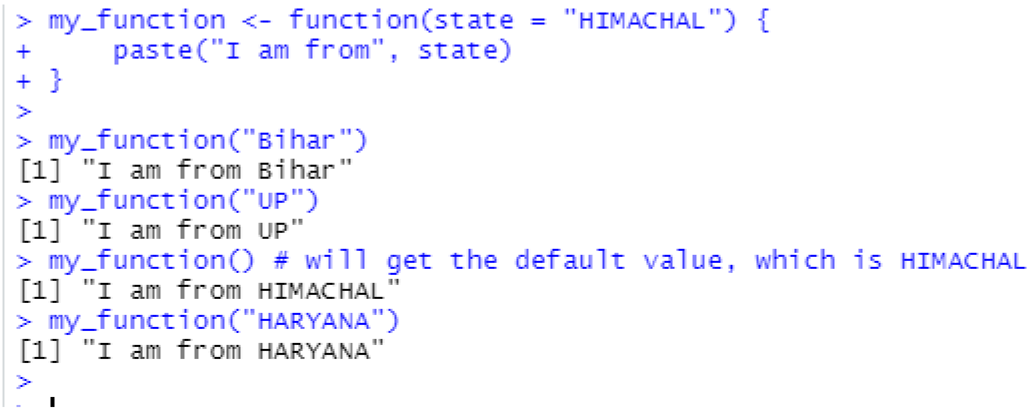


```
> my_function <- function(fname, lname) {
+     paste(fname, lname)
+ }
>
> my_function("Shivansh")
Error in paste(fname, lname) :
  argument "lname" is missing, with no default
> 
```

Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without an argument, it uses the default value:

Example:

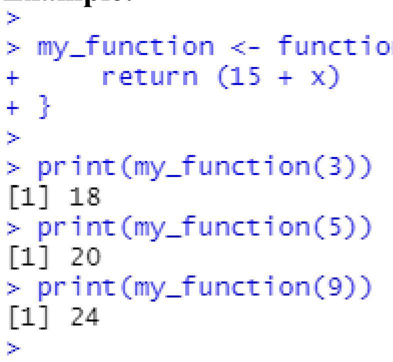


```
> my_function <- function(state = "HIMACHAL") {
+     paste("I am from", state)
+ }
>
> my_function("Bihar")
[1] "I am from Bihar"
> my_function("UP")
[1] "I am from UP"
> my_function() # will get the default value, which is HIMACHAL
[1] "I am from HIMACHAL"
> my_function("HARYANA")
[1] "I am from HARYANA"
>
```

Return Values

- To let a function return a result, use the return() function:

Example:

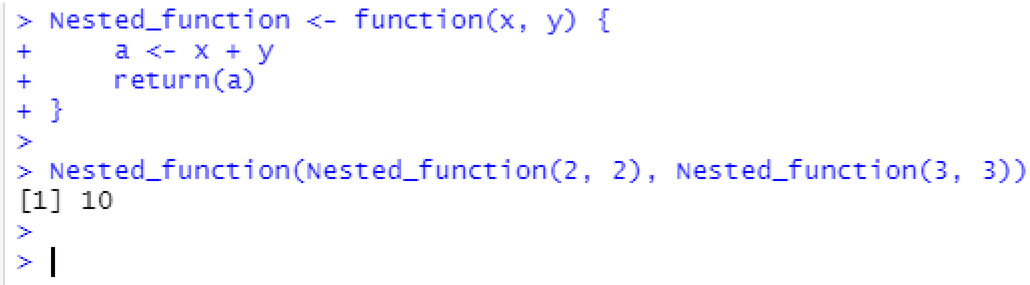


```
>
> my_function <- function(x) {
+     return (15 + x)
+ }
>
> print(my_function(3))
[1] 18
> print(my_function(5))
[1] 20
> print(my_function(9))
[1] 24
>
```

Nested Functions

There are two ways to create a nested function:

- **Call a function within another function.**

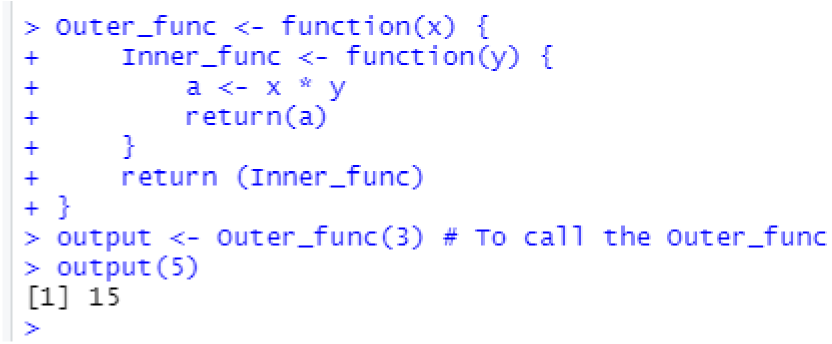


```
> Nested_function <- function(x, y) {
+     a <- x + y
+     return(a)
+ }
>
> Nested_function(Nested_function(2, 2), Nested_function(3, 3))
[1] 10
>
>
```

Example Explained

The function tells x to add y.
The first input Nested_function(2,2) is "x" of the main function.
The second input Nested_function(3,3) is "y" of the main function.
The output is therefore (2+2) + (3+3) = 10.

- **Write a function within a function.**

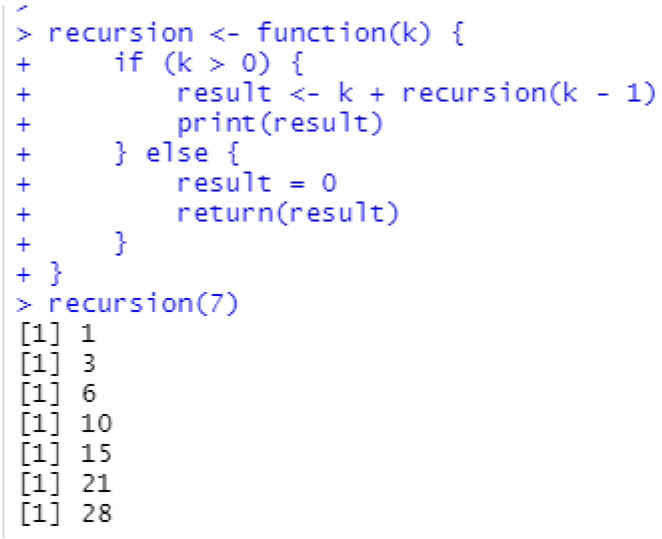


```
> outer_func <- function(x) {
+     Inner_func <- function(y) {
+         a <- x * y
+         return(a)
+     }
+     return (Inner_func)
+ }
> output <- outer_func(3) # To call the outer_func
> output(5)
[1] 15
> 
```

Recursion

- Recursion means that a function calls itself.
- This has the benefit of meaning that you can loop through data to reach a result.

Example:

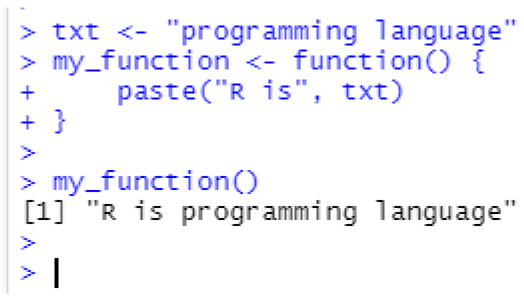


```
> recursion <- function(k) {
+     if (k > 0) {
+         result <- k + recursion(k - 1)
+         print(result)
+     } else {
+         result = 0
+         return(result)
+     }
+ }
> recursion(7)
[1] 1
[1] 3
[1] 6
[1] 10
[1] 15
[1] 21
[1] 28
```

Global Variables

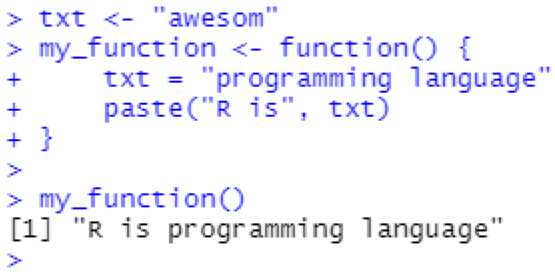
- Variables that are created outside of a function are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Example:



```
> txt <- "programming language"
> my_function <- function() {
+     paste("R is", txt)
+ }
>
> my_function()
[1] "R is programming language"
>
> 
```

Example:

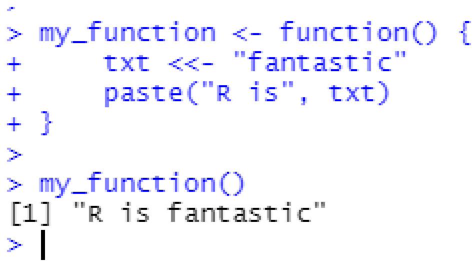


```
> txt <- "awesom"
> my_function <- function() {
+     txt = "programming language"
+     paste("R is", txt)
+ }
>
> my_function()
[1] "R is programming language"
>
```

The Global Assignment Operator

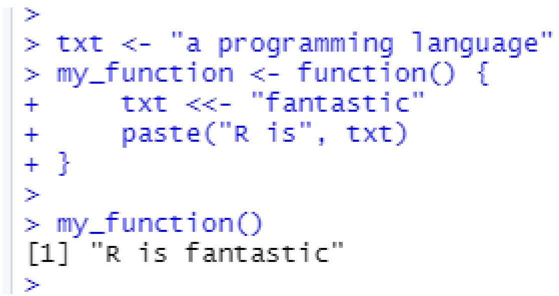
- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the global assignment operator <<- .

Example:



```
> my_function <- function() {
+     txt <<- "fantastic"
+     paste("R is", txt)
+ }
>
> my_function()
[1] "R is fantastic"
> 
```

Example:



```
>
> txt <- "a programming language"
> my_function <- function() {
+     txt <<- "fantastic"
+     paste("R is", txt)
+ }
>
> my_function()
[1] "R is fantastic"
>
```

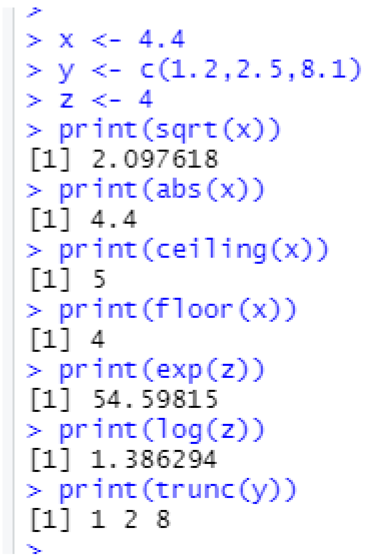
Built in Function

Math Functions

In R, there are the following functions which are used:

Function name	Description
abs(x)	It returns the absolute value of input x.
sqrt(x)	It returns the square root of input x.
ceiling(x)	It returns the smallest integer which is larger than or equal to x.
floor(x)	It returns the larger integer which is smaller than or equal to x.
trunc(x)	It returns the truncate value of input x.
log(x)	It returns the natural logarithm of input x.
exp(x)	It returns exponent.

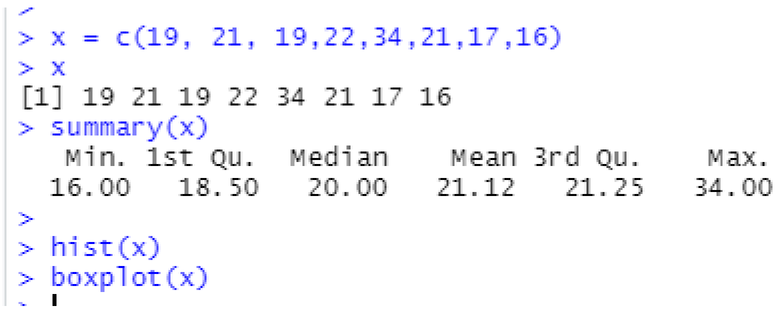
Example:



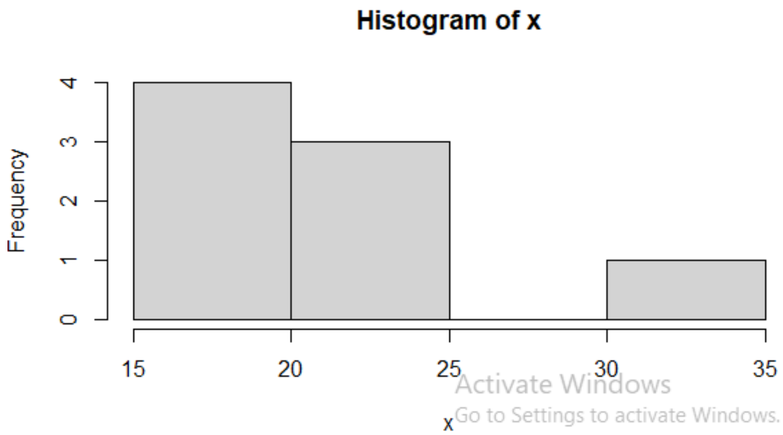
```
>
> x <- 4.4
> y <- c(1.2,2.5,8.1)
> z <- 4
> print(sqrt(x))
[1] 2.097618
> print(abs(x))
[1] 4.4
> print(ceiling(x))
[1] 5
> print(floor(x))
[1] 4
> print(exp(z))
[1] 54.59815
> print(log(z))
[1] 1.386294
> print(trunc(y))
[1] 1 2 8
>
```

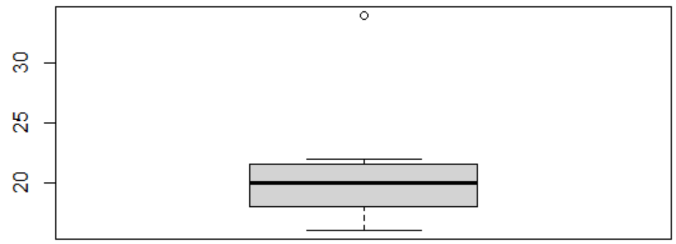
Plot Functions in R

Example:

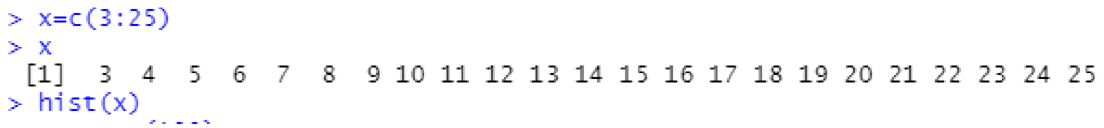


```
>
> x = c(19, 21, 19,22,34,21,17,16)
> x
[1] 19 21 19 22 34 21 17 16
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  16.00   18.50   20.00   21.12   21.25   34.00 
>
> hist(x)
> boxplot(x)
>
```

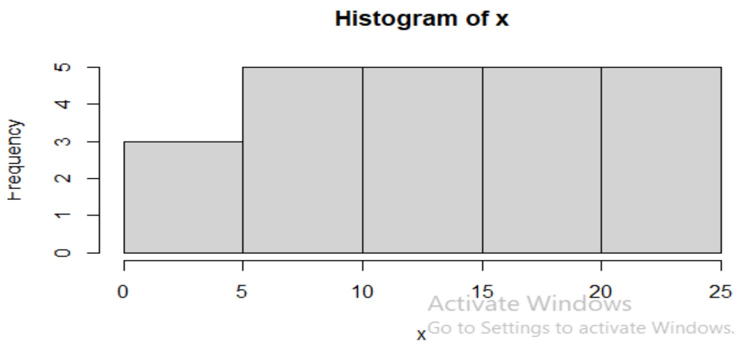




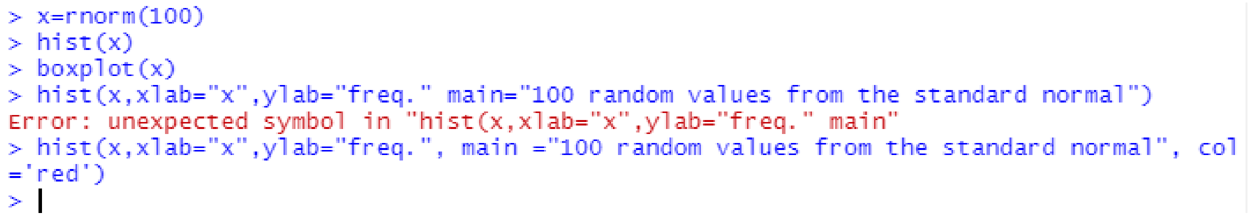
Example:



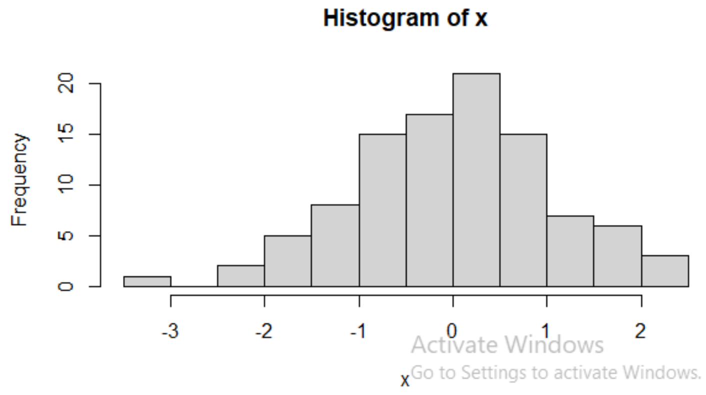
```
> x=c(3:25)
> x
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
> hist(x)
```

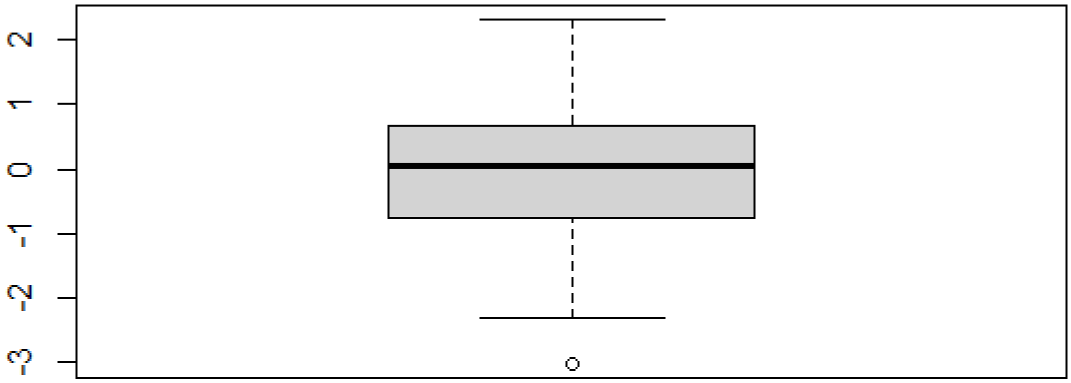


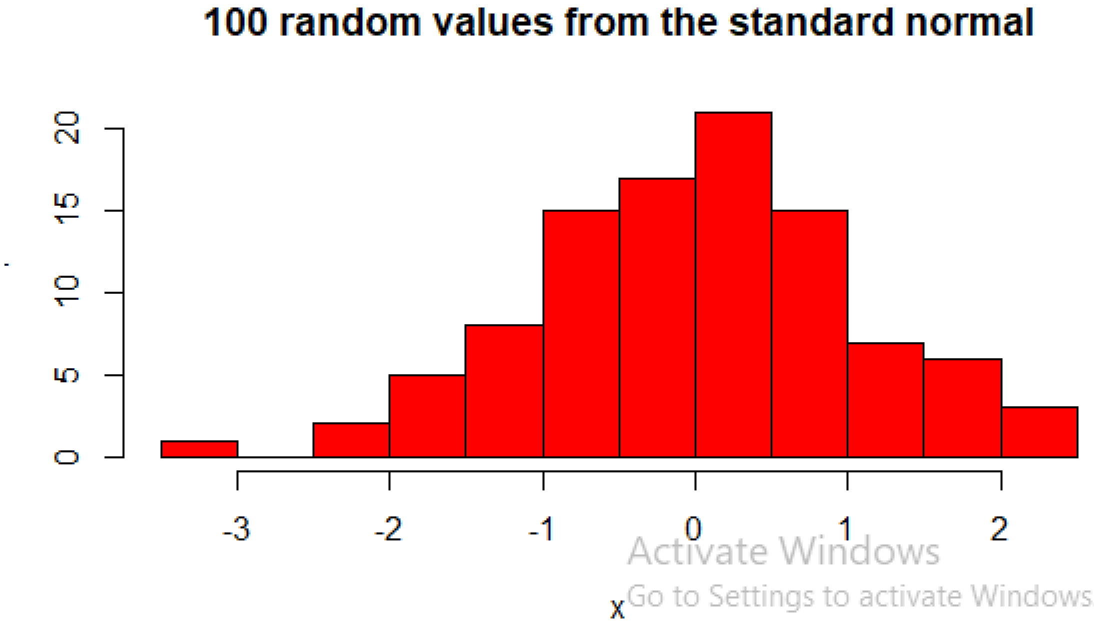
Example:



```
> x=rnorm(100)
> hist(x)
> boxplot(x)
> hist(x,xlab="x",ylab="freq." main="100 random values from the standard normal")
Error: unexpected symbol in "hist(x,xlab="x",ylab="freq." main"
> hist(x,xlab="x",ylab="freq.", main ="100 random values from the standard normal", col
='red')
> 
```







1. **Plot**

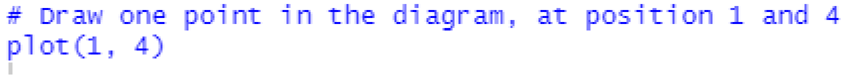
- The **plot()** function is used to draw points (markers) in a diagram, which takes positive numbers as vector input. Additional parameters are used to control labels, colors, titles, etc..
- The function takes parameters for specifying points in the diagram.
- Parameter 1 specifies points on the x-axis.
- Parameter 2 specifies points on the y-axis.

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

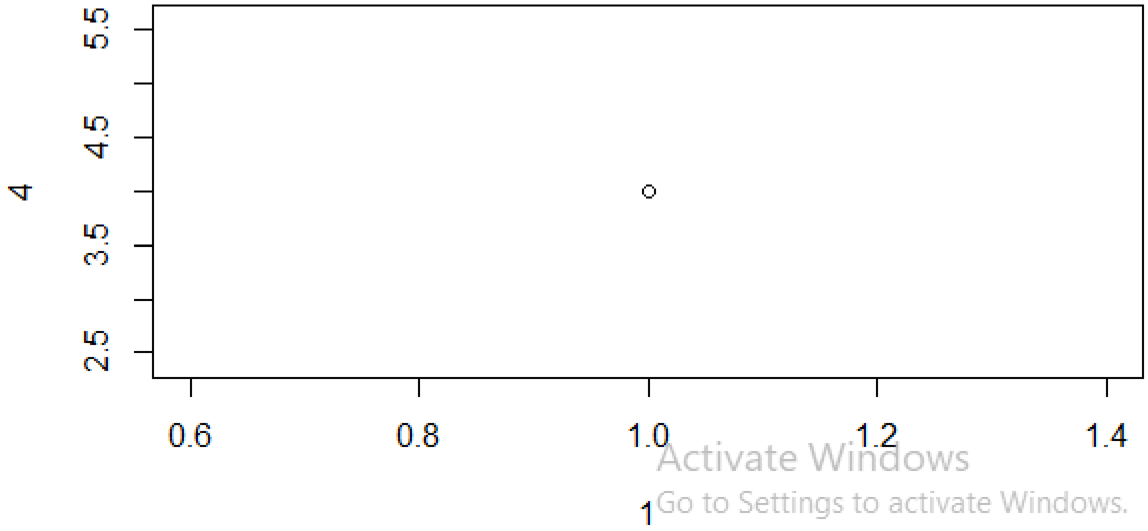
pie(X, Labels, Radius, Main, Col, Clockwise)

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.

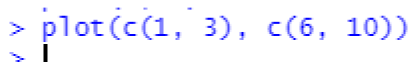
Example:Draw one point in the diagram, at position (1) and position (4):



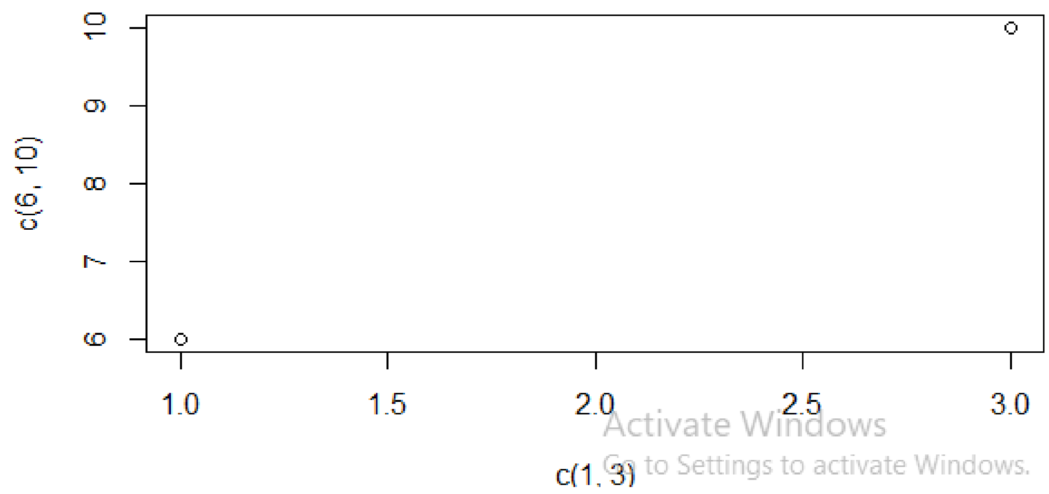
```
# Draw one point in the diagram, at position 1 and 4
plot(1, 4)
```



Example: Draw two points in the diagram, at position (1,3) and position (6,10):

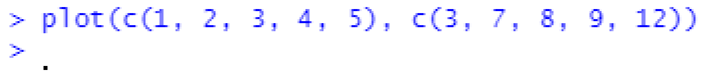


```
> plot(c(1, 3), c(6, 10))
> 
```

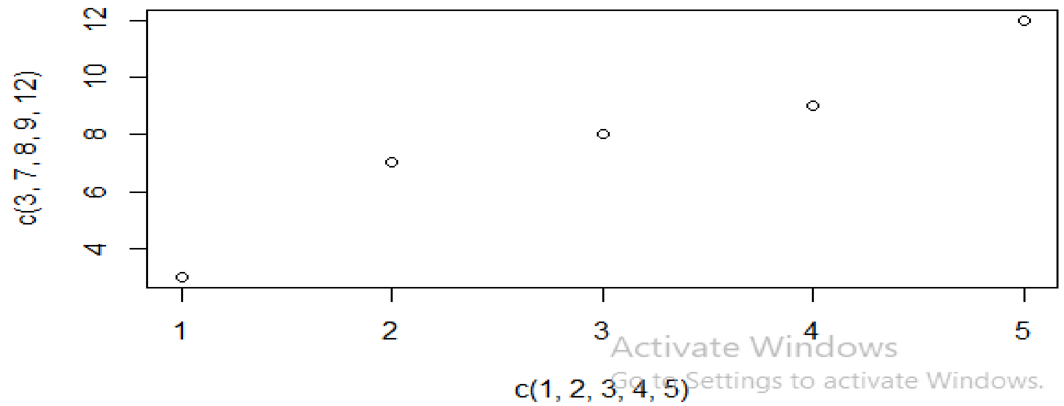


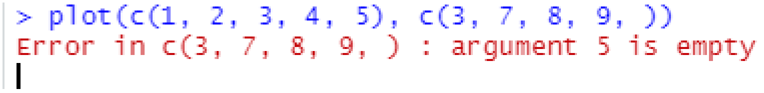
Multiple points

You can plot as many points as you like, however you have to use the same number of points on both axis.



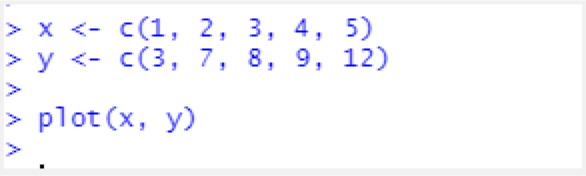
```
> plot(c(1, 2, 3, 4, 5), c(3, 7, 8, 9, 12))
> 
```



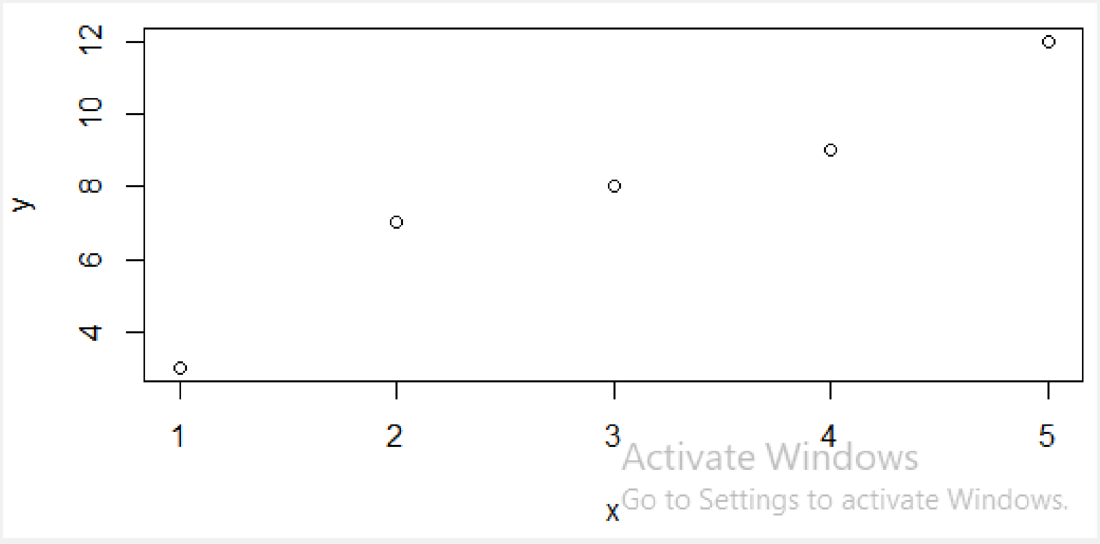


```
> plot(c(1, 2, 3, 4, 5), c(3, 7, 8, 9, ))
Error in c(3, 7, 8, 9, ) : argument 5 is empty
```

Example:



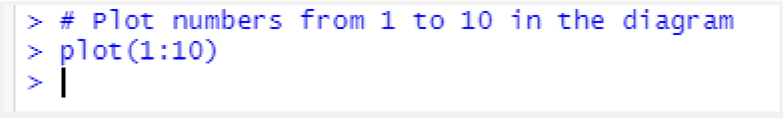
```
> x <- c(1, 2, 3, 4, 5)
> y <- c(3, 7, 8, 9, 12)
>
> plot(x, y)
>
```



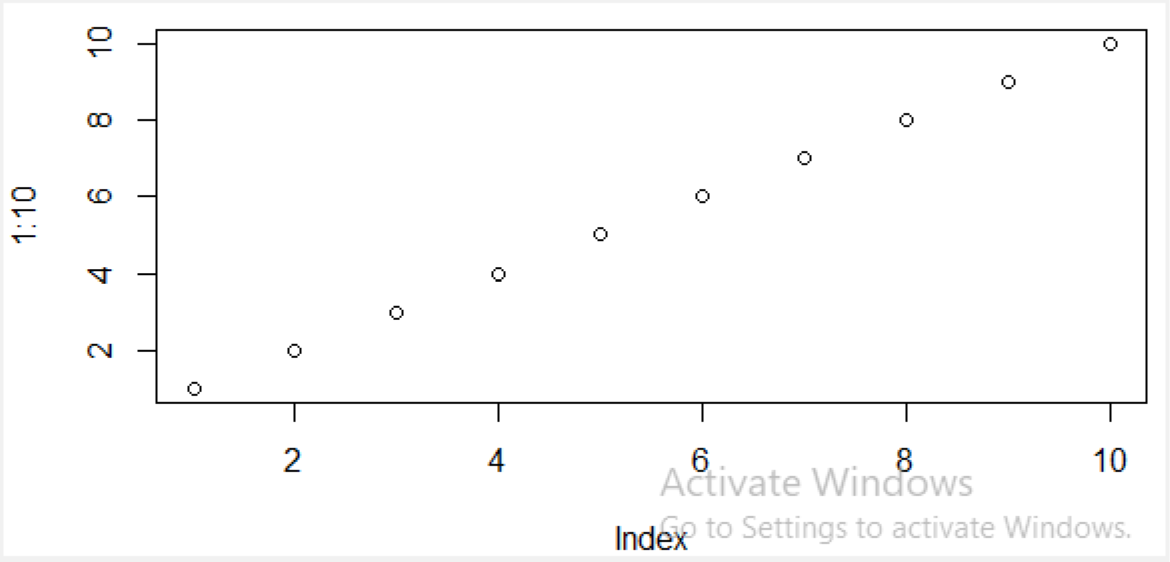
Sequences of points

If you want to draw dots in sequence, on both the x-axis and the y-axis, use the **:** operator-

Example:



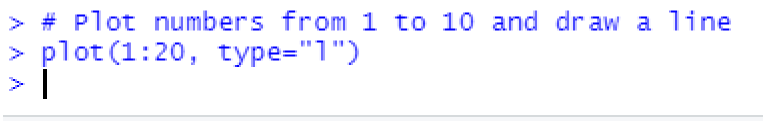
```
> # Plot numbers from 1 to 10 in the diagram
> plot(1:10)
>
```



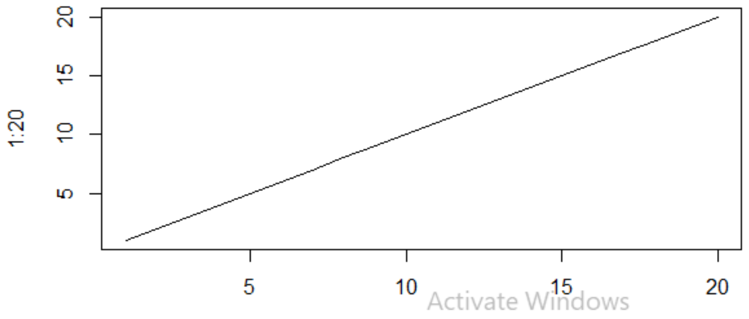
Draw a line

The **plot()** function also takes a **type** parameter with the value **l** to draw a line to connect all the points in the diagram.

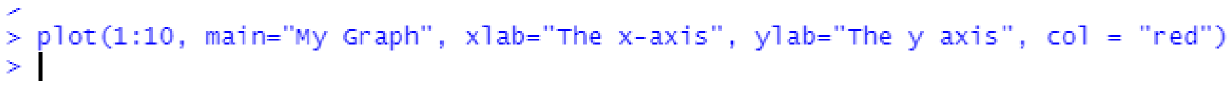
Example: Plot Labels:



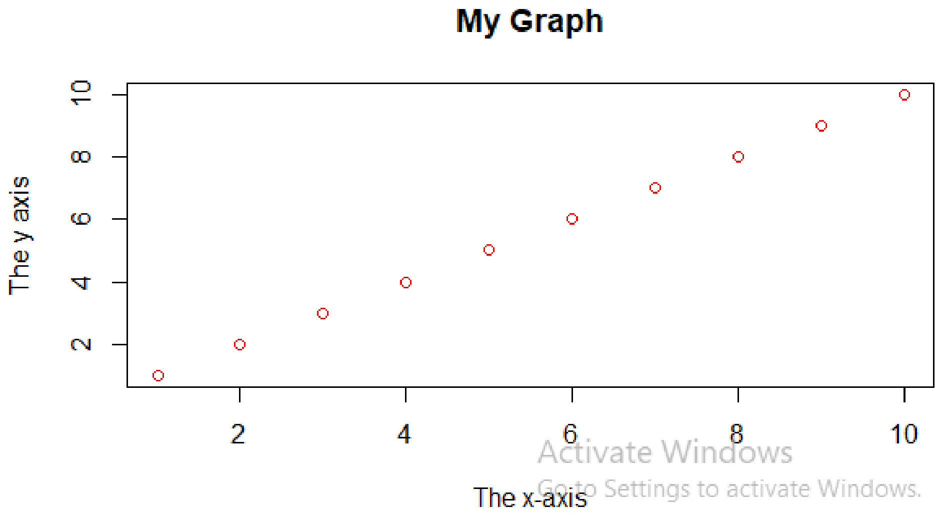
```
> # Plot numbers from 1 to 10 and draw a line
> plot(1:20, type="l")
> 
```



Example:



```
> plot(1:10, main="My Graph", xlab="The x-axis", ylab="The y axis", col = "red")
> 
```



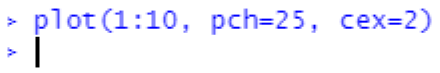
Size

Use **cex=*number*** to change the size of the points (1 is default, while 0.5 means 50% smaller, and 2 means 100% larger):

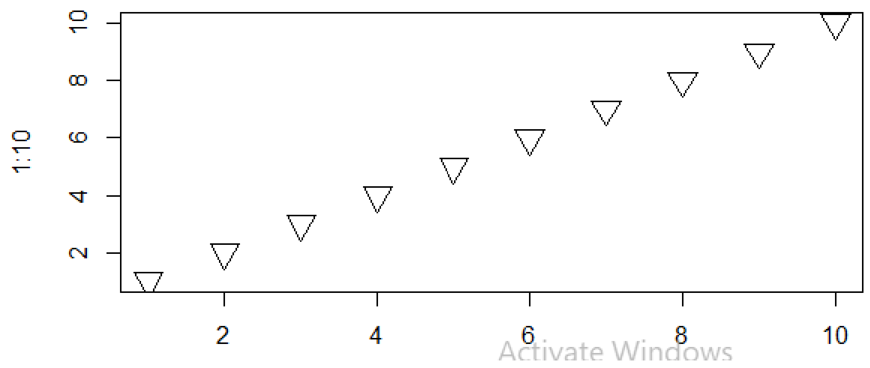
Point shape

Use **pch** with a value from 0 to 25 to change the point shape format:

Example:



```
> plot(1:10, pch=25, cex=2)
> 
```

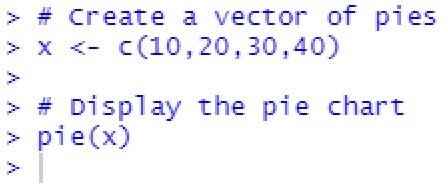


Note:

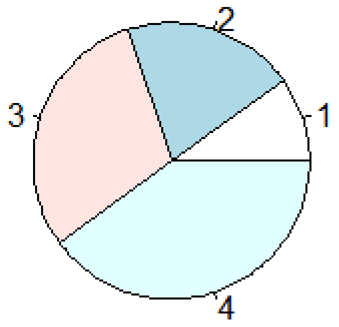
The values of the **pch** parameter ranges from 0 to 25, which means that we can choose up to 26 different types of point shapes.

Pie chart

Example:



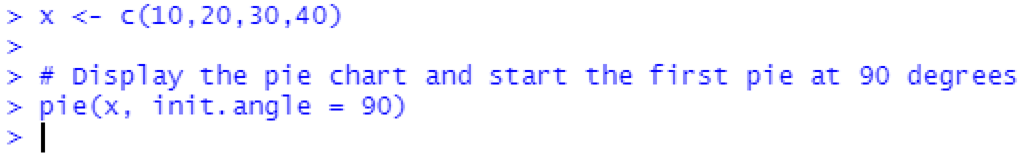
```
> # Create a vector of pies
> x <- c(10,20,30,40)
>
> # Display the pie chart
> pie(x)
>
```



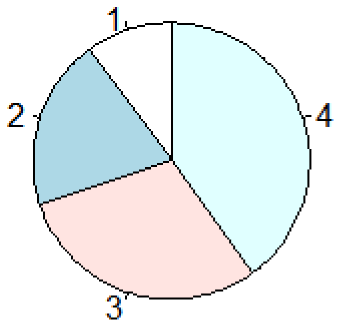
Start Angle

- You can change the start angle of the pie chart with the **init.angle** parameter.
- The value of **init.angle** is defined with angle in degrees, where default angle is 0.

Ex:



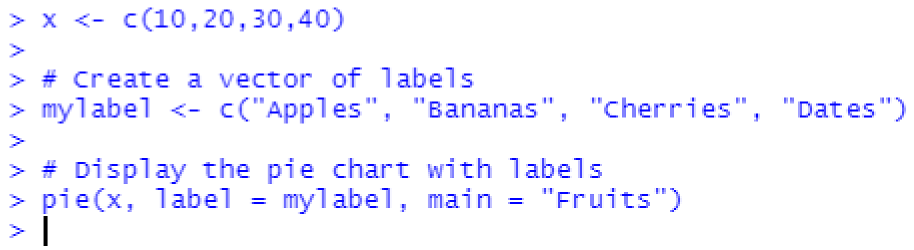
```
> x <- c(10,20,30,40)
>
> # Display the pie chart and start the first pie at 90 degrees
> pie(x, init.angle = 90)
>
```



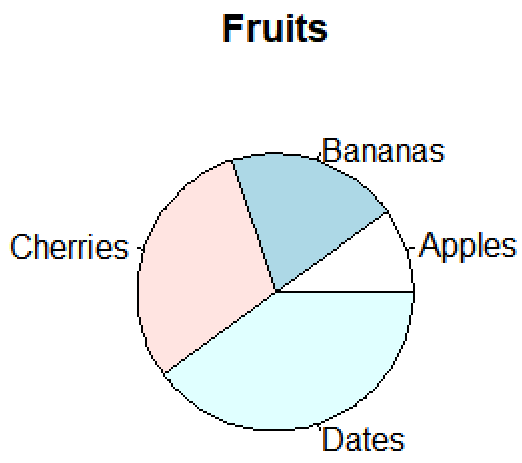
Labels and Header

- Use the **label** parameter to add a label to the pie chart, and use the **main** parameter to add a header:

Ex:



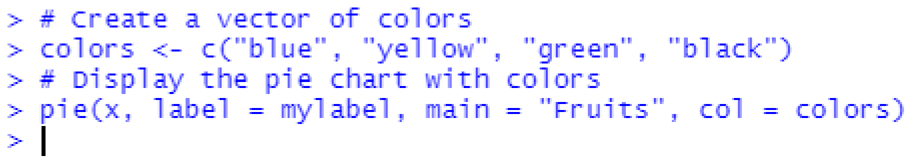
```
> x <- c(10,20,30,40)
>
> # Create a vector of labels
> mylabel <- c("Apples", "Bananas", "Cherries", "Dates")
>
> # Display the pie chart with labels
> pie(x, label = mylabel, main = "Fruits")
> 
```



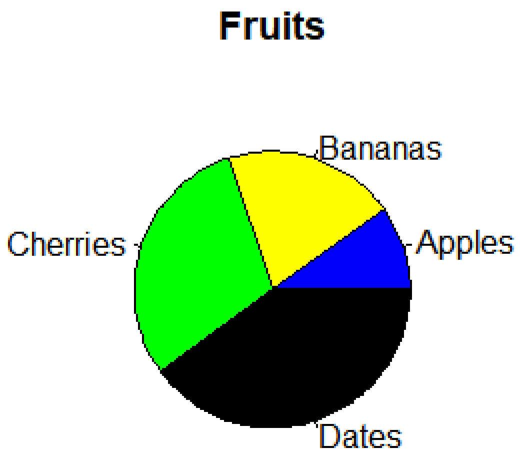
Colors

- You can add a color to each pie with the **col** parameter:

Example:



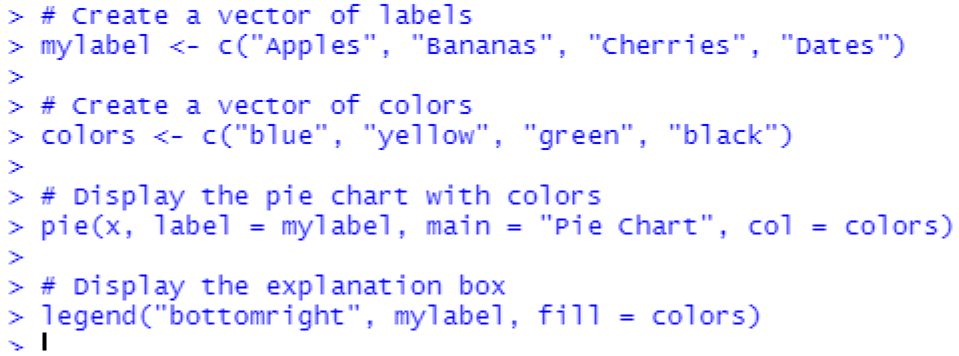
```
> # Create a vector of colors
> colors <- c("blue", "yellow", "green", "black")
> # Display the pie chart with colors
> pie(x, label = mylabel, main = "Fruits", col = colors)
> 
```



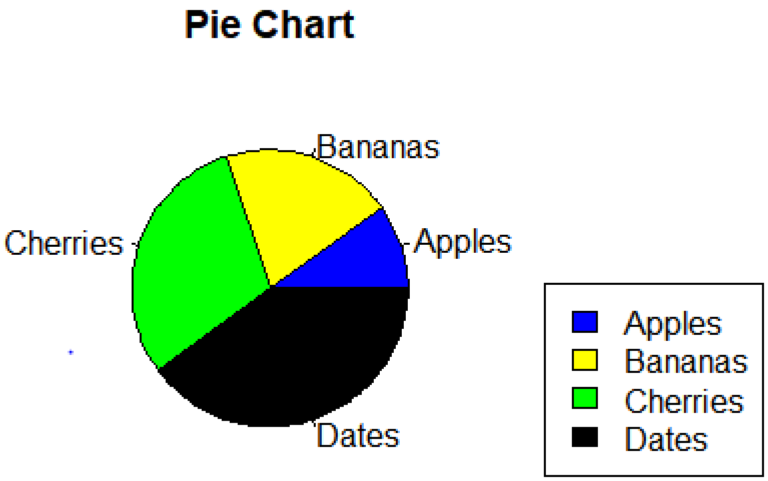
Legend

- To add a list of explanation for each pie, use the **legend()** function:

Example:



```
> # Create a vector of labels
> mylabel <- c("Apples", "Bananas", "Cherries", "Dates")
>
> # Create a vector of colors
> colors <- c("blue", "yellow", "green", "black")
>
> # Display the pie chart with colors
> pie(x, label = mylabel, main = "Pie Chart", col = colors)
>
> # Display the explanation box
> legend("bottomright", mylabel, fill = colors)
> 
```



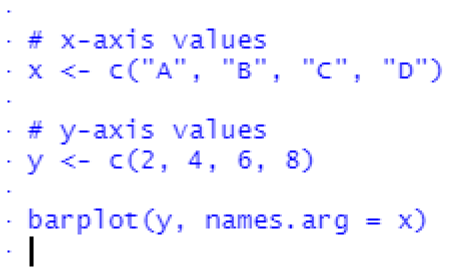
Note:

- The legend can be positioned as either:
- bottomright, bottom, bottomleft, left, topleft, top, topright, right, center.

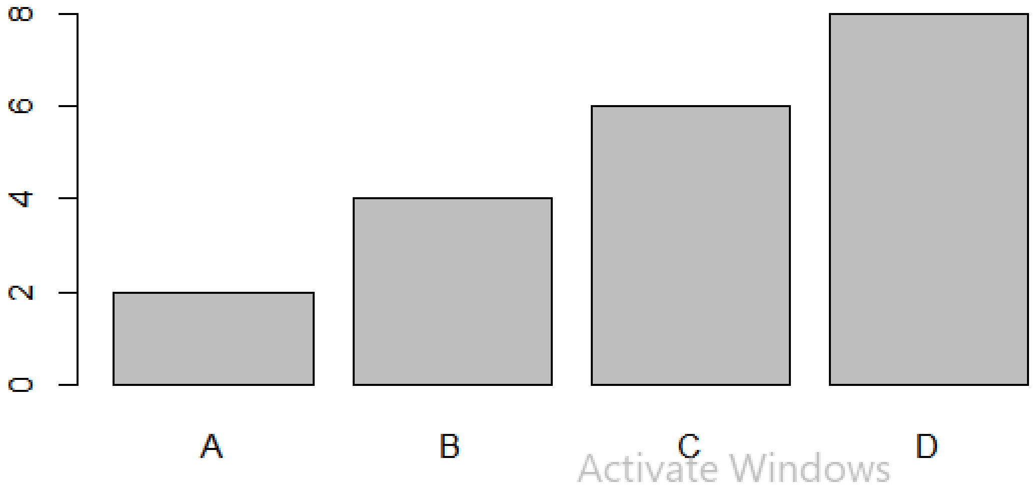
Bar Charts

- A bar chart uses rectangular bars to visualize data. Bar charts can be displayed horizontally or vertically.
- The height or length of the bars are proportional to the values they represent.
- Use the **barplot()** function to draw a vertical bar chart.

Ex:



```
# x-axis values
x <- c("A", "B", "C", "D")

# y-axis values
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x)
```



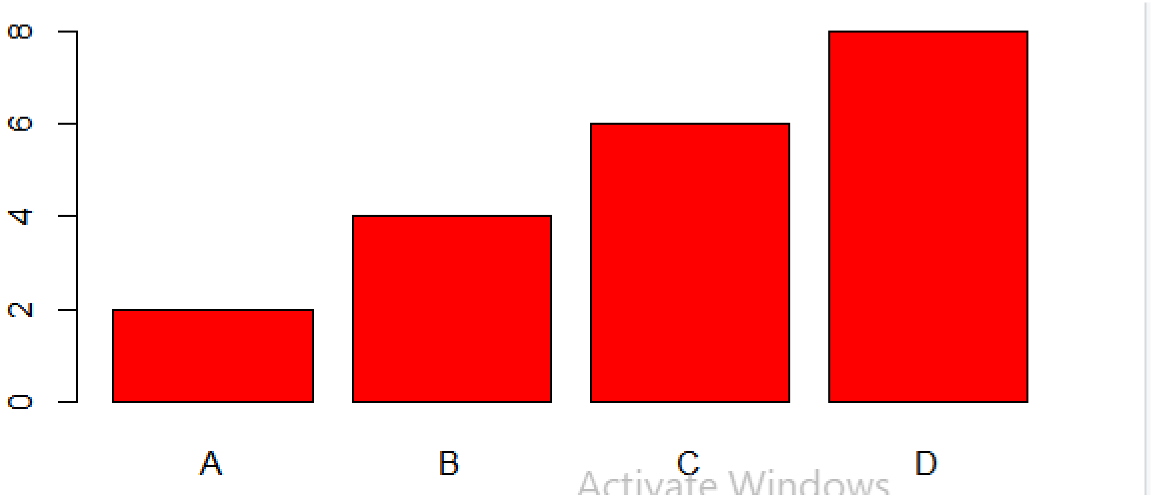
Note:

- The **x** variable represents values in the x-axis **(A,B,C,D)**
- The **y** variable represents values in the y-axis **(2,4,6,8)**
- Then we use the **barplot()** function to create a bar chart of the values
- **names.arg** defines the names of each observation in the x-axis.

Bar Color

- Use the **col** parameter to change the color of the bars:

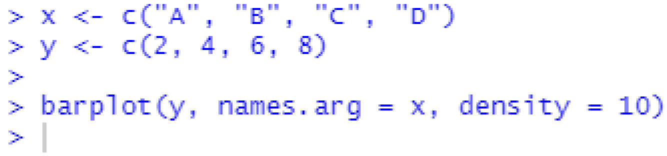
Ex:



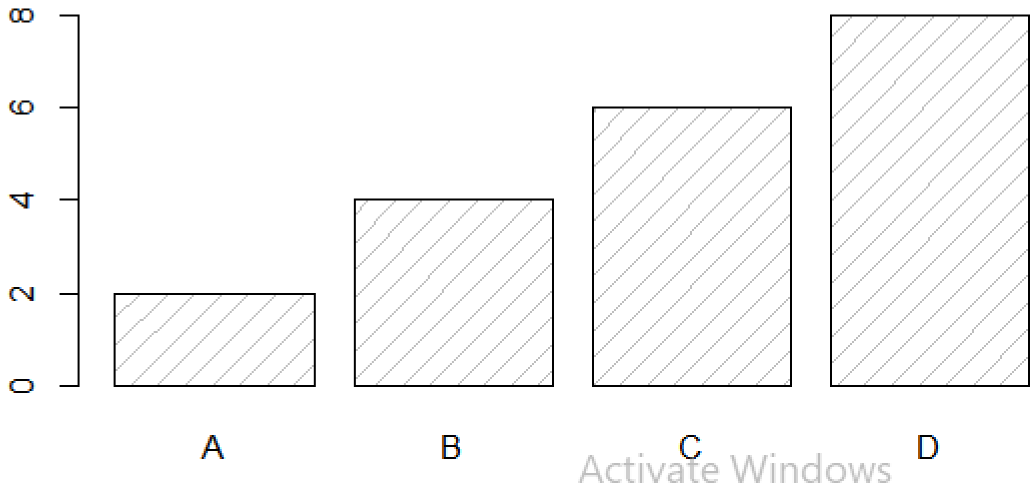
Density / Bar Texture

- To change the bar texture, use the **density** parameter:

Ex:



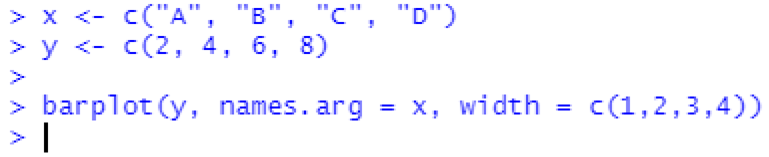
```
> x <- c("A", "B", "C", "D")
> y <- c(2, 4, 6, 8)
>
> barplot(y, names.arg = x, density = 10)
> |
```



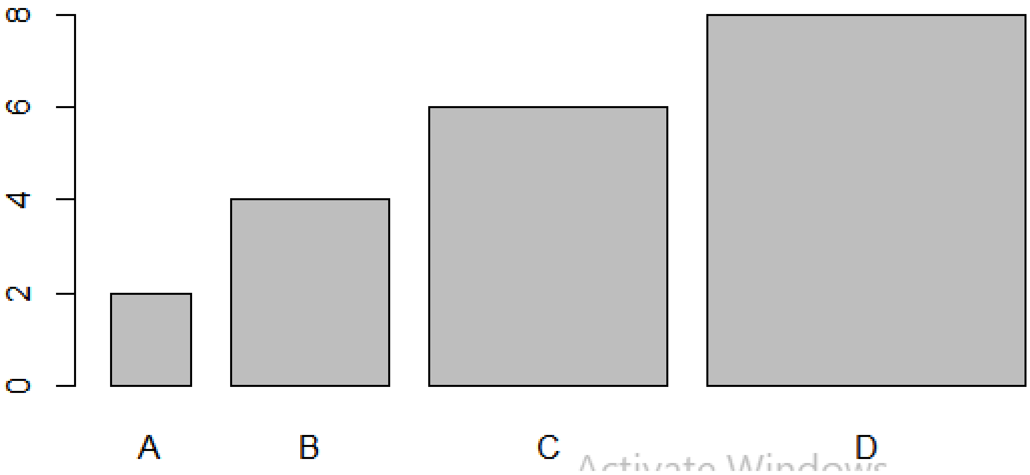
Bar Width

- Use the **width** parameter to change the width of the bars:

Ex:



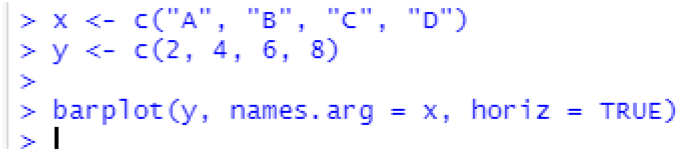
```
> x <- c("A", "B", "C", "D")
> y <- c(2, 4, 6, 8)
>
> barplot(y, names.arg = x, width = c(1,2,3,4))
> 
```



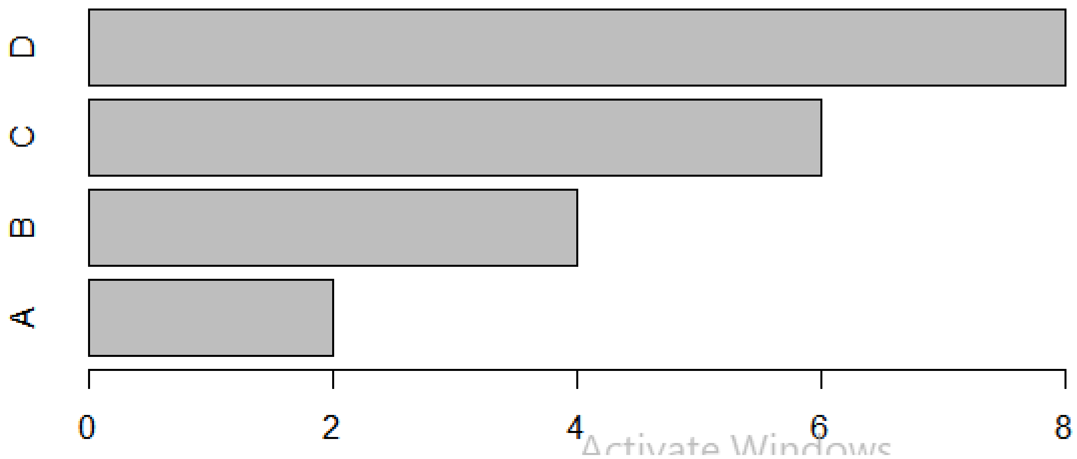
Horizontal Bars

- If you want the bars to be displayed horizontally instead of vertically, use **horiz=TRUE:**

Ex:



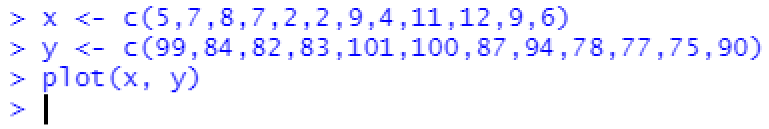
```
> x <- c("A", "B", "C", "D")
> y <- c(2, 4, 6, 8)
>
> barplot(y, names.arg = x, horiz = TRUE)
> 
```



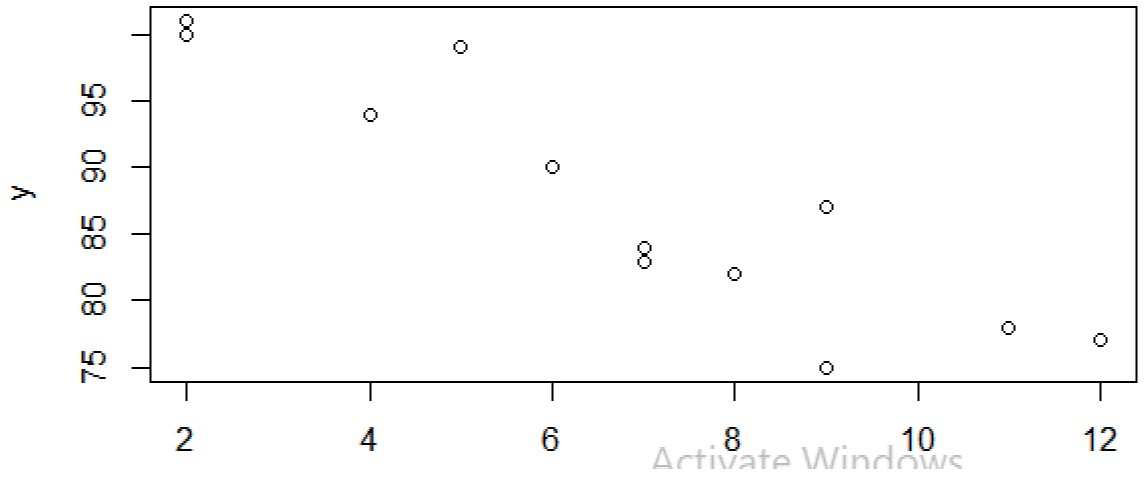
Scatter Plot

- A "scatter plot" is a type of plot used to display the relationship between two numerical variables, and plots one dot for each observation.
- It needs two vectors of same length, one for the x-axis (horizontal) and one for the y-axis (vertical).

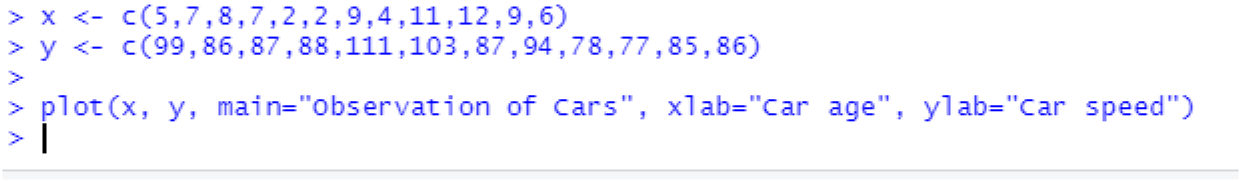
Ex:



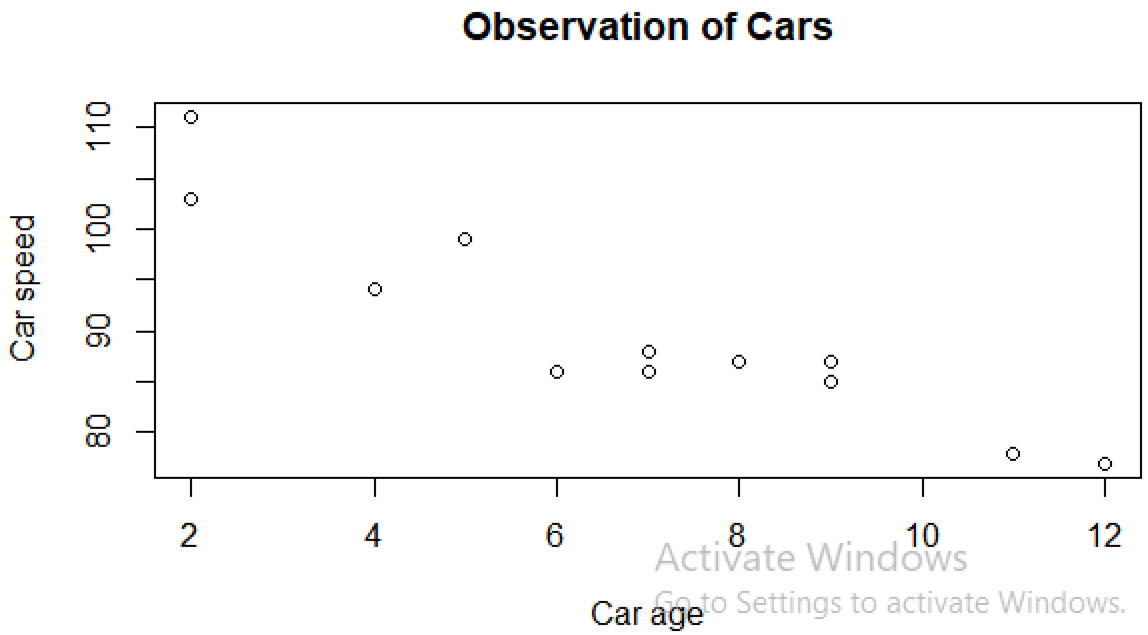
```
> x <- c(5,7,8,7,2,2,9,4,11,12,9,6)
> y <- c(99,84,82,83,101,100,87,94,78,77,75,90)
> plot(x, y)
> 
```



Ex:



```
> x <- c(5,7,8,7,2,2,9,4,11,12,9,6)
> y <- c(99,86,87,88,111,103,87,94,78,77,85,86)
>
> plot(x, y, main="Observation of Cars", xlab="Car age", ylab="Car speed")
> 
```



Note:

- The x-axis shows how old the car is.
- The y-axis shows the speed of the car when it passes.

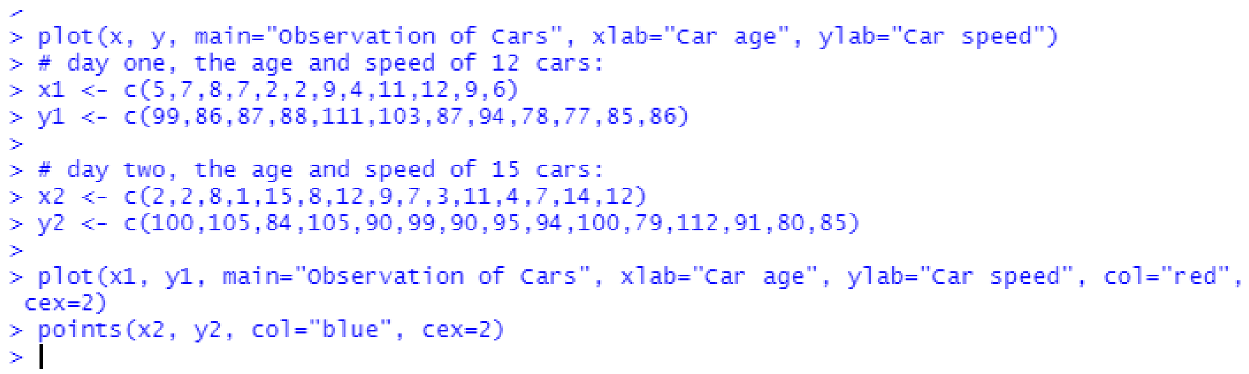
Q.1) Relationships between the observations?

- It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 12 cars.

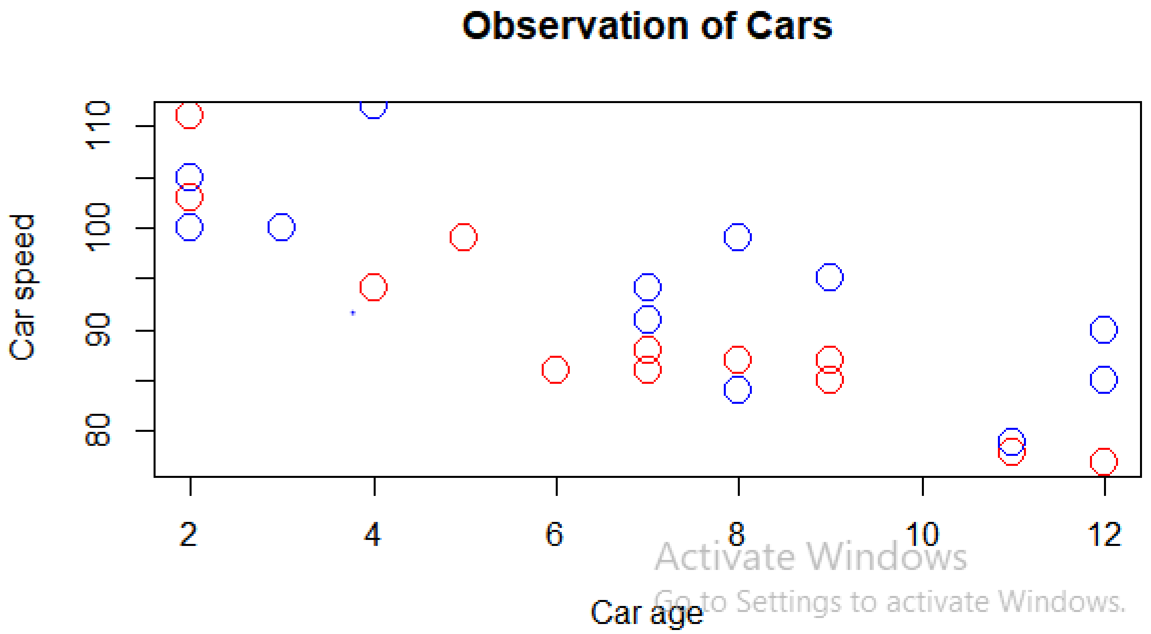
Compare Plots

- In above example, we have a relationship between the car speed and age,
- What if we plot the observations from another day as well? Will the scatter plot tell us something else?
- To compare the plot with another plot, use the **points()** function:

Example:



```
> plot(x, y, main="Observation of Cars", xlab="Car age", ylab="Car speed")
> # day one, the age and speed of 12 cars:
> x1 <- c(5,7,8,7,2,2,9,4,11,12,9,6)
> y1 <- c(99,86,87,88,111,103,87,94,78,77,85,86)
>
> # day two, the age and speed of 15 cars:
> x2 <- c(2,2,8,1,15,8,12,9,7,3,11,4,7,14,12)
> y2 <- c(100,105,84,105,90,99,90,95,94,100,79,112,91,80,85)
>
> plot(x1, y1, main="Observation of Cars", xlab="Car age", ylab="Car speed", col="red",
 cex=2)
> points(x2, y2, col="blue", cex=2)
> |
```



Note:

- Red represents the values of day 1, while blue represents day 2.

Statistics Introduction

- Statistics is the science of analyzing, reviewing and concluding data.

Some basic statistical numbers include:

- Mean, median and mode
- Minimum and maximum value
- Percentiles
- Variance and Standard Deviation

- Covariance and Correlation
- Probability distributions

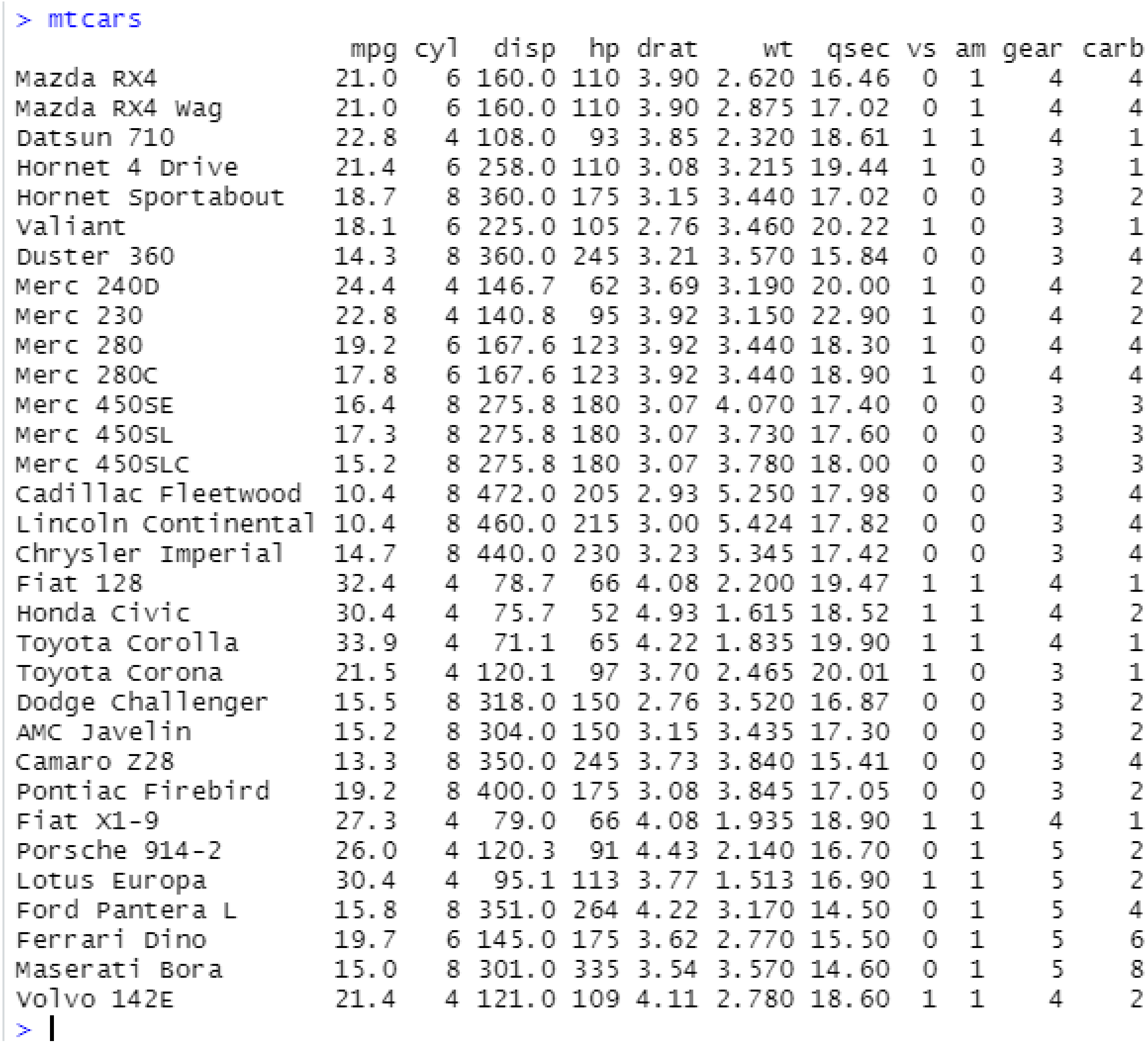
It has many built-in functionalities, in addition to libraries for the exact purpose of statistical analysis.

Data Set

- A data set is a collection of data, often presented in a table.
- There is a popular built-in data set in R called "mtcars" (Motor Trend Car Road Tests), which is retrieved from the 1974 Motor Trend US Magazine.

Note: We will use the **mtcars** data set, for statistical purposes:

Example:To print the **mtcars** dataset:-

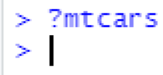


```
> mtcars
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
> 
```

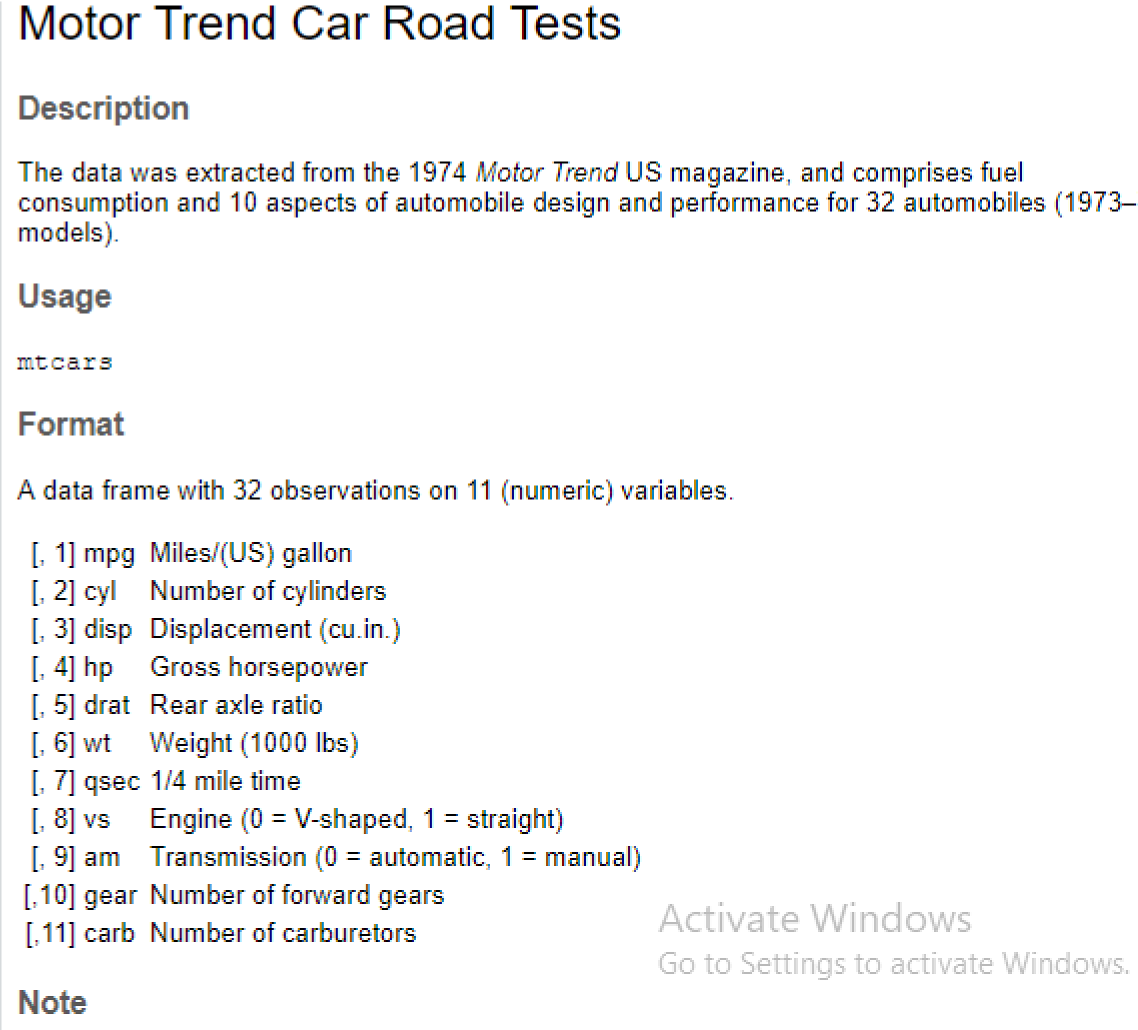
Information About the Data Set

- You can use the question mark **(?)** to get information about the **mtcars** data set.

Example:



```
> ?mtcars
> |
```



Motor Trend Car Road Tests

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

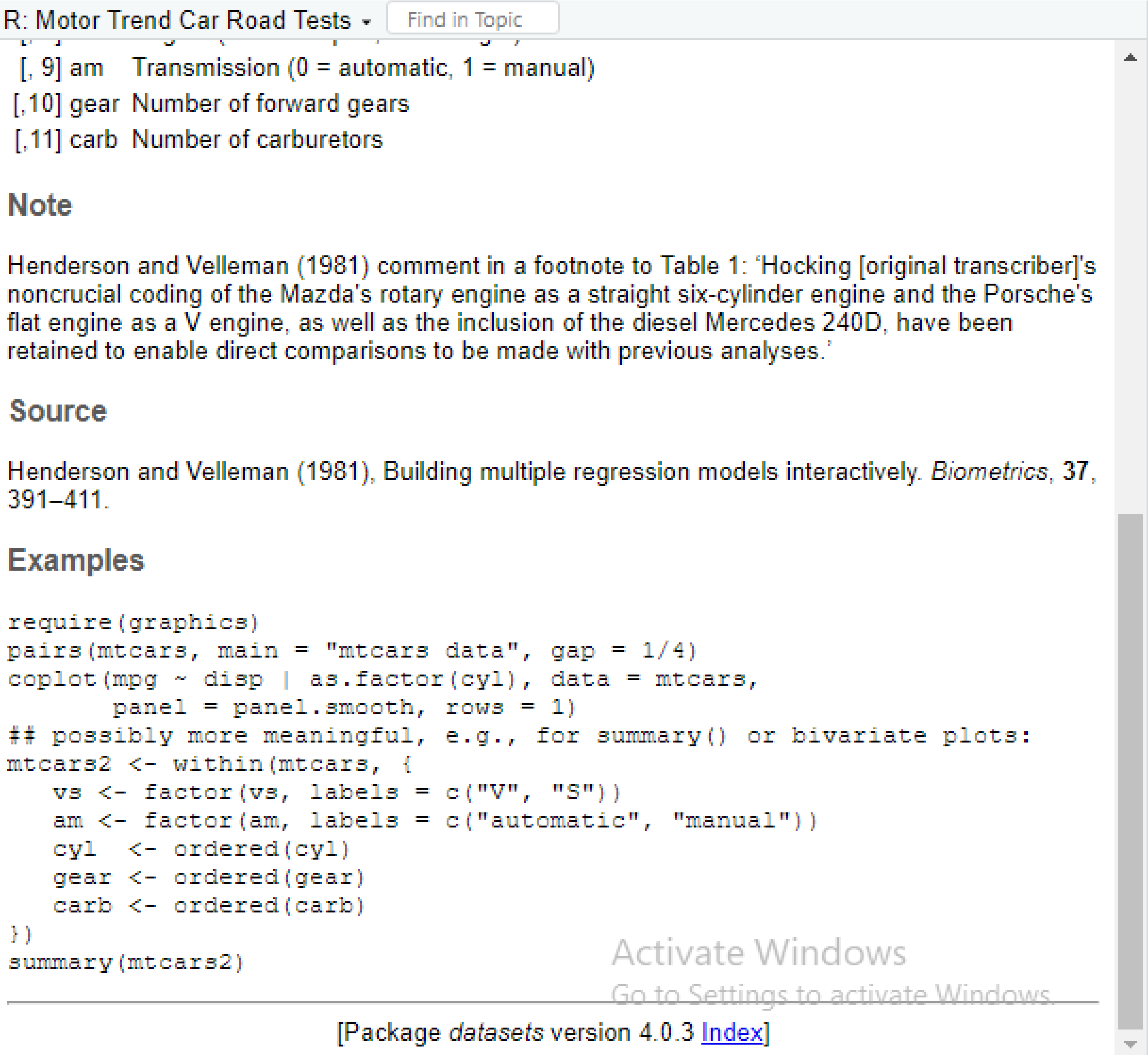
```
mtcars
```

Format

A data frame with 32 observations on 11 (numeric) variables.

[, 1]	mpg	Miles/(US) gallon
[, 2]	cyl	Number of cylinders
[, 3]	disp	Displacement (cu.in.)
[, 4]	hp	Gross horsepower
[, 5]	drat	Rear axle ratio
[, 6]	wt	Weight (1000 lbs)
[, 7]	qsec	1/4 mile time
[, 8]	vs	Engine (0 = V-shaped, 1 = straight)
[, 9]	am	Transmission (0 = automatic, 1 = manual)
[,10]	gear	Number of forward gears
[,11]	carb	Number of carburetors

Note



[, 9] am Transmission (0 = automatic, 1 = manual)
[,10] gear Number of forward gears
[,11] carb Number of carburetors

Note

Henderson and Velleman (1981) comment in a footnote to Table 1: ‘Hocking [original transcriber]'s noncrucial coding of the Mazda's rotary engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the inclusion of the diesel Mercedes 240D, have been retained to enable direct comparisons to be made with previous analyses.’

Source

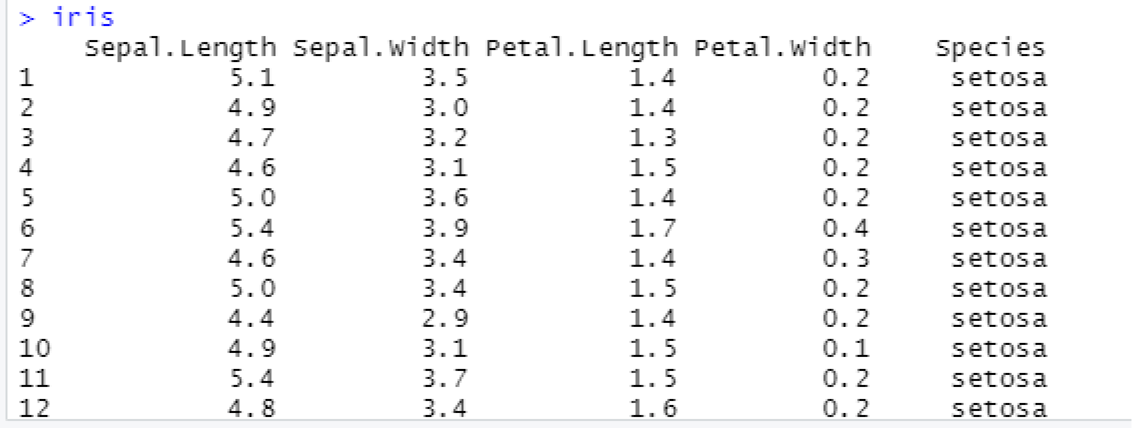
Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Examples

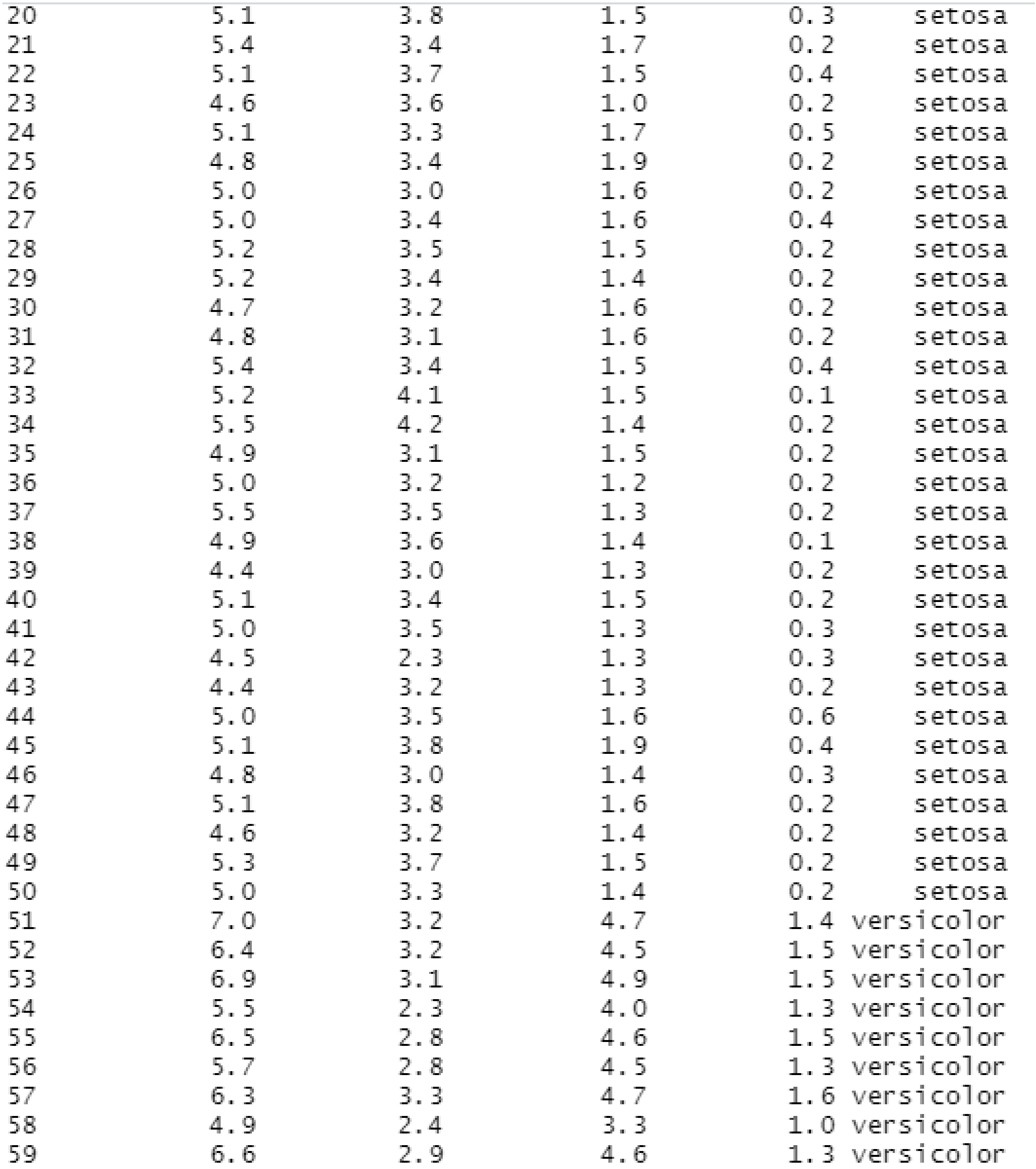
```
require(graphics)
pairs(mtcars, main = "mtcars data", gap = 1/4)
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
## possibly more meaningful, e.g., for summary() or bivariate plots:
mtcars2 <- within(mtcars, {
   vs <- factor(vs, labels = c("V", "S"))
   am <- factor(am, labels = c("automatic", "manual"))
   cyl  <- ordered(cyl)
   gear <- ordered(gear)
   carb <- ordered(carb)
})
summary(mtcars2)
```

[Package *datasets* version 4.0.3 Index]

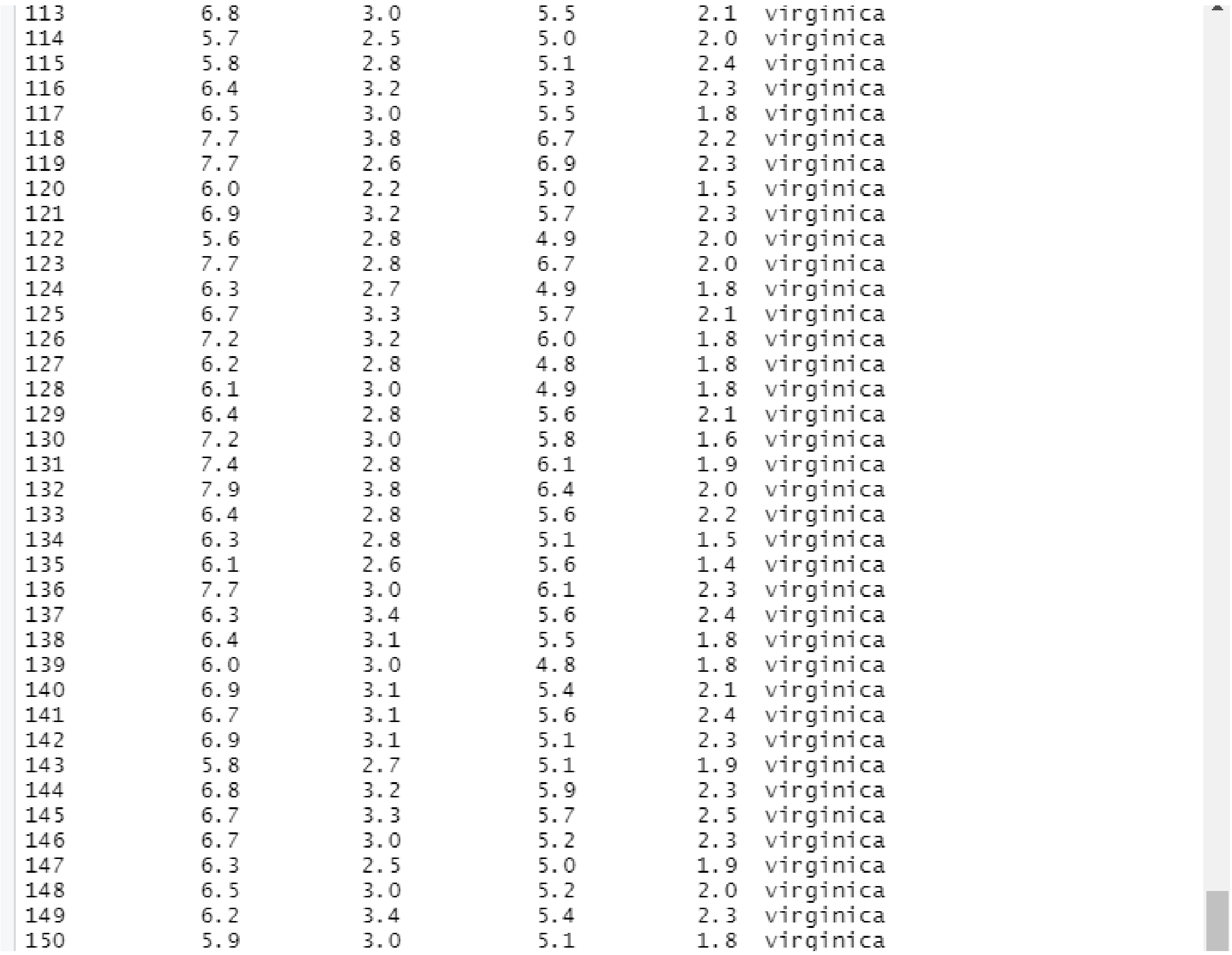
Example : iris data set



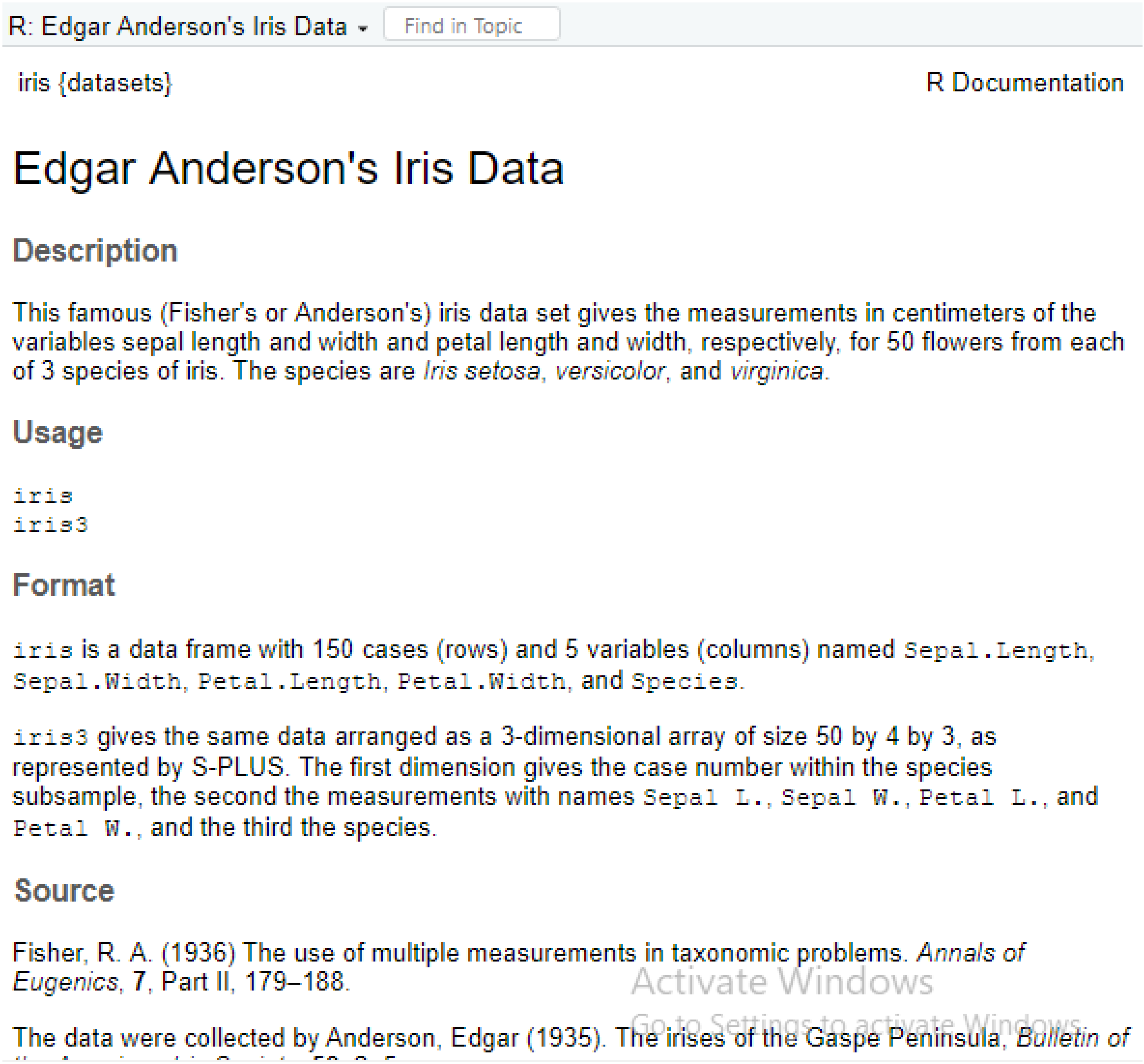
```
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
6            5.4         3.9          1.7         0.4     setosa
7            4.6         3.4          1.4         0.3     setosa
8            5.0         3.4          1.5         0.2     setosa
9            4.4         2.9          1.4         0.2     setosa
10           4.9         3.1          1.5         0.1     setosa
11           5.4         3.7          1.5         0.2     setosa
12           4.8         3.4          1.6         0.2     setosa
```



```
20          5.1          3.8          1.5          0.3      setosa
21          5.4          3.4          1.7          0.2      setosa
22          5.1          3.7          1.5          0.4      setosa
23          4.6          3.6          1.0          0.2      setosa
24          5.1          3.3          1.7          0.5      setosa
25          4.8          3.4          1.9          0.2      setosa
26          5.0          3.0          1.6          0.2      setosa
27          5.0          3.4          1.6          0.4      setosa
28          5.2          3.5          1.5          0.2      setosa
29          5.2          3.4          1.4          0.2      setosa
30          4.7          3.2          1.6          0.2      setosa
31          4.8          3.1          1.6          0.2      setosa
32          5.4          3.4          1.5          0.4      setosa
33          5.2          4.1          1.5          0.1      setosa
34          5.5          4.2          1.4          0.2      setosa
35          4.9          3.1          1.5          0.2      setosa
36          5.0          3.2          1.2          0.2      setosa
37          5.5          3.5          1.3          0.2      setosa
38          4.9          3.6          1.4          0.1      setosa
39          4.4          3.0          1.3          0.2      setosa
40          5.1          3.4          1.5          0.2      setosa
41          5.0          3.5          1.3          0.3      setosa
42          4.5          2.3          1.3          0.3      setosa
43          4.4          3.2          1.3          0.2      setosa
44          5.0          3.5          1.6          0.6      setosa
45          5.1          3.8          1.9          0.4      setosa
46          4.8          3.0          1.4          0.3      setosa
47          5.1          3.8          1.6          0.2      setosa
48          4.6          3.2          1.4          0.2      setosa
49          5.3          3.7          1.5          0.2      setosa
50          5.0          3.3          1.4          0.2      setosa
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
56          5.7          2.8          4.5          1.3  versicolor
57          6.3          3.3          4.7          1.6  versicolor
58          4.9          2.4          3.3          1.0  versicolor
59          6.6          2.9          4.6          1.3  versicolor
```



113	6.8	3.0	5.5	2.1	virginica
114	5.7	2.5	5.0	2.0	virginica
115	5.8	2.8	5.1	2.4	virginica
116	6.4	3.2	5.3	2.3	virginica
117	6.5	3.0	5.5	1.8	virginica
118	7.7	3.8	6.7	2.2	virginica
119	7.7	2.6	6.9	2.3	virginica
120	6.0	2.2	5.0	1.5	virginica
121	6.9	3.2	5.7	2.3	virginica
122	5.6	2.8	4.9	2.0	virginica
123	7.7	2.8	6.7	2.0	virginica
124	6.3	2.7	4.9	1.8	virginica
125	6.7	3.3	5.7	2.1	virginica
126	7.2	3.2	6.0	1.8	virginica
127	6.2	2.8	4.8	1.8	virginica
128	6.1	3.0	4.9	1.8	virginica
129	6.4	2.8	5.6	2.1	virginica
130	7.2	3.0	5.8	1.6	virginica
131	7.4	2.8	6.1	1.9	virginica
132	7.9	3.8	6.4	2.0	virginica
133	6.4	2.8	5.6	2.2	virginica
134	6.3	2.8	5.1	1.5	virginica
135	6.1	2.6	5.6	1.4	virginica
136	7.7	3.0	6.1	2.3	virginica
137	6.3	3.4	5.6	2.4	virginica
138	6.4	3.1	5.5	1.8	virginica
139	6.0	3.0	4.8	1.8	virginica
140	6.9	3.1	5.4	2.1	virginica
141	6.7	3.1	5.6	2.4	virginica
142	6.9	3.1	5.1	2.3	virginica
143	5.8	2.7	5.1	1.9	virginica
144	6.8	3.2	5.9	2.3	virginica
145	6.7	3.3	5.7	2.5	virginica
146	6.7	3.0	5.2	2.3	virginica
147	6.3	2.5	5.0	1.9	virginica
148	6.5	3.0	5.2	2.0	virginica
149	6.2	3.4	5.4	2.3	virginica
150	5.9	3.0	5.1	1.8	virginica



R: Edgar Anderson's Iris Data ▾ Find in Topic

iris {datasets} R Documentation

Edgar Anderson's Iris Data

Description

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters 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*.

Usage

```
iris
iris3
```

Format

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

`iris3` gives the same data arranged as a 3-dimensional array of size 50 by 4 by 3, as represented by S-PLUS. The first dimension gives the case number within the species subsample, the second the measurements with names `Sepal L.`, `Sepal W.`, `Petal L.`, and `Petal W.`, and the third the species.

Source

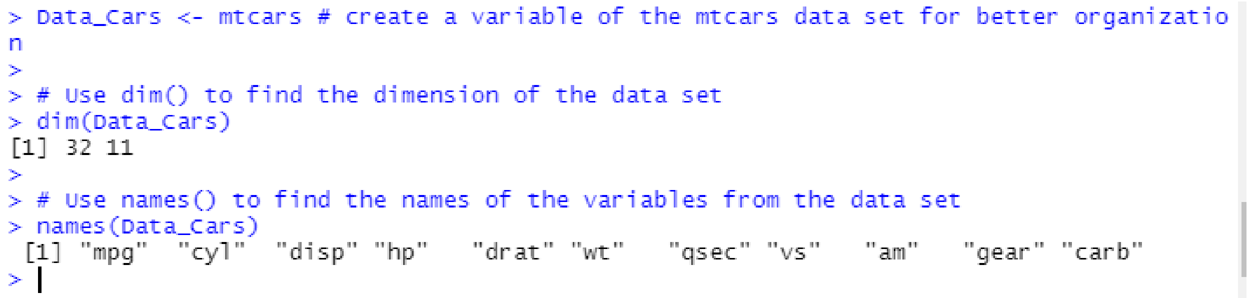
Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188.

The data were collected by Anderson, Edgar (1935). The irises of the Gaspe Peninsula, *Bulletin of*

Get Information

- Use the **dim()** function to find the dimensions of the data set, and the **names()** function to view the names of the variables:

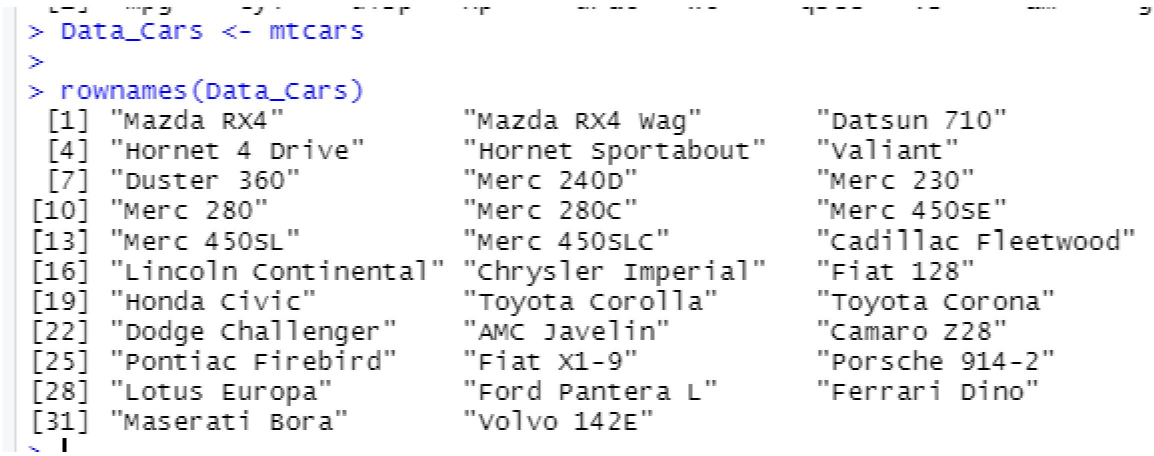
Example:



```
> Data_Cars <- mtcars # create a variable of the mtcars data set for better organizatio
n
>
> # Use dim() to find the dimension of the data set
> dim(Data_Cars)
[1] 32 11
>
> # Use names() to find the names of the variables from the data set
> names(Data_Cars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"
> |
```

- Use the **rownames()** function to get the name of each row in the first column, which is the name of each car.

Ex:

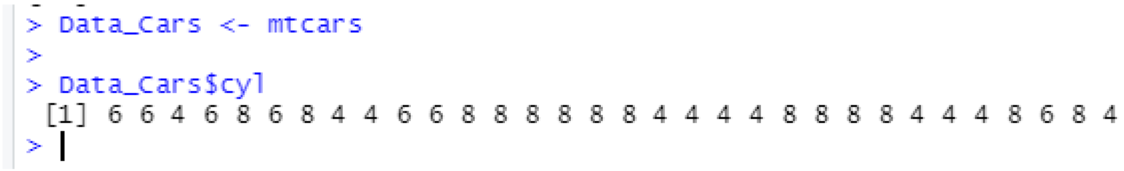


```
> Data_Cars <- mtcars
>
> rownames(Data_Cars)
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"         
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"            
 [7] "Duster 360"          "Merc 240D"           "Merc 230"           
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"         
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood" 
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"           
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"      
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"         
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"      
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"       
[31] "Maserati Bora"       "Volvo 142E"         
```

Print Variable Values

- If you want to print all values that belong to a variable, access the data frame by using the **$** sign, and the name of the variable (for example **cyl** (cylinders)):

Ex:

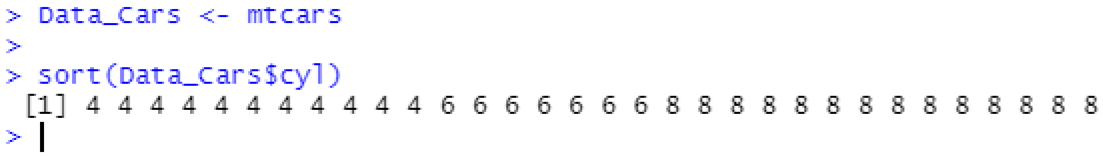


```
> Data_Cars <- mtcars
>
> Data_Cars$cyl
 [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
> 
```

Sort Variable Values

- To sort the values, use the **sort()** function:

Example:

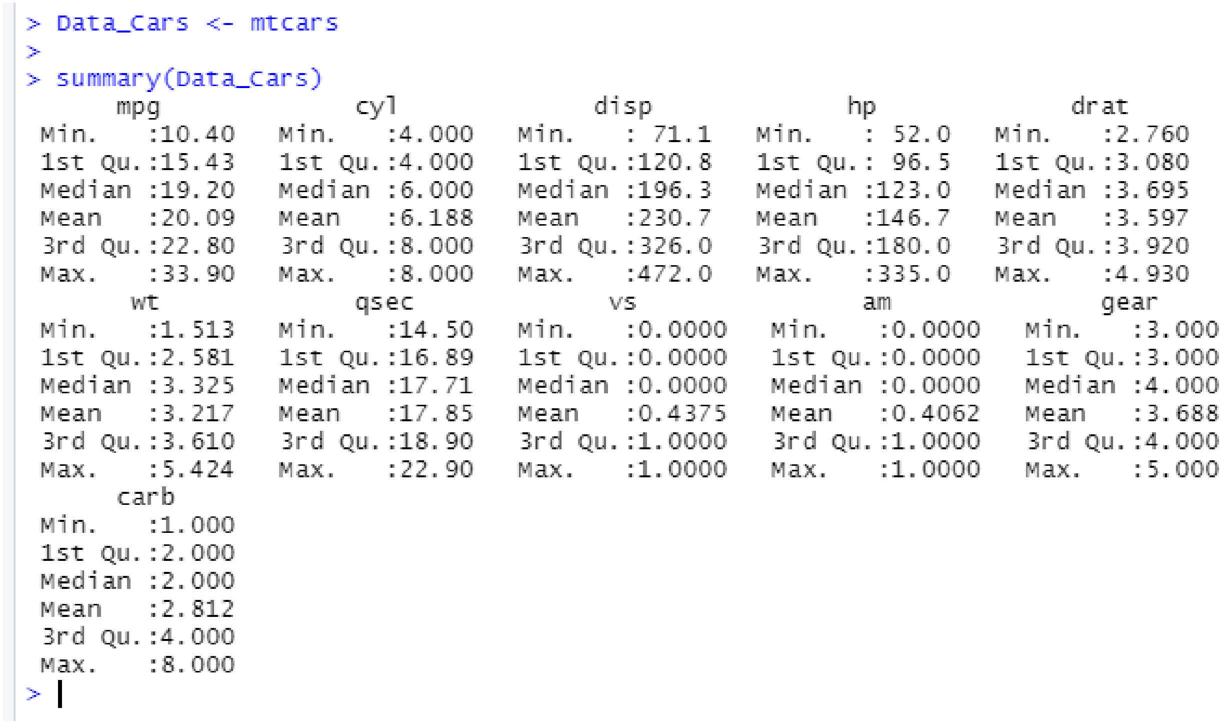


```
> Data_Cars <- mtcars
>
> sort(Data_Cars$cyl)
 [1] 4 4 4 4 4 4 4 4 4 4 4 6 6 6 6 6 6 6 8 8 8 8 8 8 8 8 8 8 8 8 8 8
> 
```

Analyzing the Data

- Now that we have some information about the data set, we can start to analyze it with some statistical numbers.
- **summary()** function to get a statistical summary of the data.

Ex:



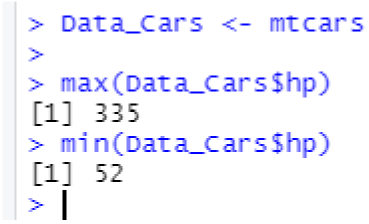
```
> Data_Cars <- mtcars
>
> summary(Data_Cars)
      mpg             cyl             disp             hp             drat      
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0   Min.   :2.760  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5   1st Qu.:3.080  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0   Median :3.695  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7   Mean   :3.597  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0   3rd Qu.:3.920  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0   Max.   :4.930  
       wt             qsec             vs               am              gear      
 Min.   :1.513   Min.   :14.50   Min.   :0.0000   Min.   :0.0000   Min.   :3.000  
 1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:3.000  
 Median :3.325   Median :17.71   Median :0.0000   Median :0.0000   Median :4.000  
 Mean   :3.217   Mean   :17.85   Mean   :0.4375   Mean   :0.4062   Mean   :3.688  
 3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:4.000  
 Max.   :5.424   Max.   :22.90   Max.   :1.0000   Max.   :1.0000   Max.   :5.000  
      carb      
 Min.   :1.000  
 1st Qu.:2.000  
 Median :2.000  
 Mean   :2.812  
 3rd Qu.:4.000  
 Max.   :8.000  
>
```

The **summary()** function returns six statistical numbers for each variable:

- Min
- First quantile (percentile)
- Median
- Mean
- Third quantile (percentile)
- Max

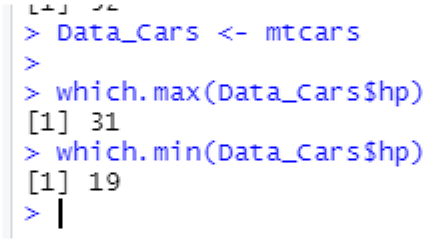
R max and min function

Example: Find the largest and smallest value of the variable **hp (horsepower).**



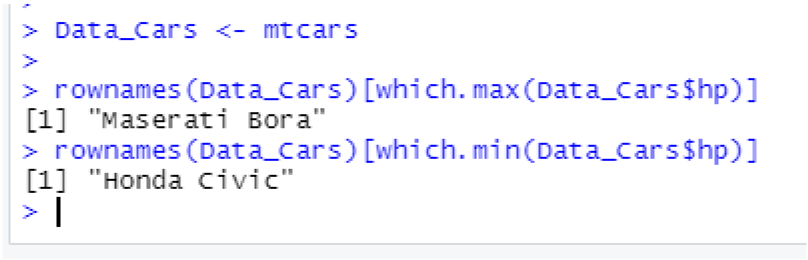
```
> Data_Cars <- mtcars
>
> max(Data_Cars$hp)
[1] 335
> min(Data_Cars$hp)
[1] 52
>
```

Example: To find the index position of the max and min value in the table use the **which.max()** and **which.min()** functions.



```
> Data_Cars <- mtcars
>
> which.max(Data_Cars$hp)
[1] 31
> which.min(Data_Cars$hp)
[1] 19
> 
```

Example: To get the name of the car with the largest and smallest horsepower combine **which.max()** and **which.min()** with the **rownames()** function



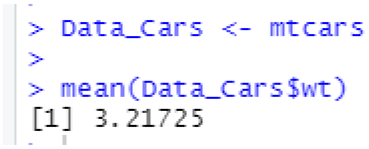
```
> Data_Cars <- mtcars
>
> rownames(Data_Cars)[which.max(Data_Cars$hp)]
[1] "Maserati Bora"
> rownames(Data_Cars)[which.min(Data_Cars$hp)]
[1] "Honda Civic"
> 
```

Mean, Median, and Mode

In statistics, there are often three values that interests us:

- Mean - The average value
- Median - The middle value
- Mode - The most common value

Ex:Find the average(wt) of cars:



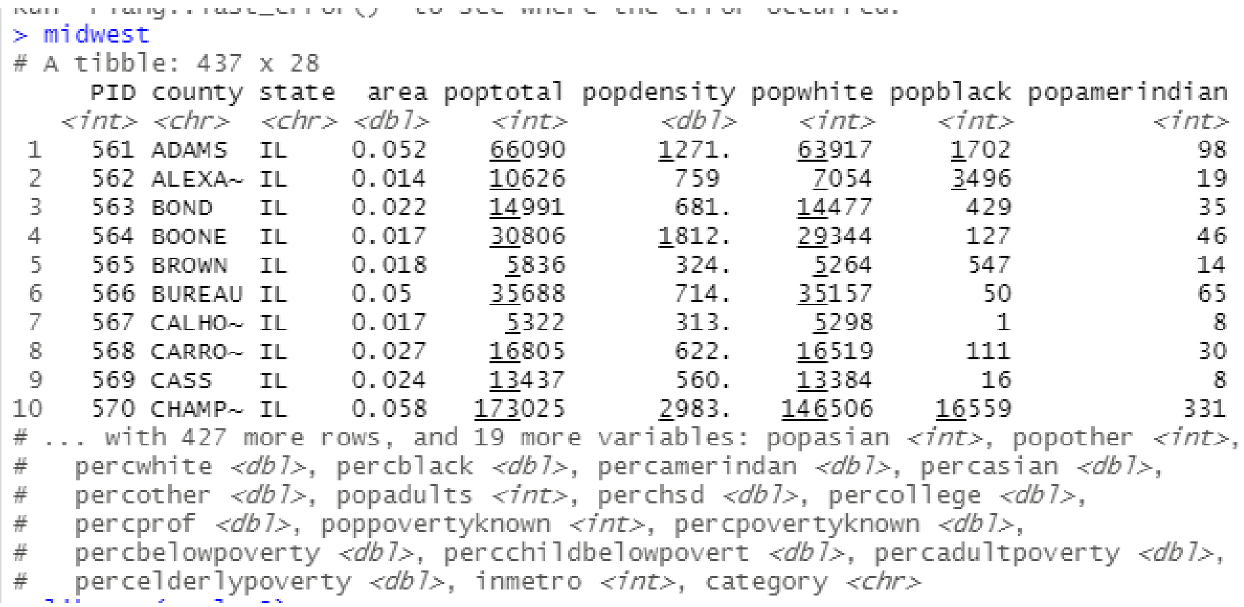
```
> Data_Cars <- mtcars
>
> mean(Data_Cars$wt)
[1] 3.21725
```

Median

- The median value is the value in the middle, after you have sorted all the values.
- If we take a look at the values of the **wt** variable (from the **mtcars** data set), we will see that there are two numbers in the middle:

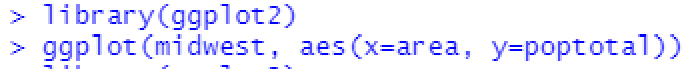
ggplot()

Example:

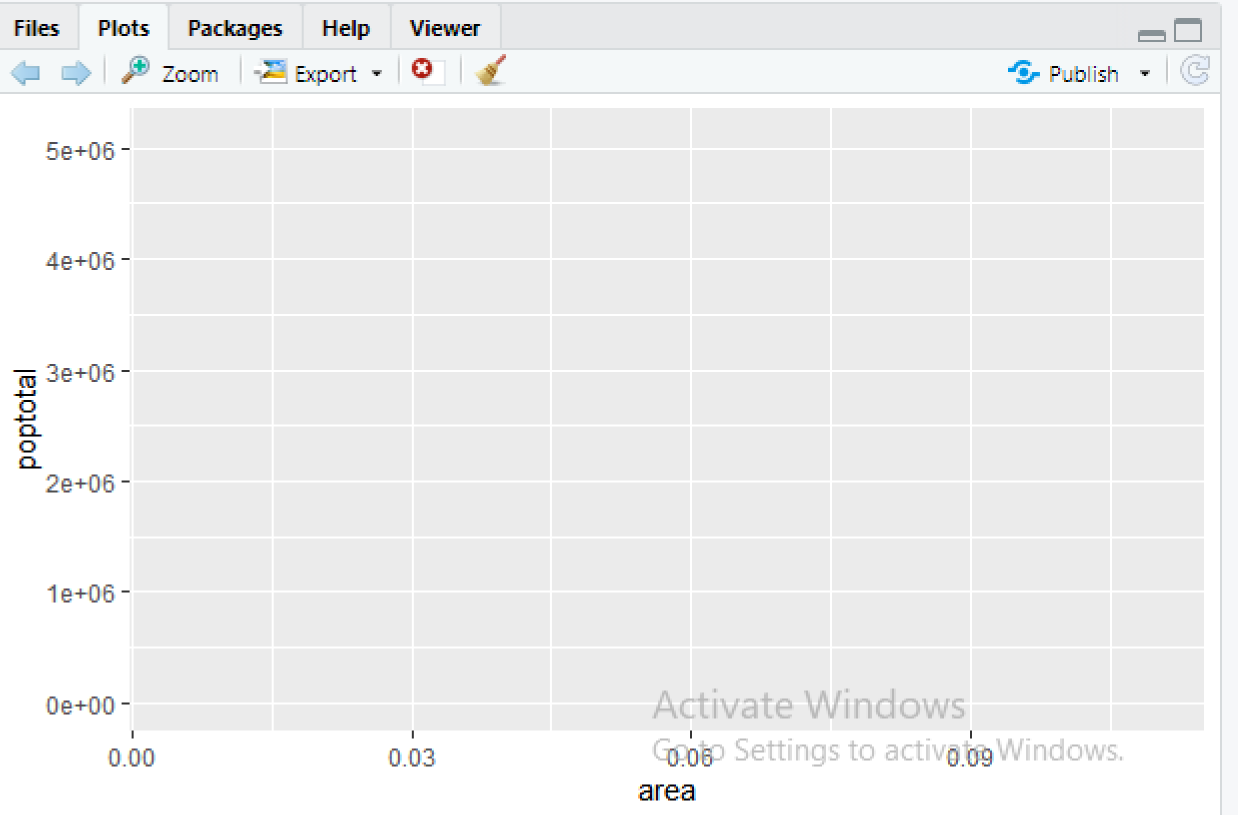


```
Run `rlang::last_error()` to see where the error occurred.
> midwest
# A tibble: 437 x 28
     PID county state  area poptotal popdensity popwhite popblack popamerindian
   <int> <chr>  <chr> <dbl>    <int>      <dbl>    <int>    <int>         <int>
 1   561 ADAMS  IL    0.052    66090      1271.    63917     1702            98
 2   562 ALEXA~ IL    0.014    10626       759      7054     3496            19
 3   563 BOND   IL    0.022    14991       681.    14477      429            35
 4   564 BOONE  IL    0.017    30806      1812.    29344      127            46
 5   565 BROWN  IL    0.018     5836       324.     5264      547            14
 6   566 BUREAU IL    0.05     35688       714.    35157       50            65
 7   567 CALHO~ IL    0.017     5322       313.     5298        1             8
 8   568 CARRO~ IL    0.027    16805       622.    16519      111            30
 9   569 CASS   IL    0.024    13437       560.    13384       16             8
10   570 CHAMP~ IL    0.058   173025      2983.   146506    16559           331
# ... with 427 more rows, and 19 more variables: popasian <int>, popother <int>,
#   percwhite <dbl>, percblack <dbl>, percamerindan <dbl>, percasian <dbl>,
#   percother <dbl>, popadults <int>, perchsd <dbl>, percollege <dbl>,
#   percprof <dbl>, poppovertyknown <int>, percpovertyknown <dbl>,
#   percbelowpoverty <dbl>, percchildbelowpovert <dbl>, percadultpoverty <dbl>,
#   percelderlypoverty <dbl>, inmetro <int>, category <chr>
```

Example:



```
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=poptotal))
```



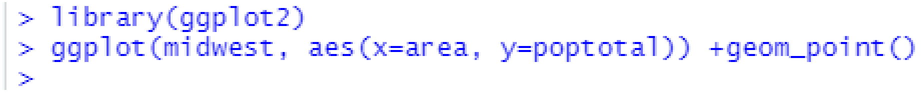
Note:

1. A blank ggplot is drawn. Even though the x and y are specified, there are no points or lines in it. This is because ggplot doesn’t assume that you meant a scatterplot or a line chart to be drawn.
2. Also note that aes() function is used to specify the X and Y axes. That’s because, any information that is part of the source data frame has to be specified inside the aes() function.

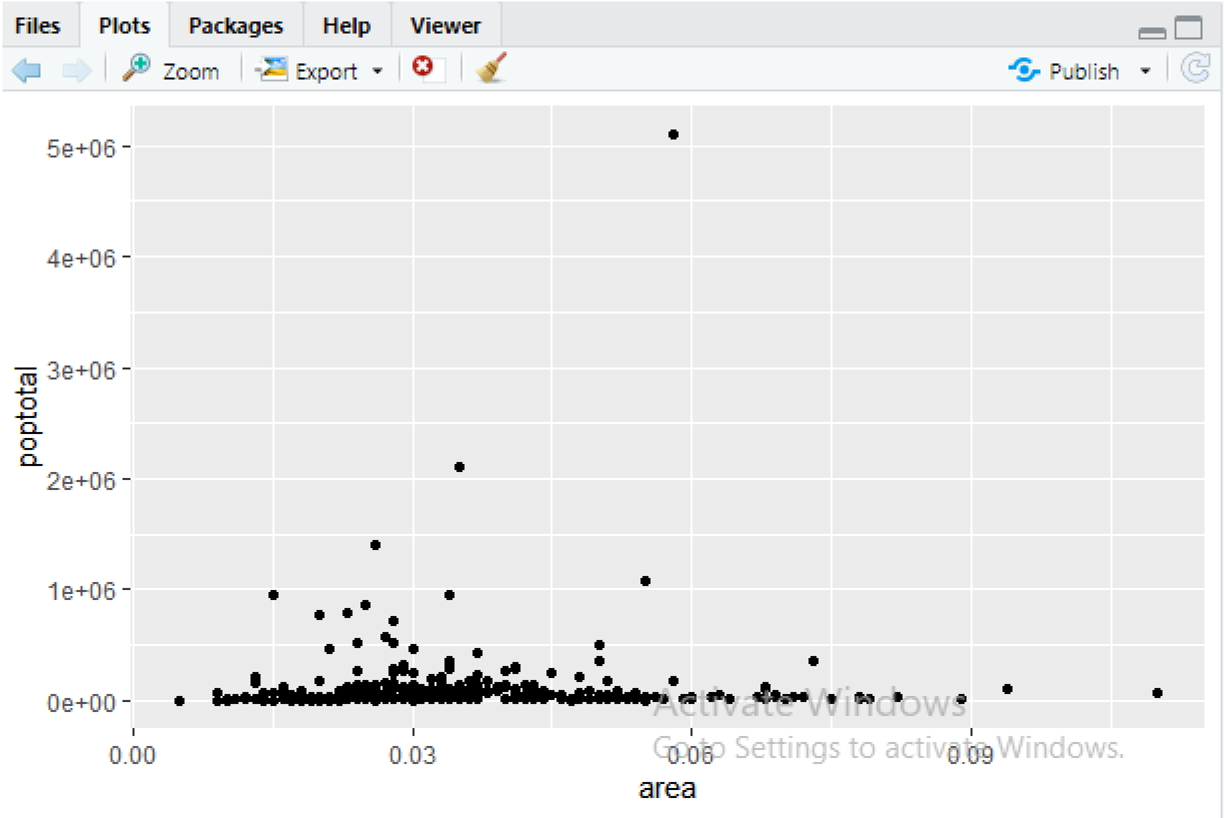
How to Make a Simple Scatterplot

Let’s make a scatterplot on top of the blank ggplot by adding points using a geom layer called geom_point

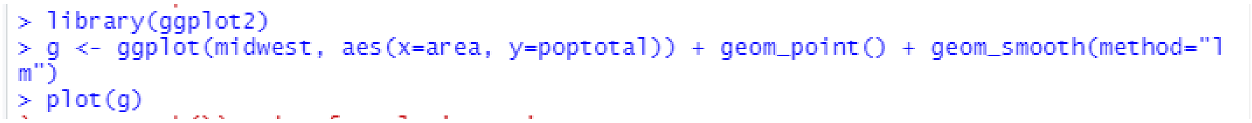
Example:



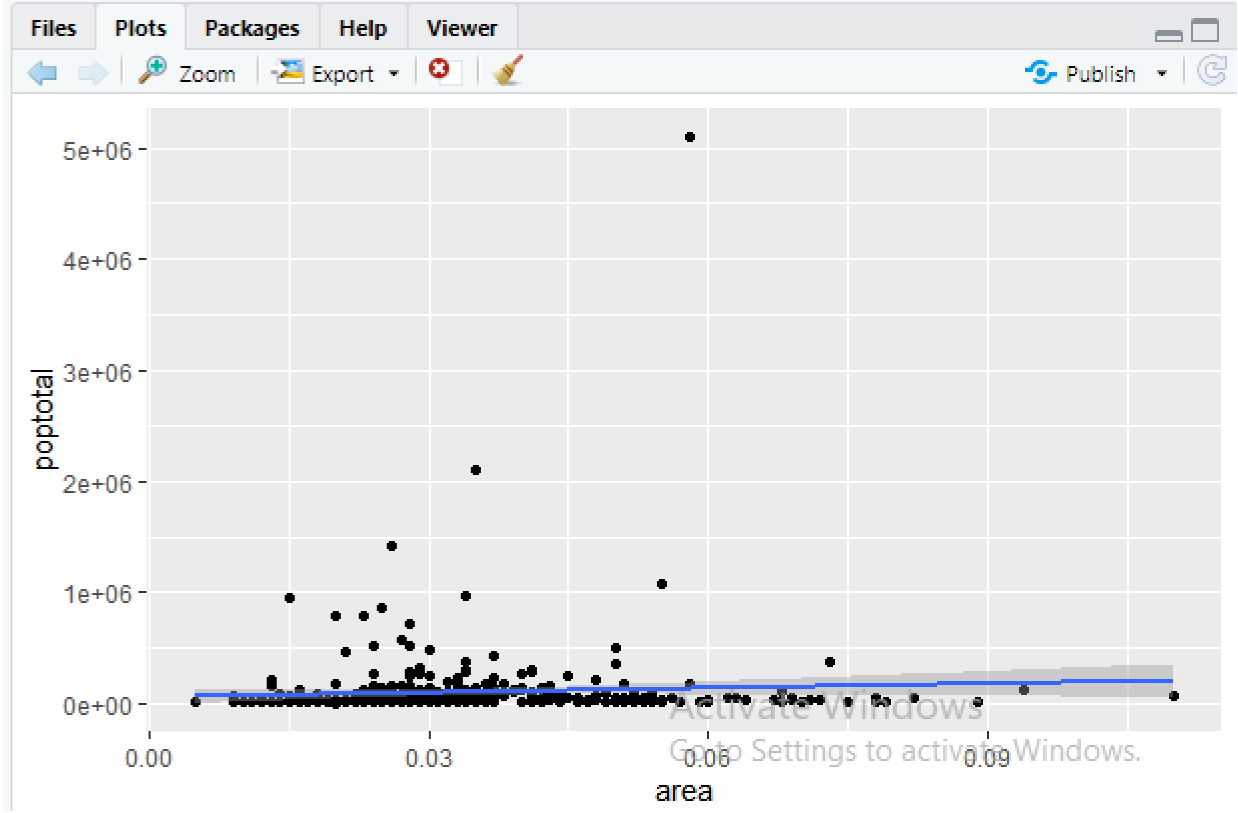
```
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=poptotal)) +geom_point()
>
```



Ex.



```
> library(ggplot2)
> g <- ggplot(midwest, aes(x=area, y=poptotal)) + geom_point() + geom_smooth(method="l
m")
> plot(g)
```



Adjusting the X and Y axis limits

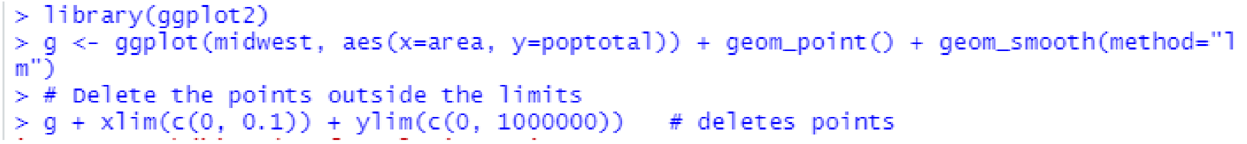
The X and Y axis limits can be controlled in 2 ways.

Method 1: By deleting the points outside the range

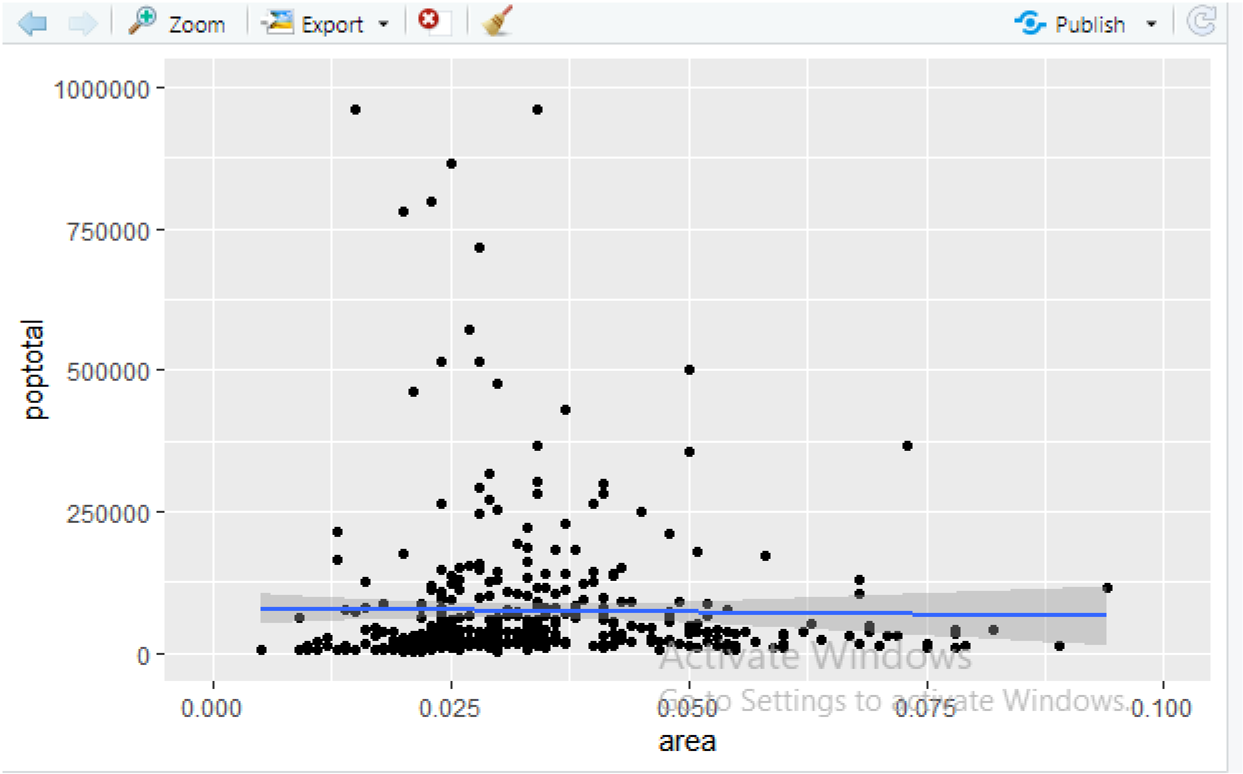
This will change the lines of best fit or smoothing lines as compared to the original data.

This can be done by xlim() and ylim(). You can pass a numeric vector of length 2 (with max and min values) or just the max and min values itself.

Ex:

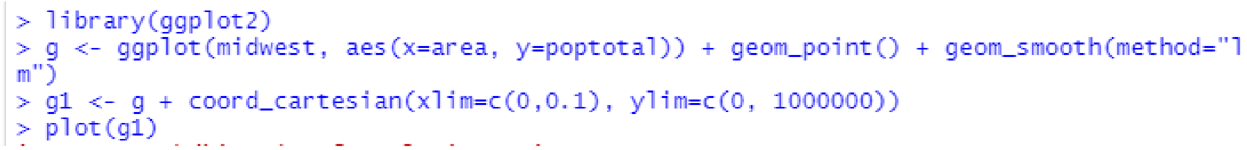


```
> library(ggplot2)
> g <- ggplot(midwest, aes(x=area, y=poptotal)) + geom_point() + geom_smooth(method="l
m")
> # Delete the points outside the limits
> g + xlim(c(0, 0.1)) + ylim(c(0, 1000000))   # deletes points
```

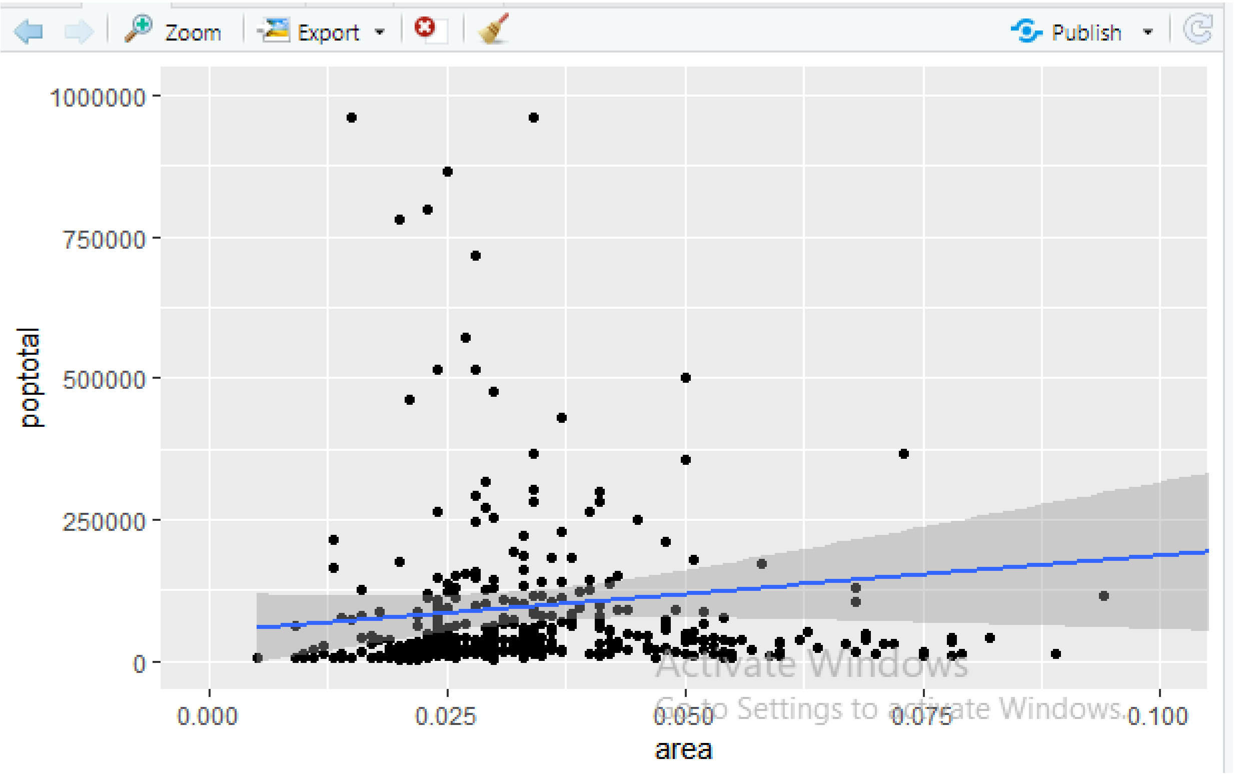


In this case, the chart was not built from scratch but rather was built on top of g. This is because the previous plot was stored as g, a ggplot object, which when called will reproduce the original plot.

Ex:



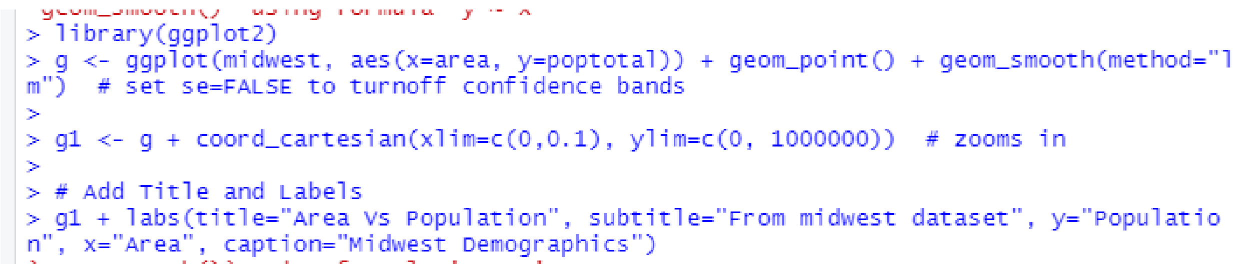
```
> library(ggplot2)
> g <- ggplot(midwest, aes(x=area, y=poptotal)) + geom_point() + geom_smooth(method="l
m")
> g1 <- g + coord_cartesian(xlim=c(0,0.1), ylim=c(0, 1000000))
> plot(g1)
```



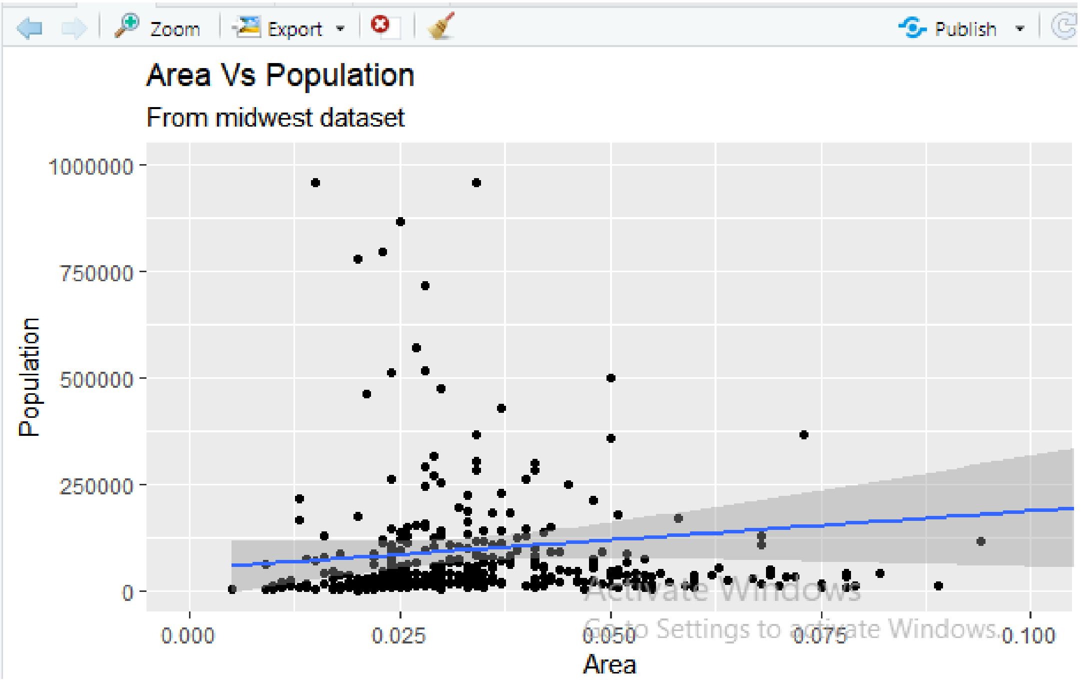
How to Change the Title and Axis Labels

I have stored this as g1. Let's add the plot title and labels for X and Y axis. This can be done in one go using the labs() function with title, x and y arguments. Another option is to use the ggtitle(), xlab() and ylab().

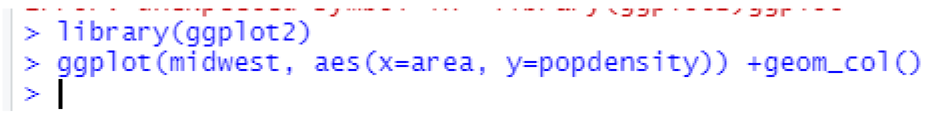
Ex:



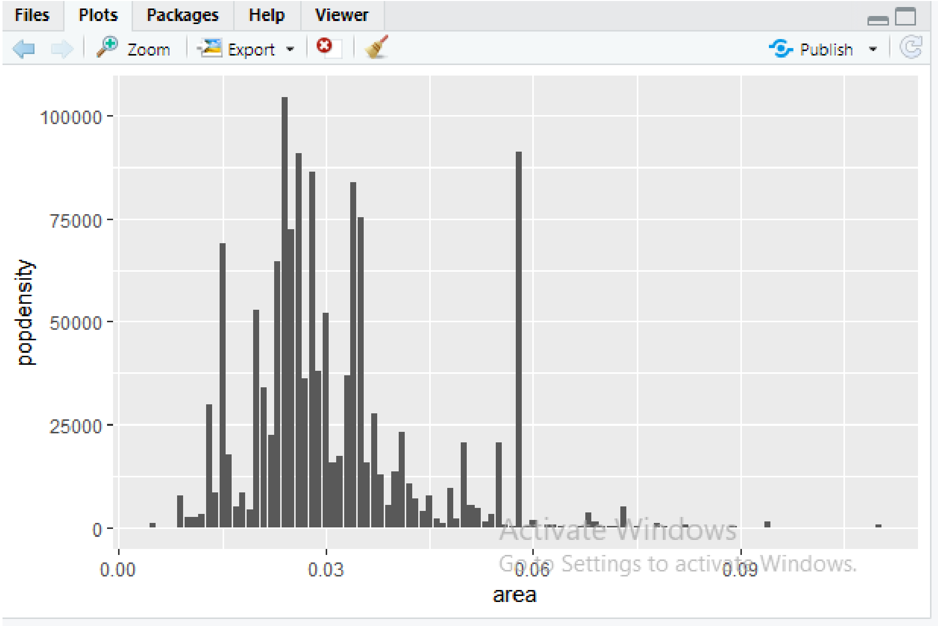
```
`geom_smooth()` using formula 'y ~ x'
> library(ggplot2)
> g <- ggplot(midwest, aes(x=area, y=poptotal)) + geom_point() + geom_smooth(method="l
m")  # set se=FALSE to turnoff confidence bands
>
> g1 <- g + coord_cartesian(xlim=c(0,0.1), ylim=c(0, 1000000))  # zooms in
>
> # Add Title and Labels
> g1 + labs(title="Area Vs Population", subtitle="From midwest dataset", y="Populatio
n", x="Area", caption="Midwest Demographics")
```



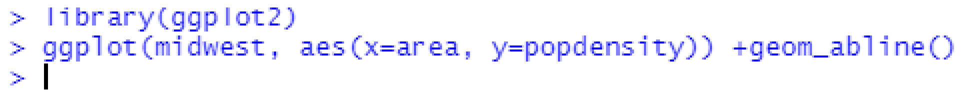
Ex:



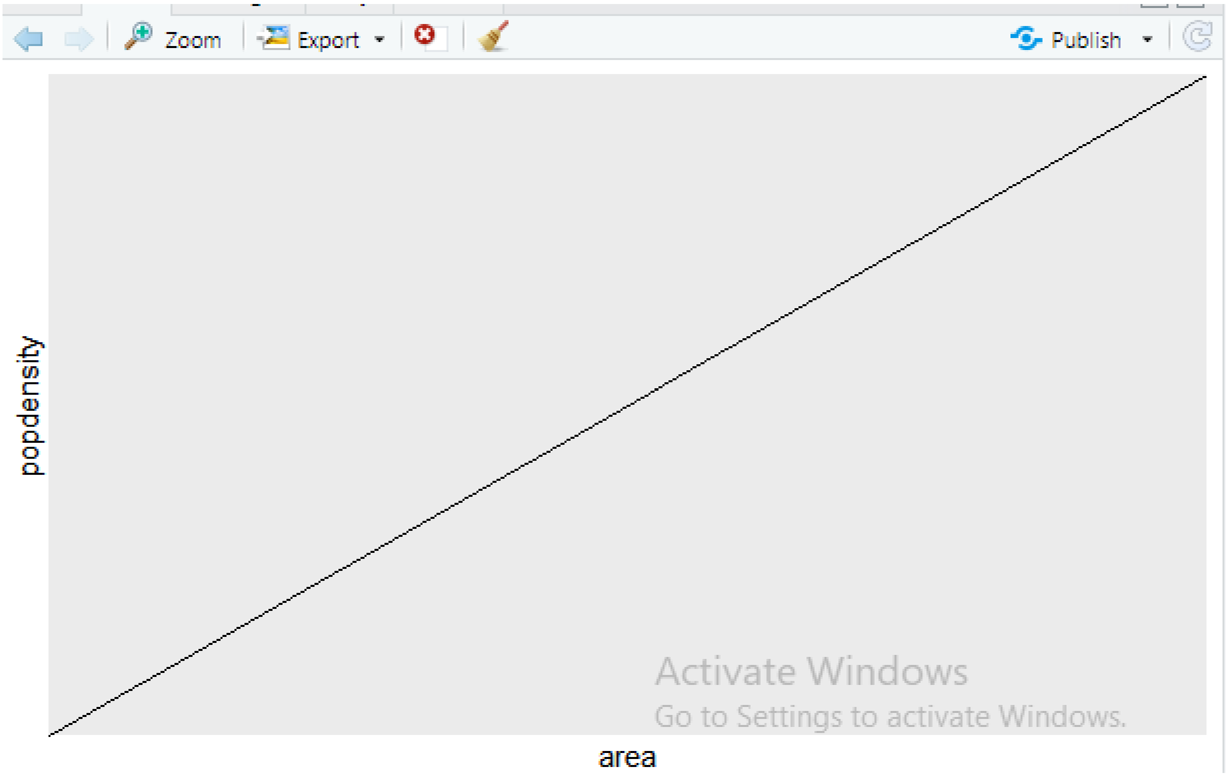
```
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=popdensity)) +geom_col()
> 
```



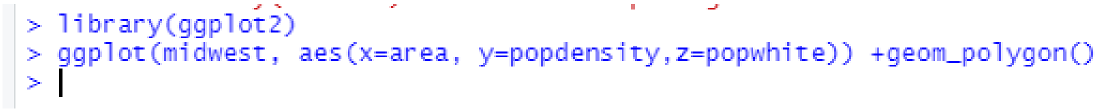
Ex:



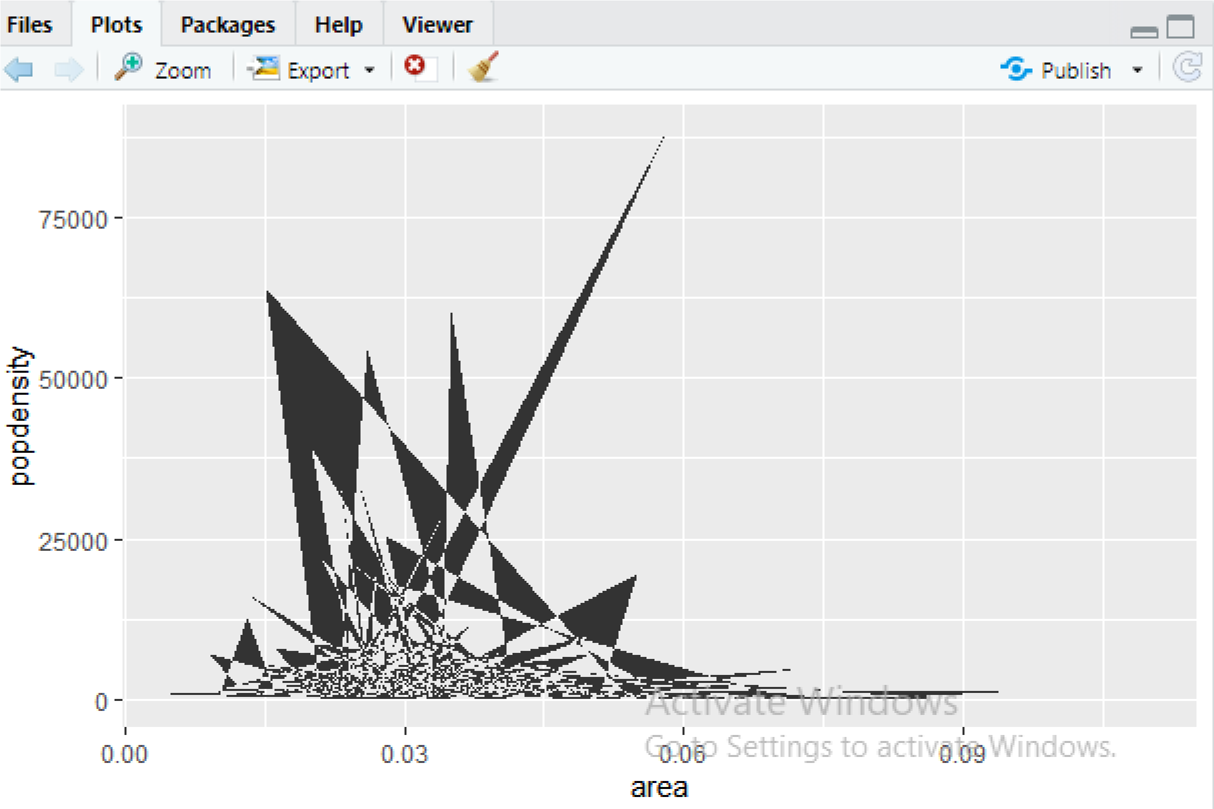
```
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=popdensity)) +geom_abline()
> 
```



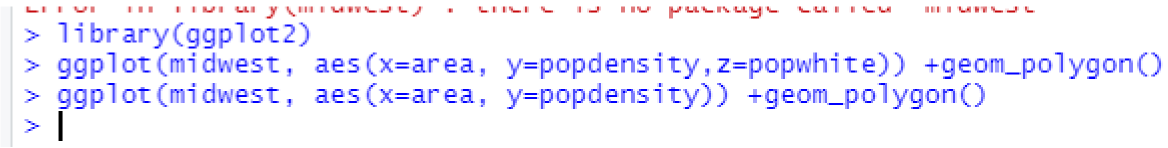
Ex:



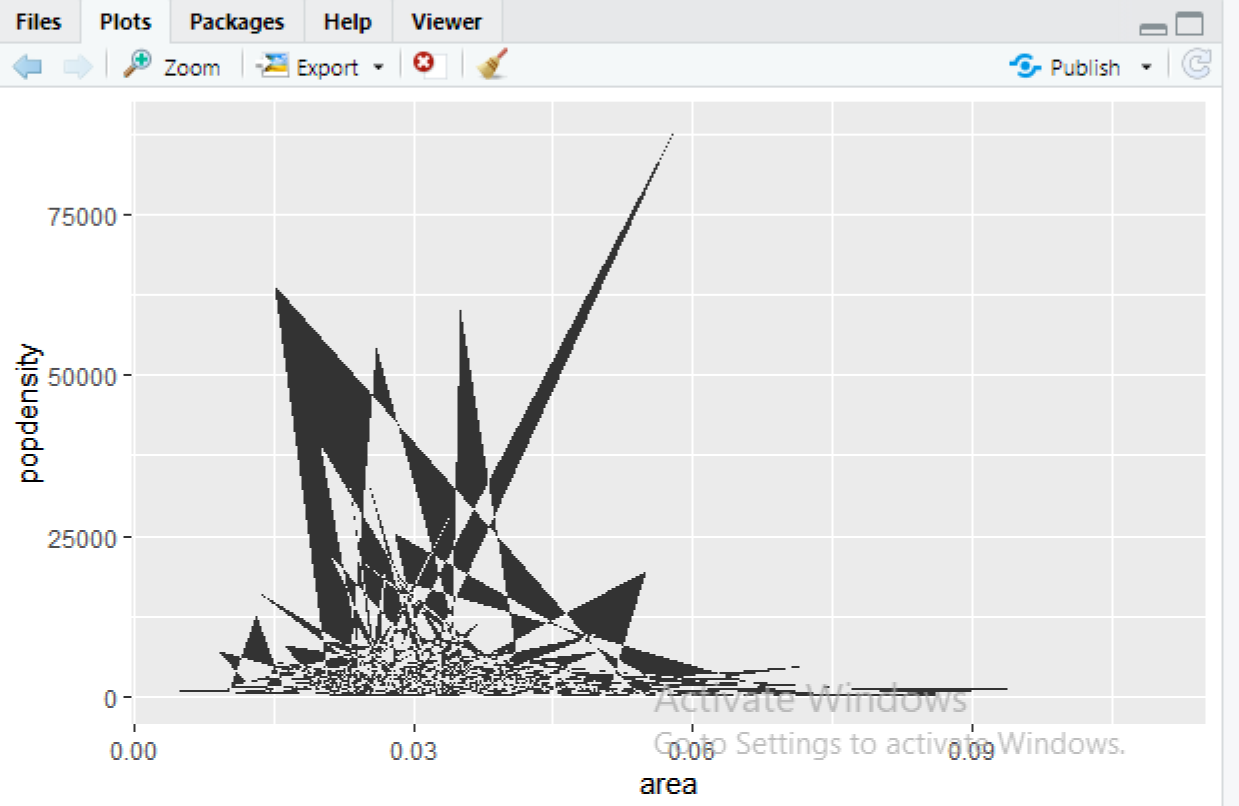
```
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=popdensity,z=popwhite)) +geom_polygon()
> 
```



Ex:



```
Error in library(midwest) : there is no package called ‘midwest’
> library(ggplot2)
> ggplot(midwest, aes(x=area, y=popdensity,z=popwhite)) +geom_polygon()
> ggplot(midwest, aes(x=area, y=popdensity)) +geom_polygon()
> 
```


Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
popdensity
75000
50000
25000
0
0.00
0.03
0.06
0.09
area
Activate Windows
Go to Settings to activate Windows.

Reading and Writing CSV Files

R CSV Files

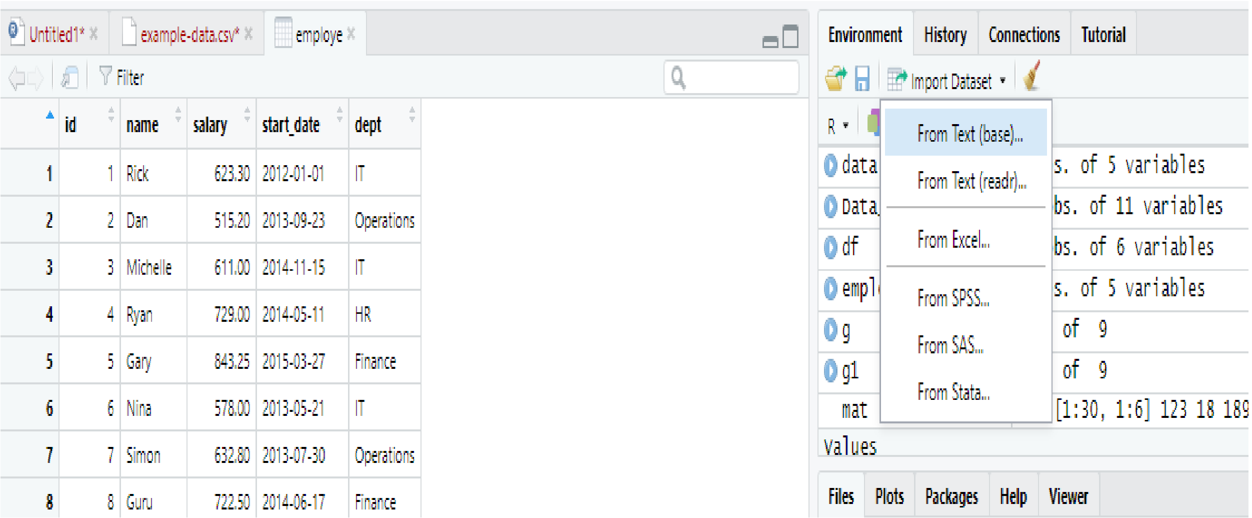
- A **Comma-Separated Values (CSV) file** is a plain text file which contains a list of data.
- These files are often used for the exchange of data between different applications.
- For example, databases and contact managers mostly support CSV files.
- These files can sometimes be called **character-separated values** or **comma-delimited files**.
- They often use the comma character to separate data, but sometimes use other characters such as semicolons. The idea is that we can export the complex data from one application to a CSV file, and then importing the data in that CSV file to another application.

Input as CSV File

- The csv file is a text file in which the values in the columns are separated by a comma.
- Let's consider the following data present in the file named input.csv.

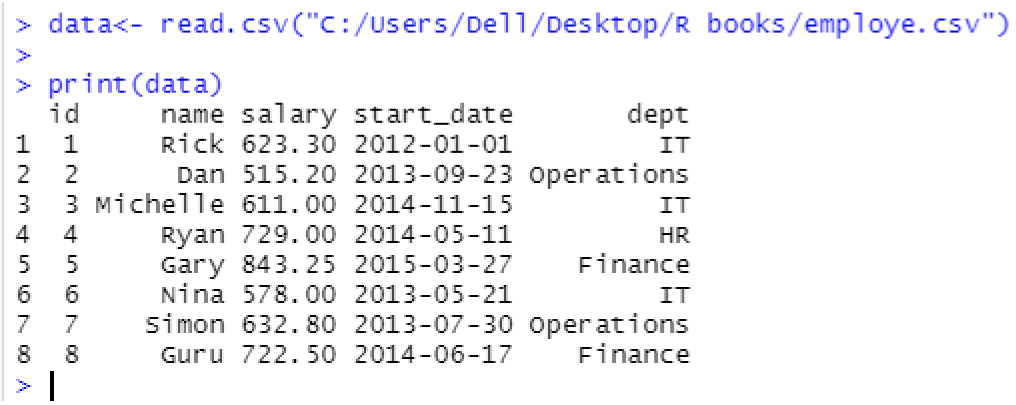
Reading a CSV File

- Following is a simple example of read.csv() function to read a CSV file available in your current working directory −
- OR just give the link where you data is located.
- You can import data set by using import function.



	id	name	salary	start_date	dept
1	1	Rick	623.30	2012-01-01	IT
2	2	Dan	515.20	2013-09-23	Operations
3	3	Michelle	611.00	2014-11-15	IT
4	4	Ryan	729.00	2014-05-11	HR
5	5	Gary	843.25	2015-03-27	Finance
6	6	Nina	578.00	2013-05-21	IT
7	7	Simon	632.80	2013-07-30	Operations
8	8	Guru	722.50	2014-06-17	Finance

```
data    s. of 5 variables
Data    bs. of 11 variables
df      bs. of 6 variables
empl    s. of 5 variables
g        of  9
g1       of  9
mat     [1:30, 1:6] 123 18 189
```

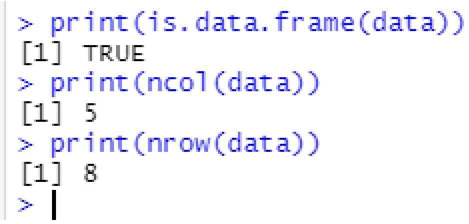


```
> data<- read.csv("C:/Users/Dell/Desktop/R books/employe.csv")
>
> print(data)
  id     name salary start_date       dept
1  1     Rick 623.30 2012-01-01         IT
2  2      Dan 515.20 2013-09-23 Operations
3  3 Michelle 611.00 2014-11-15         IT
4  4     Ryan 729.00 2014-05-11         HR
5  5     Gary 843.25 2015-03-27    Finance
6  6     Nina 578.00 2013-05-21         IT
7  7    Simon 632.80 2013-07-30 Operations
8  8     Guru 722.50 2014-06-17    Finance
> 
```

Analyzing the CSV File

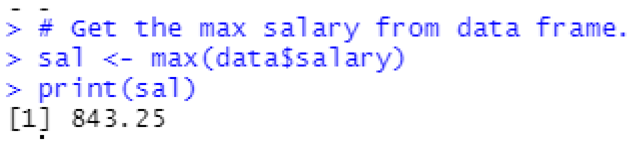
By default the read.csv() function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

Ex:



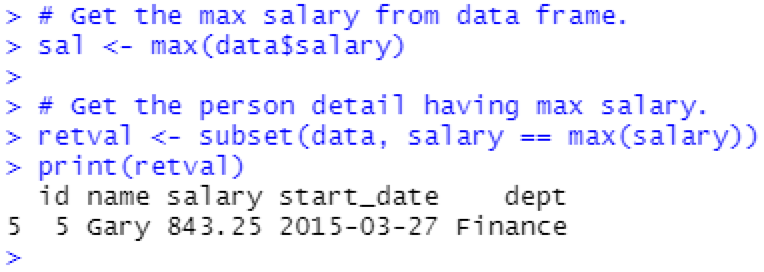
```
> print(is.data.frame(data))
[1] TRUE
> print(ncol(data))
[1] 5
> print(nrow(data))
[1] 8
> 
```

Ex: Get the max salary from the data frame.



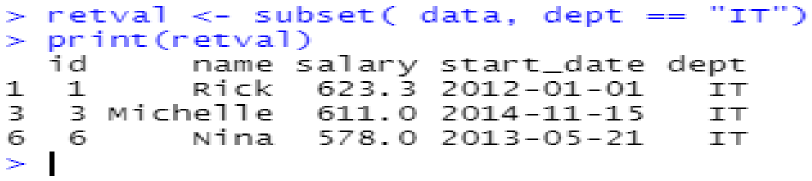
```
> # Get the max salary from data frame.
> sal <- max(data$salary)
> print(sal)
[1] 843.25
```

Ex: Get the person details having max salary.



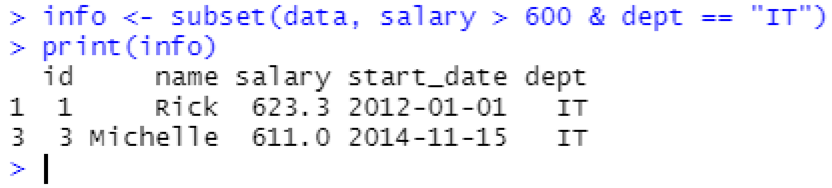
```
> # Get the max salary from data frame.
> sal <- max(data$salary)
>
> # Get the person detail having max salary.
> retval <- subset(data, salary == max(salary))
> print(retval)
  id name salary start_date    dept
5  5 Gary 843.25 2015-03-27 Finance
>
```

Ex: Get all the people working in IT department



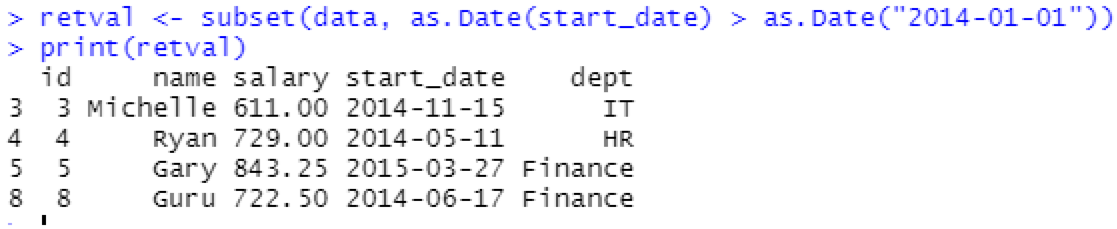
```
> retval <- subset( data, dept == "IT")
> print(retval)
  id     name salary start_date dept
1  1     Rick  623.3 2012-01-01   IT
3  3 Michelle  611.0 2014-11-15   IT
6  6     Nina  578.0 2013-05-21   IT
> 
```

Ex: Get the persons in IT department whose salary is greater than 600



```
> info <- subset(data, salary > 600 & dept == "IT")
> print(info)
  id     name salary start_date dept
1  1     Rick  623.3 2012-01-01   IT
3  3 Michelle  611.0 2014-11-15   IT
> 
```

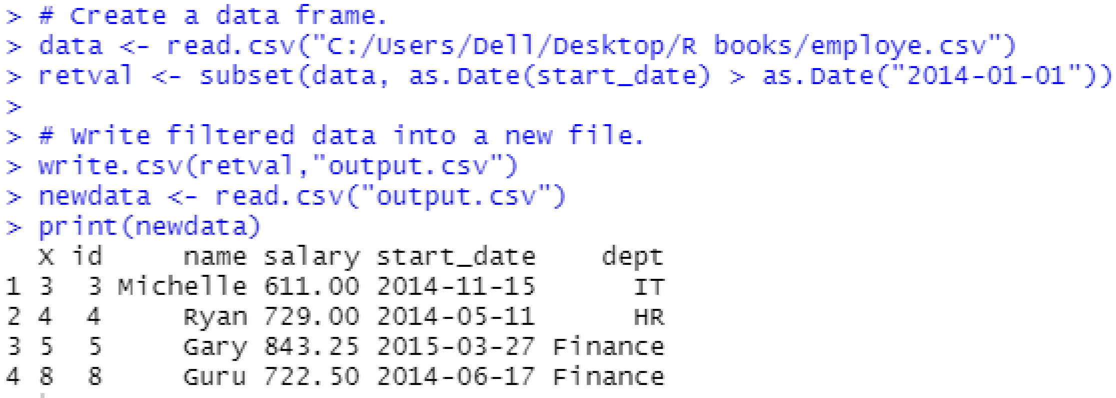
Ex: Get the people who joined on or after 2014



```
> retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
> print(retval)
  id     name salary start_date    dept
3  3 Michelle 611.00 2014-11-15      IT
4  4     Ryan 729.00 2014-05-11      HR
5  5     Gary 843.25 2015-03-27 Finance
8  8     Guru 722.50 2014-06-17 Finance
```

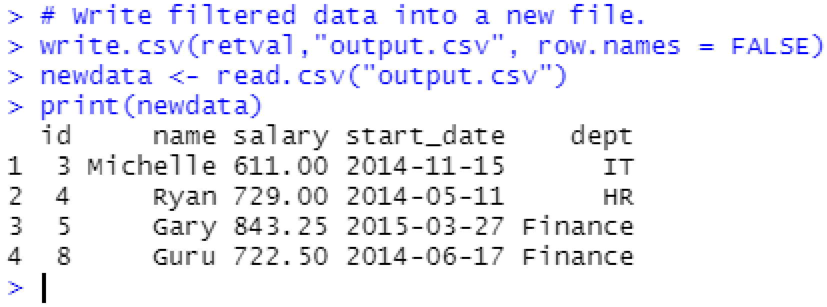
Writing into a CSV File

- R can create a csv file from an existing data frame.
- The write.csv() function is used to create the csv file**.**



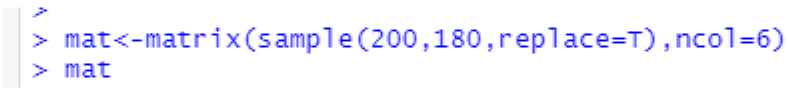
```
> # Create a data frame.
> data <- read.csv("C:/Users/Dell/Desktop/R books/employe.csv")
> retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
> 
> # Write filtered data into a new file.
> write.csv(retval,"output.csv")
> newdata <- read.csv("output.csv")
> print(newdata)
  X id     name salary start_date    dept
1 3  3 Michelle 611.00 2014-11-15      IT
2 4  4     Ryan 729.00 2014-05-11      HR
3 5  5     Gary 843.25 2015-03-27 Finance
4 8  8     Guru 722.50 2014-06-17 Finance
```

Here the column X comes from the data set newper. This can be dropped using additional parameters while writing the file.



```
> # Write filtered data into a new file.
> write.csv(retval,"output.csv", row.names = FALSE)
> newdata <- read.csv("output.csv")
> print(newdata)
  id     name salary start_date    dept
1  3 Michelle 611.00 2014-11-15      IT
2  4     Ryan 729.00 2014-05-11      HR
3  5     Gary 843.25 2015-03-27 Finance
4  8     Guru 722.50 2014-06-17 Finance
> 
```

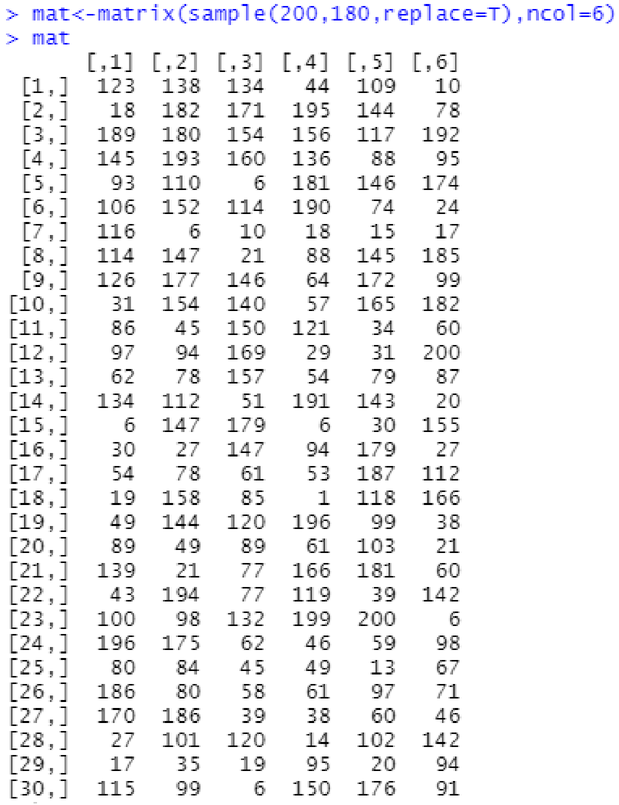
Example:



```
>
> mat<-matrix(sample(200,180,replace=T),ncol=6)
> mat
```

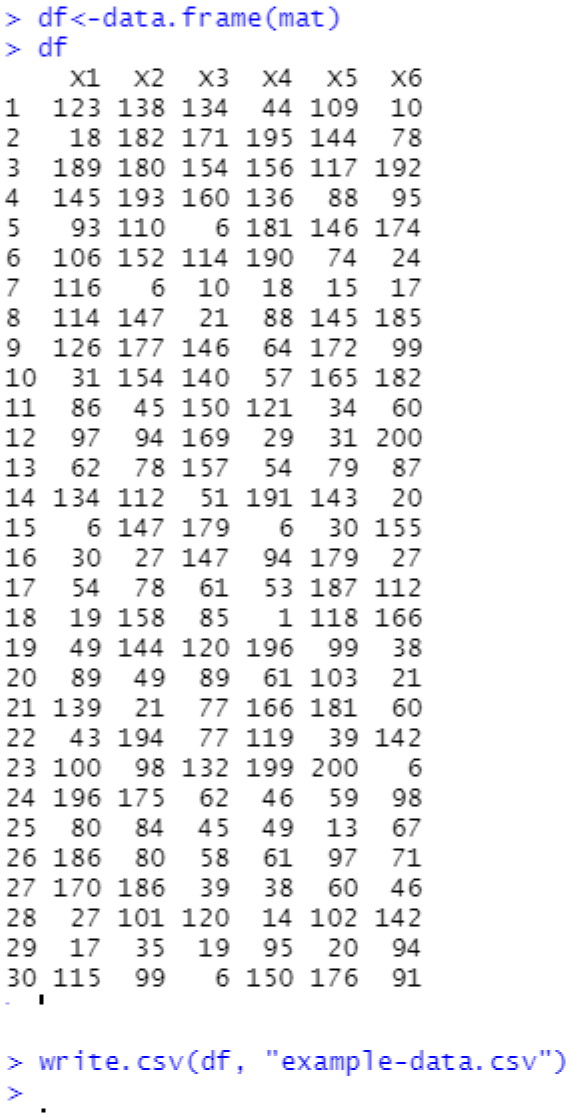
Sample() function

- The sample R function takes a random sample or permutation of a data object.
- Each element of our data can be selected multiple times. In the following R code, we are specifying the replace argument to be TRUE:

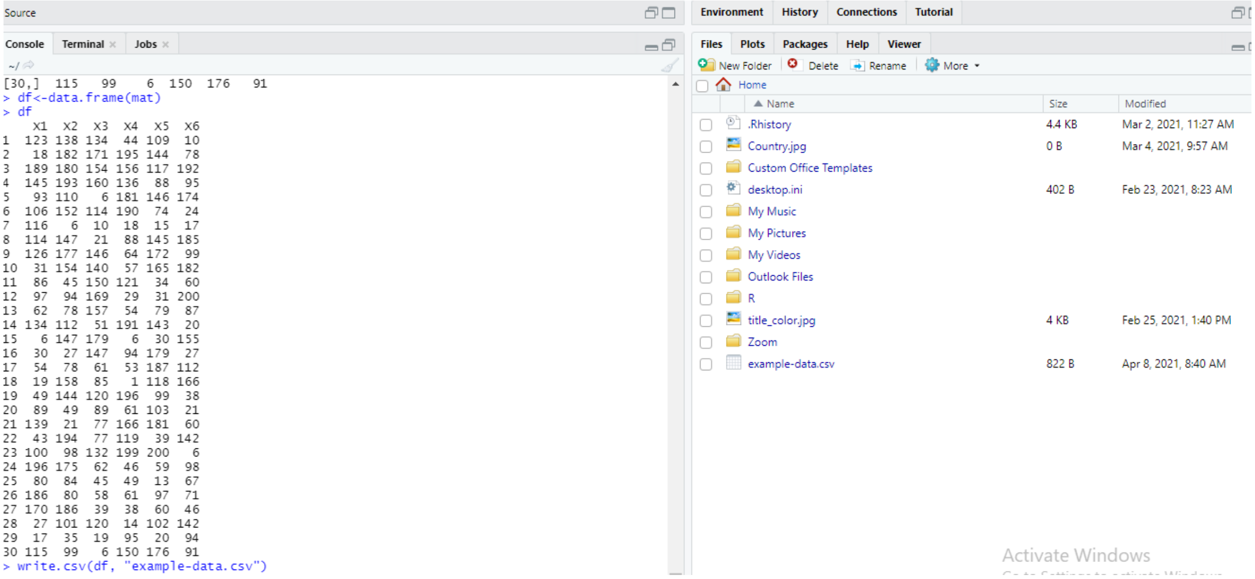


```
> mat<-matrix(sample(200,180,replace=T),ncol=6)
> mat
      [,1] [,2] [,3] [,4] [,5] [,6]
 [1,]  123  138  134   44  109   10
 [2,]   18  182  171  195  144   78
 [3,]  189  180  154  156  117  192
 [4,]  145  193  160  136   88   95
 [5,]   93  110    6  181  146  174
 [6,]  106  152  114  190   74   24
 [7,]  116    6   10   18   15   17
 [8,]  114  147   21   88  145  185
 [9,]  126  177  146   64  172   99
[10,]   31  154  140   57  165  182
[11,]   86   45  150  121   34   60
[12,]   97   94  169   29   31  200
[13,]   62   78  157   54   79   87
[14,]  134  112   51  191  143   20
[15,]    6  147  179    6   30  155
[16,]   30   27  147   94  179   27
[17,]   54   78   61   53  187  112
[18,]   19  158   85    1  118  166
[19,]   49  144  120  196   99   38
[20,]   89   49   89   61  103   21
[21,]  139   21   77  166  181   60
[22,]   43  194   77  119   39  142
[23,]  100   98  132  199  200    6
[24,]  196  175   62   46   59   98
[25,]   80   84   45   49   13   67
[26,]  186   80   58   61   97   71
[27,]  170  186   39   38   60   46
[28,]   27  101  120   14  102  142
[29,]   17   35   19   95   20   94
[30,]  115   99    6  150  176   91
```

Ex: Creating a data frame



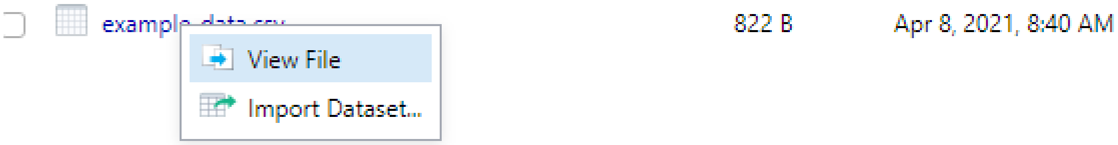
```
> df<-data.frame(mat)
> df
    X1  X2  X3  X4  X5  X6
1  123 138 134  44 109  10
2   18 182 171 195 144  78
3  189 180 154 156 117 192
4  145 193 160 136  88  95
5   93 110   6 181 146 174
6  106 152 114 190  74  24
7  116   6  10  18  15  17
8  114 147  21  88 145 185
9  126 177 146  64 172  99
10  31 154 140  57 165 182
11  86  45 150 121  34  60
12  97  94 169  29  31 200
13  62  78 157  54  79  87
14 134 112  51 191 143  20
15   6 147 179   6  30 155
16  30  27 147  94 179  27
17  54  78  61  53 187 112
18  19 158  85   1 118 166
19  49 144 120 196  99  38
20  89  49  89  61 103  21
21 139  21  77 166 181  60
22  43 194  77 119  39 142
23 100  98 132 199 200   6
24 196 175  62  46  59  98
25  80  84  45  49  13  67
26 186  80  58  61  97  71
27 170 186  39  38  60  46
28  27 101 120  14 102 142
29  17  35  19  95  20  94
30 115  99   6 150 176  91

> write.csv(df, "example-data.csv")
>
```



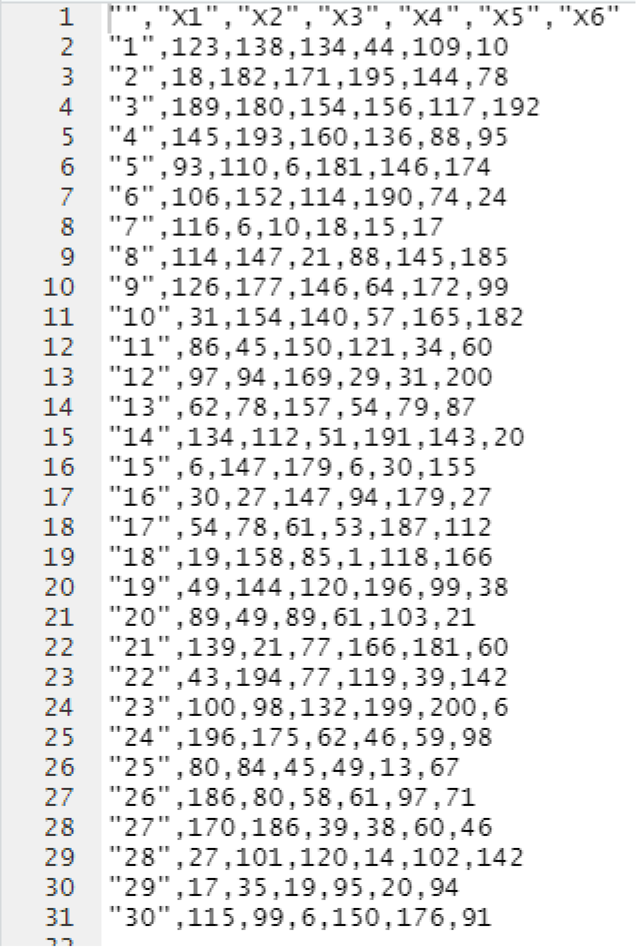
```
[30,]  115   99    6  150  176   91
> df<-data.frame(mat)
> df
    X1  X2  X3  X4  X5  X6
1  123 138 134  44 109  10
2   18 182 171 195 144  78
3  189 180 154 156 117 192
4  145 193 160 136  88  95
5   93 110   6 181 146 174
6  106 152 114 190  74  24
7  116   6  10  18  15  17
8  114 147  21  88 145 185
9  126 177 146  64 172  99
10  31 154 140  57 165 182
11  86  45 150 121  34  60
12  97  94 169  29  31 200
13  62  78 157  54  79  87
14 134 112  51 191 143  20
15   6 147 179   6  30 155
16  30  27 147  94 179  27
17  54  78  61  53 187 112
18  19 158  85   1 118 166
19  49 144 120 196  99  38
20  89  49  89  61 103  21
21 139  21  77 166 181  60
22  43 194  77 119  39 142
23 100  98 132 199 200   6
24 196 175  62  46  59  98
25  80  84  45  49  13  67
26 186  80  58  61  97  71
27 170 186  39  38  60  46
28  27 101 120  14 102 142
29  17  35  19  95  20  94
30 115  99   6 150 176  91
> write.csv(df, "example-data.csv")
```

▲ Name	Size	Modified
.Rhistory	4.4 KB	Mar 2, 2021, 11:27 AM
Country.jpg	0 B	Mar 4, 2021, 9:57 AM
Custom Office Templates		
desktop.ini	402 B	Feb 23, 2021, 8:23 AM
My Music		
My Pictures		
My Videos		
Outlook Files		
R		
title_color.jpg	4 KB	Feb 25, 2021, 1:40 PM
Zoom		
example-data.csv	822 B	Apr 8, 2021, 8:40 AM

example-data.csv	822 B	Apr 8, 2021, 8:40 AM

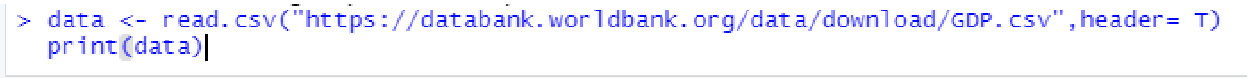


example-data.csv	822 B	Apr 8, 2021, 8:40 AM

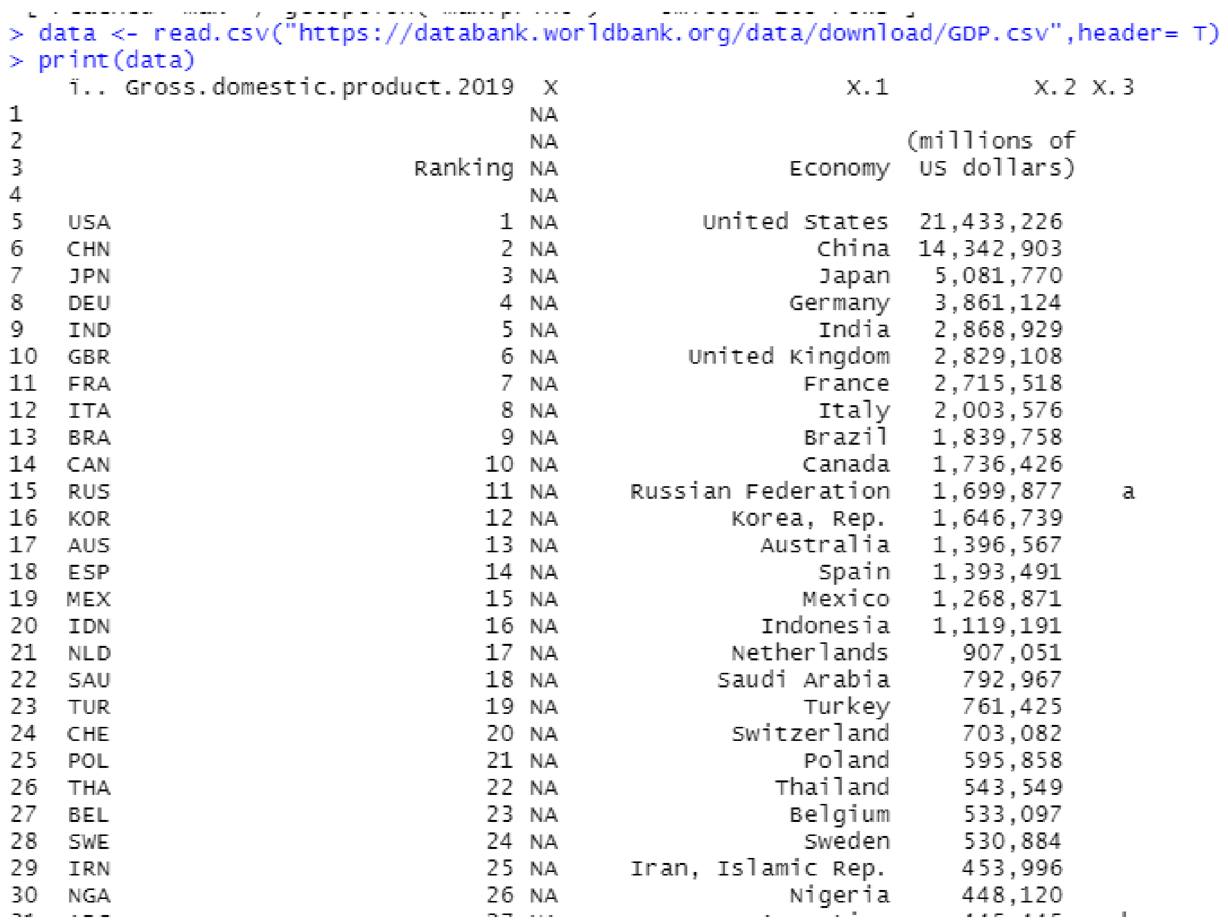


```
"","X1","X2","X3","X4","X5","X6"
"1",123,138,134,44,109,10
"2",18,182,171,195,144,78
"3",189,180,154,156,117,192
"4",145,193,160,136,88,95
"5",93,110,6,181,146,174
"6",106,152,114,190,74,24
"7",116,6,10,18,15,17
"8",114,147,21,88,145,185
"9",126,177,146,64,172,99
"10",31,154,140,57,165,182
"11",86,45,150,121,34,60
"12",97,94,169,29,31,200
"13",62,78,157,54,79,87
"14",134,112,51,191,143,20
"15",6,147,179,6,30,155
"16",30,27,147,94,179,27
"17",54,78,61,53,187,112
"18",19,158,85,1,118,166
"19",49,144,120,196,99,38
"20",89,49,89,61,103,21
"21",139,21,77,166,181,60
"22",43,194,77,119,39,142
"23",100,98,132,199,200,6
"24",196,175,62,46,59,98
"25",80,84,45,49,13,67
"26",186,80,58,61,97,71
"27",170,186,39,38,60,46
"28",27,101,120,14,102,142
"29",17,35,19,95,20,94
"30",115,99,6,150,176,91
```

How to read data files directly from the web in R?

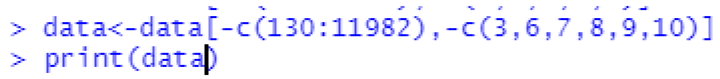


```
> data <- read.csv("https://databank.worldbank.org/data/download/GDP.csv",header= T)
  print(data)
```

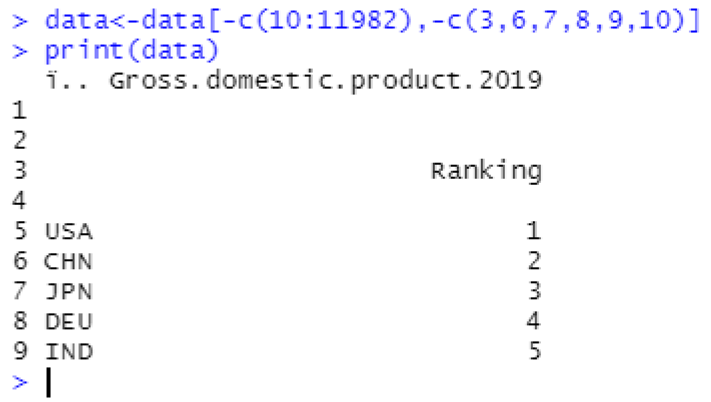


```
> data <- read.csv("https://databank.worldbank.org/data/download/GDP.csv",header= T)
> print(data)
   ï.. Gross.domestic.product.2019  X                X.1        X.2 X.3
1                                  NA                                   
2                                  NA                     (millions of    
3                          Ranking NA            Economy  US dollars)    
4                                  NA                                   
5   USA                          1 NA      United States  21,433,226    
6   CHN                          2 NA              China  14,342,903    
7   JPN                          3 NA              Japan   5,081,770    
8   DEU                          4 NA            Germany   3,861,124    
9   IND                          5 NA              India   2,868,929    
10  GBR                          6 NA     United Kingdom   2,829,108    
11  FRA                          7 NA             France   2,715,518    
12  ITA                          8 NA              Italy   2,003,576    
13  BRA                          9 NA             Brazil   1,839,758    
14  CAN                         10 NA             Canada   1,736,426    
15  RUS                         11 NA Russian Federation   1,699,877   a
16  KOR                         12 NA        Korea, Rep.   1,646,739    
17  AUS                         13 NA          Australia   1,396,567    
18  ESP                         14 NA              Spain   1,393,491    
19  MEX                         15 NA             Mexico   1,268,871    
20  IDN                         16 NA          Indonesia   1,119,191    
21  NLD                         17 NA        Netherlands     907,051    
22  SAU                         18 NA       Saudi Arabia     792,967    
23  TUR                         19 NA             Turkey     761,425    
24  CHE                         20 NA        Switzerland     703,082    
25  POL                         21 NA             Poland     595,858    
26  THA                         22 NA           Thailand     543,549    
27  BEL                         23 NA            Belgium     533,097    
28  SWE                         24 NA             Sweden     530,884    
29  IRN                         25 NA Iran, Islamic Rep.     453,996    
30  NGA                         26 NA            Nigeria     448,120    
```

How to remove null columns and rows?



```
> data<-data[-c(130:11982),-c(3,6,7,8,9,10)]
> print(data)
```



```
> data<-data[-c(10:11982),-c(3,6,7,8,9,10)]
> print(data)
  ï.. Gross.domestic.product.2019
1                                
2                                
3                         Ranking
4                                
5 USA                           1
6 CHN                           2
7 JPN                           3
8 DEU                           4
9 IND                           5
> 
```

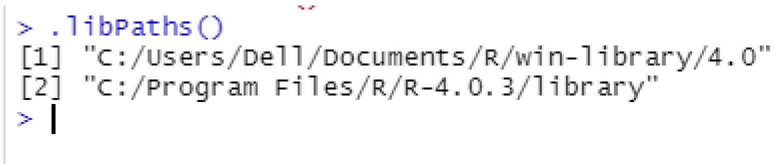
R Packages

- R packages are the collection of R functions, sample data, and compile codes.
- In the R environment, these packages are stored under a directory called "**library**."
- During installation, R installs a set of packages. We can add packages later when they are needed for some specific purpose.
- Only the default packages will be available when we start the R console.
- Other packages which are already installed will be loaded explicitly to be used by the R program.

There is the following list of commands to be used to check, verify, and use the R packages.

Check Available R Packages

- To check the available R Packages, we have to find the library location in which R packages are contained.
- R provides .**libPaths()** function to find the library locations.

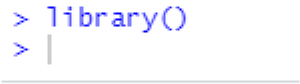


```
> .libPaths()
[1] "C:/Users/Dell/Documents/R/win-library/4.0"
[2] "C:/Program Files/R/R-4.0.3/library"
> 
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

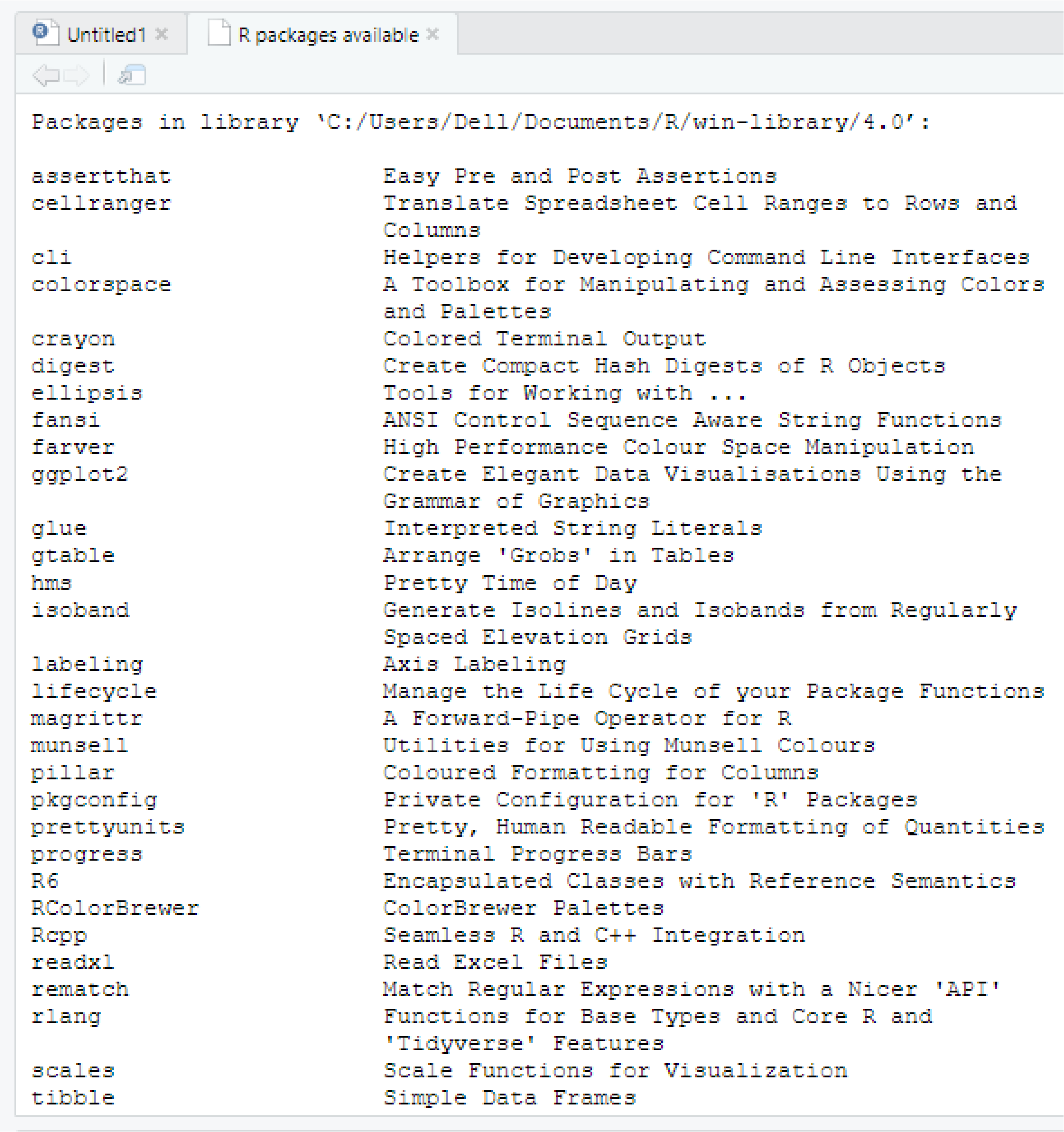
Get the list of all the packages installed

- R provides a **library()** function, which allows us to get the list of all the installed packages.



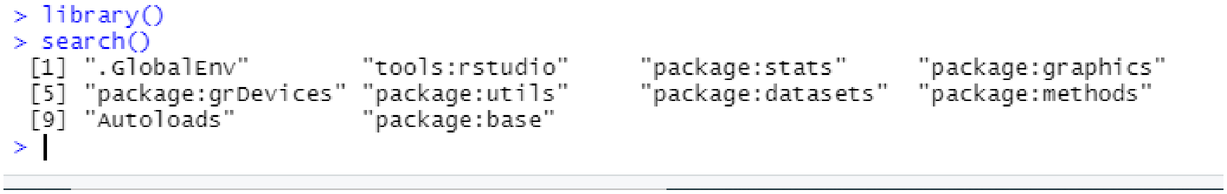
```
> library()
> 
```

- When we execute the above code, it produces the following result.
- It may vary depending on the local settings of your pc.



```
Packages in library ‘C:/Users/Dell/Documents/R/win-library/4.0’:

assertthat               Easy Pre and Post Assertions
cellranger               Translate Spreadsheet Cell Ranges to Rows and
                         Columns
cli                      Helpers for Developing Command Line Interfaces
colorspace               A Toolbox for Manipulating and Assessing Colors
                         and Palettes
crayon                   Colored Terminal Output
digest                   Create Compact Hash Digests of R Objects
ellipsis                 Tools for Working with ...
fansi                    ANSI Control Sequence Aware String Functions
farver                   High Performance Colour Space Manipulation
ggplot2                  Create Elegant Data Visualisations Using the
                         Grammar of Graphics
glue                     Interpreted String Literals
gtable                   Arrange 'Grobs' in Tables
hms                      Pretty Time of Day
isoband                  Generate Isolines and Isobands from Regularly
                         Spaced Elevation Grids
labeling                 Axis Labeling
lifecycle                Manage the Life Cycle of your Package Functions
magrittr                 A Forward-Pipe Operator for R
munsell                  Utilities for Using Munsell Colours
pillar                   Coloured Formatting for Columns
pkgconfig                Private Configuration for 'R' Packages
prettyunits              Pretty, Human Readable Formatting of Quantities
progress                 Terminal Progress Bars
R6                       Encapsulated Classes with Reference Semantics
RColorBrewer             ColorBrewer Palettes
Rcpp                     Seamless R and C++ Integration
readxl                   Read Excel Files
rematch                  Match Regular Expressions with a Nicer 'API'
rlang                    Functions for Base Types and Core R and
                         'Tidyverse' Features
scales                   Scale Functions for Visualization
tibble                   Simple Data Frames
```

- R provides **search()** function to get all packages currently loaded in the R environment.



```
> library()
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
> |
```

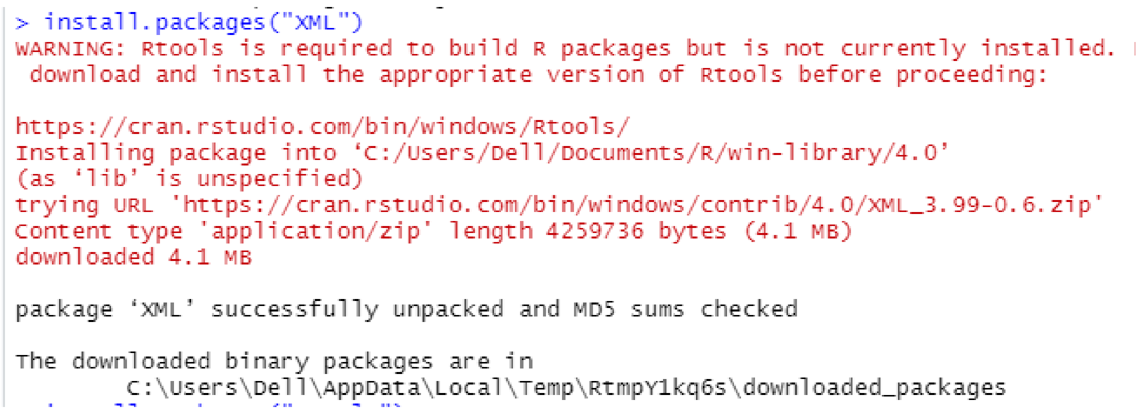
Install a New Package

- In R, there are two techniques to add new R packages.
- The first technique is installing a package directly from the CRAN directory,
- and the second one is to install it manually after downloading the package to our local system.

Install directly from CRAN

- The following command is used to get the packages directly from CRAN webpage and install the package in the R environment.
- We may be prompted to choose the nearest mirror. Choose the one appropriate to our location.

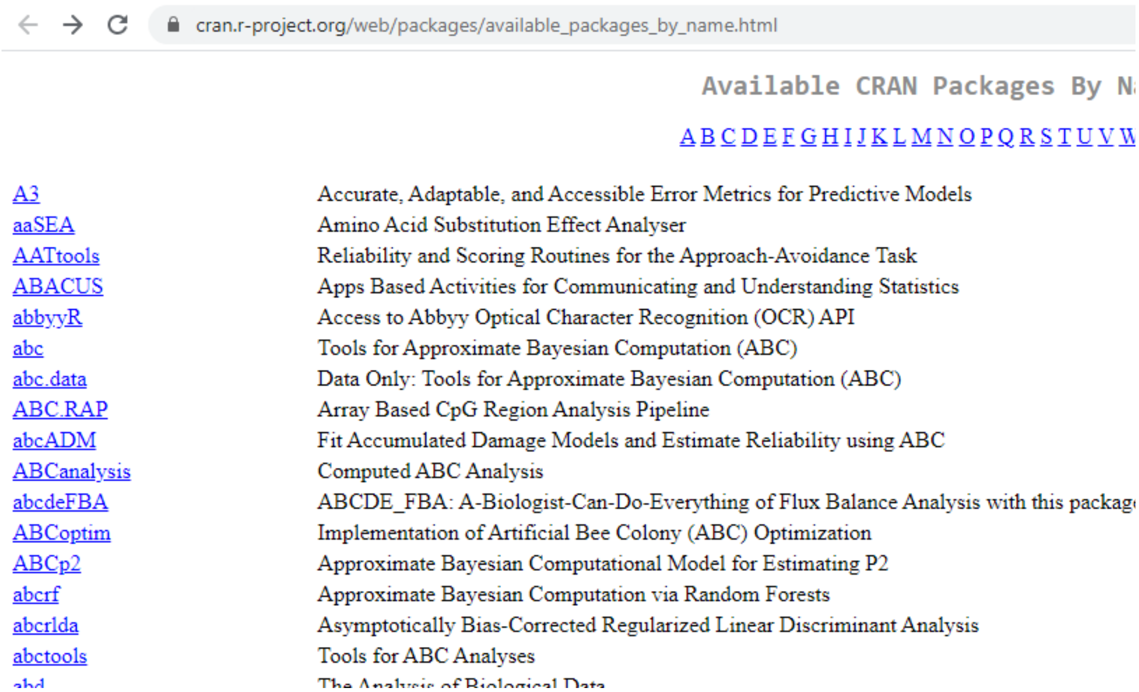
```
install.packages("Package Name")
```



```
> install.packages("XML")
WARNING: Rtools is required to build R packages but is not currently installed.
 download and install the appropriate version of Rtools before proceeding:

https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/Dell/Documents/R/win-library/4.0'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.0/XML_3.99-0.6.zip'
Content type 'application/zip' length 4259736 bytes (4.1 MB)
downloaded 4.1 MB

package 'XML' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Dell\AppData\Local\Temp\RtmpY1kq6s\downloaded_packages
```

Install package manually

- To install a package manually, we first have to download it from https://cran.r-project.org/web/packages/available_packages_by_name.html.
- The required package will be saved as a .zip file in a suitable location in the local system.

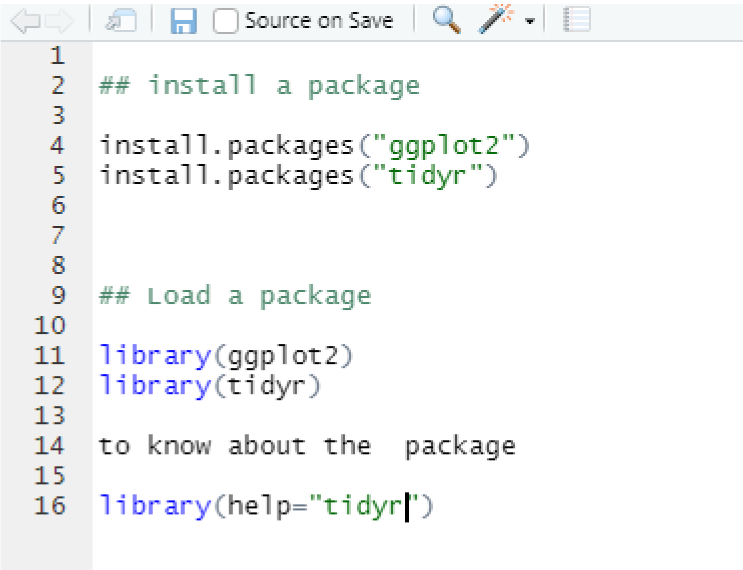


Available CRAN Packages By N

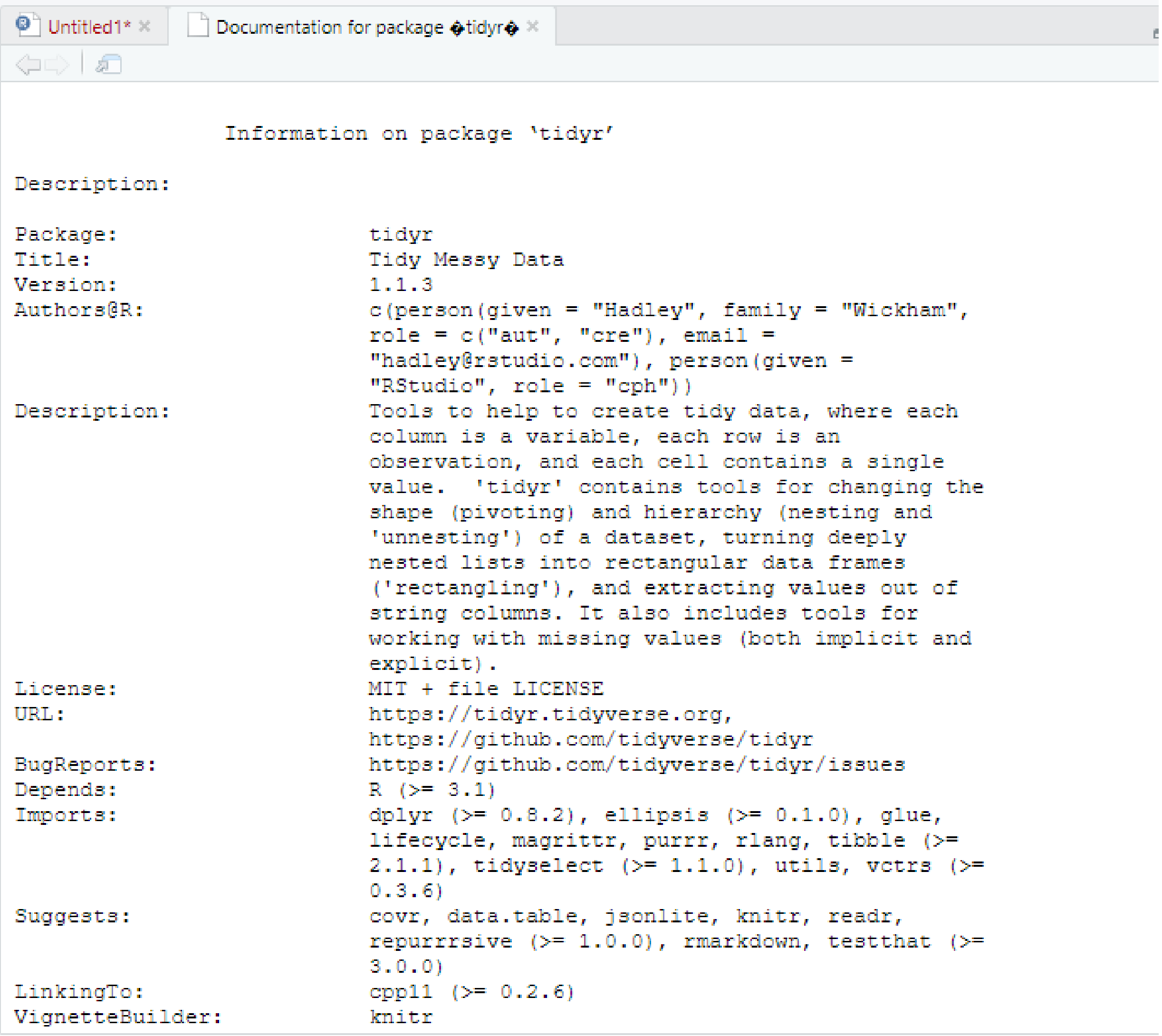
A B C D E F G H I J K L M N O P Q R S T U V W

A3	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
aaSEA	Amino Acid Substitution Effect Analyser
AATtools	Reliability and Scoring Routines for the Approach-Avoidance Task
ABACUS	Apps Based Activities for Communicating and Understanding Statistics
abbyyR	Access to Abbyy Optical Character Recognition (OCR) API
abc	Tools for Approximate Bayesian Computation (ABC)
abc.data	Data Only: Tools for Approximate Bayesian Computation (ABC)
ABC.RAP	Array Based CpG Region Analysis Pipeline
abcADM	Fit Accumulated Damage Models and Estimate Reliability using ABC
ABCanalysis	Computed ABC Analysis
abcdeFBA	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this packag
ABCoptim	Implementation of Artificial Bee Colony (ABC) Optimization
ABCp2	Approximate Bayesian Computational Model for Estimating P2
abcrf	Approximate Bayesian Computation via Random Forests
abcrlda	Asymptotically Bias-Corrected Regularized Linear Discriminant Analysis
abctools	Tools for ABC Analyses
abd	The Analysis of Biological Data

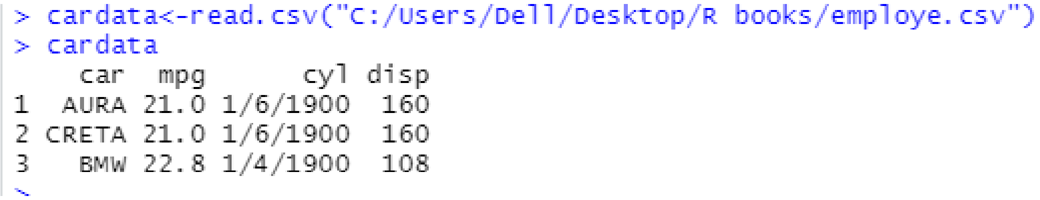
Or you can install and load package in the following manner:



```

## install a package

install.packages("ggplot2")
install.packages("tidyr")



## Load a package

library(ggplot2)
library(tidyr)

to know about the  package

library(help="tidyr")
```



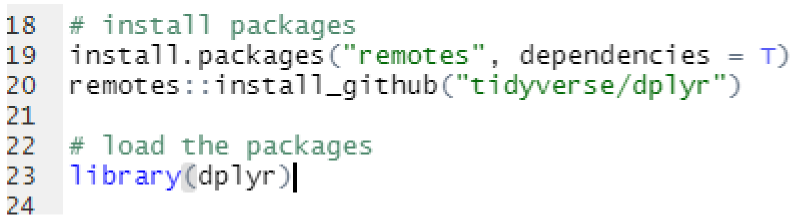
```
                Information on package 'tidyr'

Description:

Package:                   tidyr
Title:                     Tidy Messy Data
Version:                   1.1.3
Authors@R:                 c(person(given = "Hadley", family = "Wickham",
                           role = c("aut", "cre"), email =
                           "hadley@rstudio.com"), person(given =
                           "RStudio", role = "cph"))
Description:               Tools to help to create tidy data, where each
                           column is a variable, each row is an
                           observation, and each cell contains a single
                           value.  'tidyr' contains tools for changing the
                           shape (pivoting) and hierarchy (nesting and
                           'unnesting') of a dataset, turning deeply
                           nested lists into rectangular data frames
                           ('rectangling'), and extracting values out of
                           string columns. It also includes tools for
                           working with missing values (both implicit and
                           explicit).
License:                   MIT + file LICENSE
URL:                       https://tidyr.tidyverse.org,
                           https://github.com/tidyverse/tidyr
BugReports:                https://github.com/tidyverse/tidyr/issues
Depends:                   R (>= 3.1)
Imports:                   dplyr (>= 0.8.2), ellipsis (>= 0.1.0), glue,
                           lifecycle, magrittr, purrr, rlang, tibble (>=
                           2.1.1), tidyselect (>= 1.1.0), utils, vctrs (>=
                           0.3.6)
Suggests:                  covr, data.table, jsonlite, knitr, readr,
                           repurrrsive (>= 1.0.0), rmarkdown, testthat (>=
                           3.0.0)
LinkingTo:                 cpp11 (>= 0.2.6)
VignetteBuilder:           knitr
```



```
> cardata<-read.csv("C:/Users/Dell/Desktop/R books/employe.csv")
> cardata
    car  mpg      cyl disp
1  AURA 21.0 1/6/1900  160
2 CRETA 21.0 1/6/1900  160
3   BMW 22.8 1/4/1900  108
>
```

dplyr

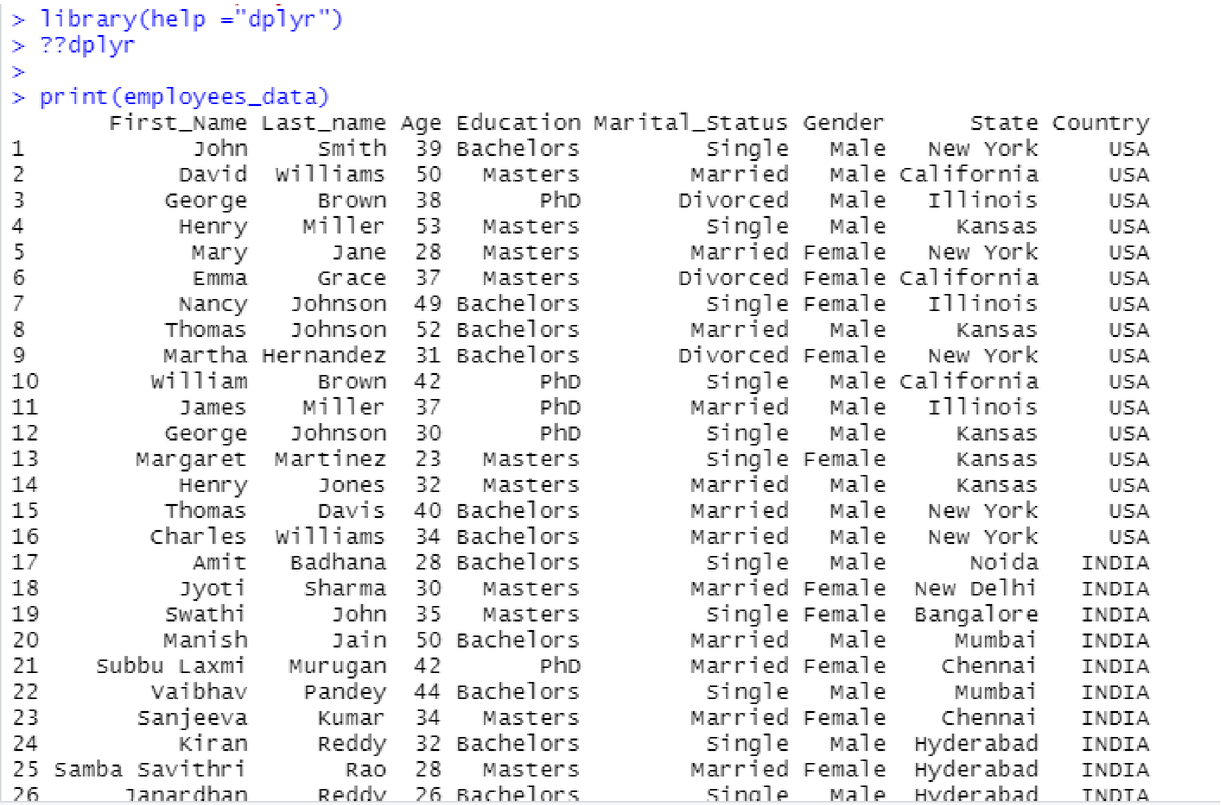
- R allows us to perform data wrangling and data analysis. R provides the **dplyr** library for this purpose. This library facilitates several functions for the data frame in R.
- A package that helps transform tabular data.



```
# install packages
install.packages("remotes", dependencies = T)
remotes::install_github("tidyverse/dplyr")

# load the packages
library(dplyr)
```

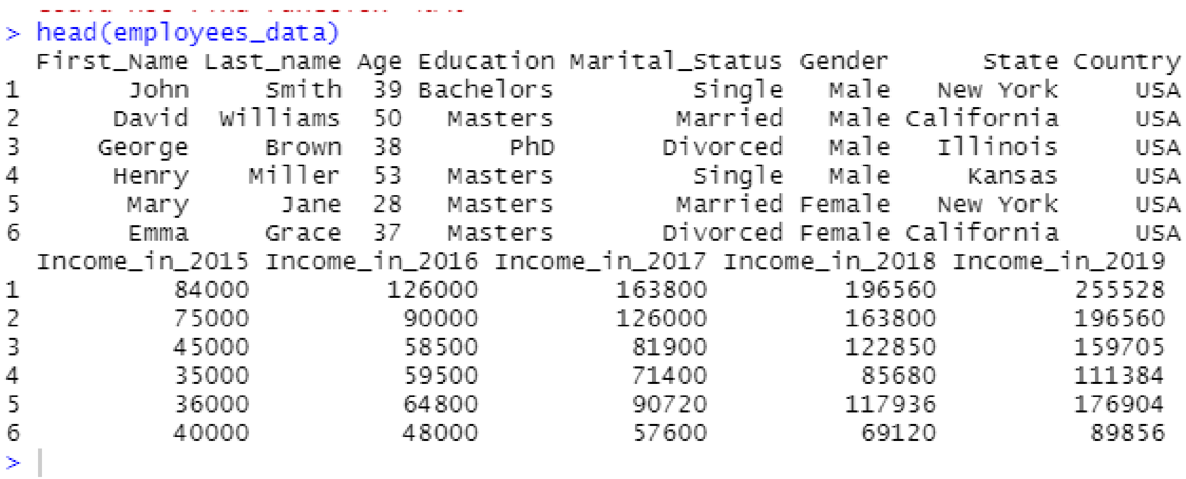
# load the data

Employees_dat<-read.csv(file=**"https://github.com/iAnalyticsGeek/Datasets/raw/master/employee_data.csv",** na.strings = c("", "NA"))

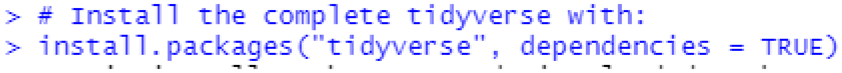


```
> library(help ="dplyr")
> ??dplyr
>
> print(employees_data)
       First_Name Last_name Age Education Marital_Status Gender      State Country
1            John     Smith  39 Bachelors         Single   Male   New York     USA
2           David  Williams  50   Masters        Married   Male California     USA
3          George     Brown  38       PhD       Divorced   Male   Illinois     USA
4           Henry    Miller  53   Masters         Single   Male     Kansas     USA
5            Mary      Jane  28   Masters        Married Female   New York     USA
6            Emma     Grace  37   Masters       Divorced Female California     USA
7           Nancy   Johnson  49 Bachelors         Single Female   Illinois     USA
8          Thomas   Johnson  52 Bachelors        Married   Male     Kansas     USA
9          Martha Hernandez  31 Bachelors       Divorced Female   New York     USA
10        William     Brown  42       PhD         Single   Male California     USA
11          James    Miller  37       PhD        Married   Male   Illinois     USA
12         George   Johnson  30       PhD         Single   Male     Kansas     USA
13       Margaret  Martinez  23   Masters         Single Female     Kansas     USA
14          Henry     Jones  32   Masters        Married   Male     Kansas     USA
15         Thomas     Davis  40 Bachelors        Married   Male   New York     USA
16        Charles  Williams  34 Bachelors        Married   Male   New York     USA
17           Amit   Badhana  28 Bachelors         Single   Male      Noida   INDIA
18          Jyoti    Sharma  30   Masters        Married Female  New Delhi   INDIA
19         Swathi      John  35   Masters         Single Female  Bangalore   INDIA
20         Manish      Jain  50 Bachelors        Married   Male     Mumbai   INDIA
21    Subbu Laxmi   Murugan  42       PhD        Married Female    Chennai   INDIA
22        Vaibhav    Pandey  44 Bachelors         Single   Male     Mumbai   INDIA
23       Sanjeeva     Kumar  34   Masters        Married Female    Chennai   INDIA
24          Kiran     Reddy  32 Bachelors         Single   Male  Hyderabad   INDIA
25 Samba Savithri       Rao  28   Masters        Married Female  Hyderabad   INDIA
26      Janardhan     Reddy  26 Bachelors         Single   Male  Hyderabad   INDIA
```

Ex: six rows of employee data.

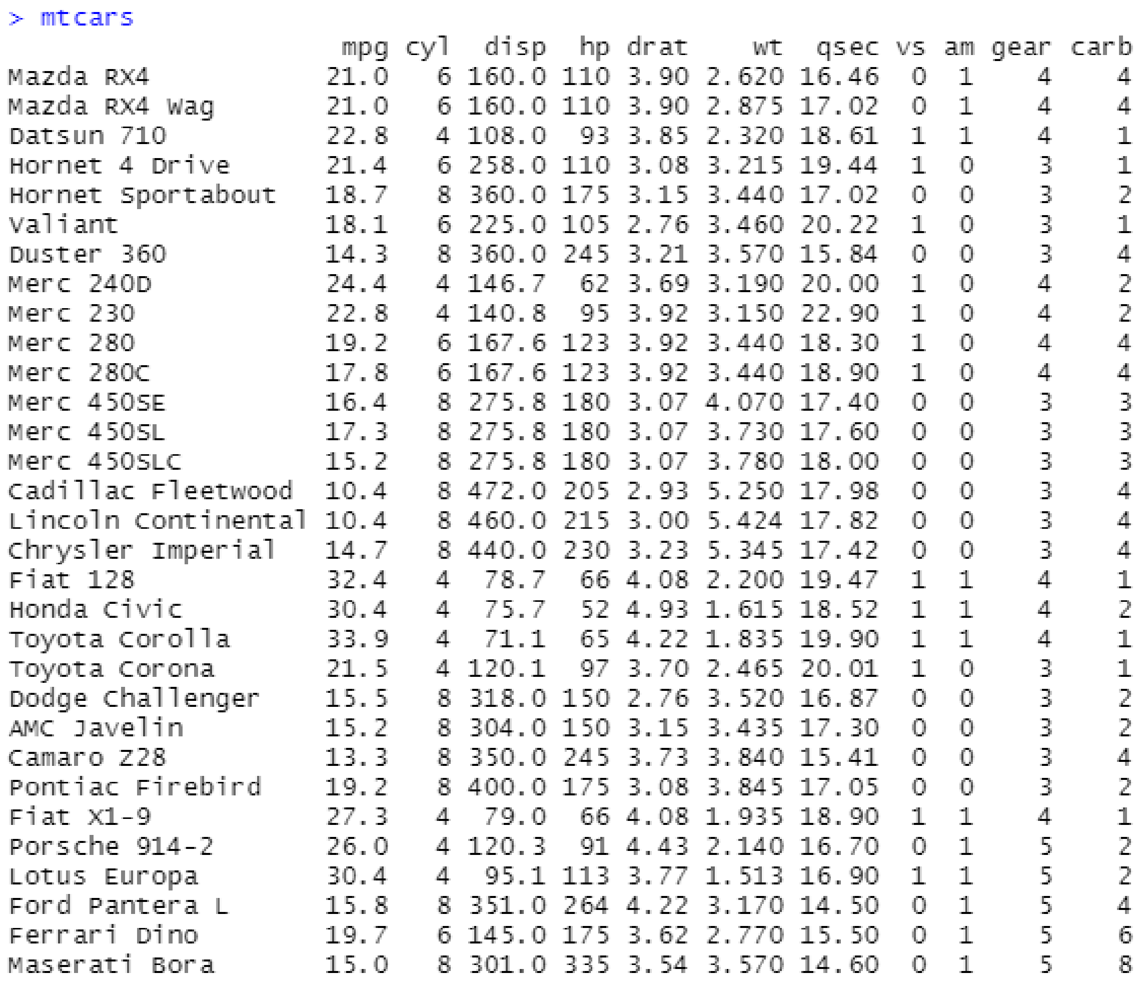


```
> head(employees_data)
  First_Name Last_name Age Education Marital_Status Gender      State Country
1       John     Smith  39 Bachelors         Single   Male   New York     USA
2      David  Williams  50   Masters        Married   Male California     USA
3     George     Brown  38       PhD       Divorced   Male   Illinois     USA
4      Henry    Miller  53   Masters         Single   Male     Kansas     USA
5       Mary      Jane  28   Masters        Married Female   New York     USA
6       Emma     Grace  37   Masters       Divorced Female California     USA
  Income_in_2015 Income_in_2016 Income_in_2017 Income_in_2018 Income_in_2019
1          84000         126000         163800         196560         255528
2          75000          90000         126000         163800         196560
3          45000          58500          81900         122850         159705
4          35000          59500          71400          85680         111384
5          36000          64800          90720         117936         176904
6          40000          48000          57600          69120          89856
> 
```



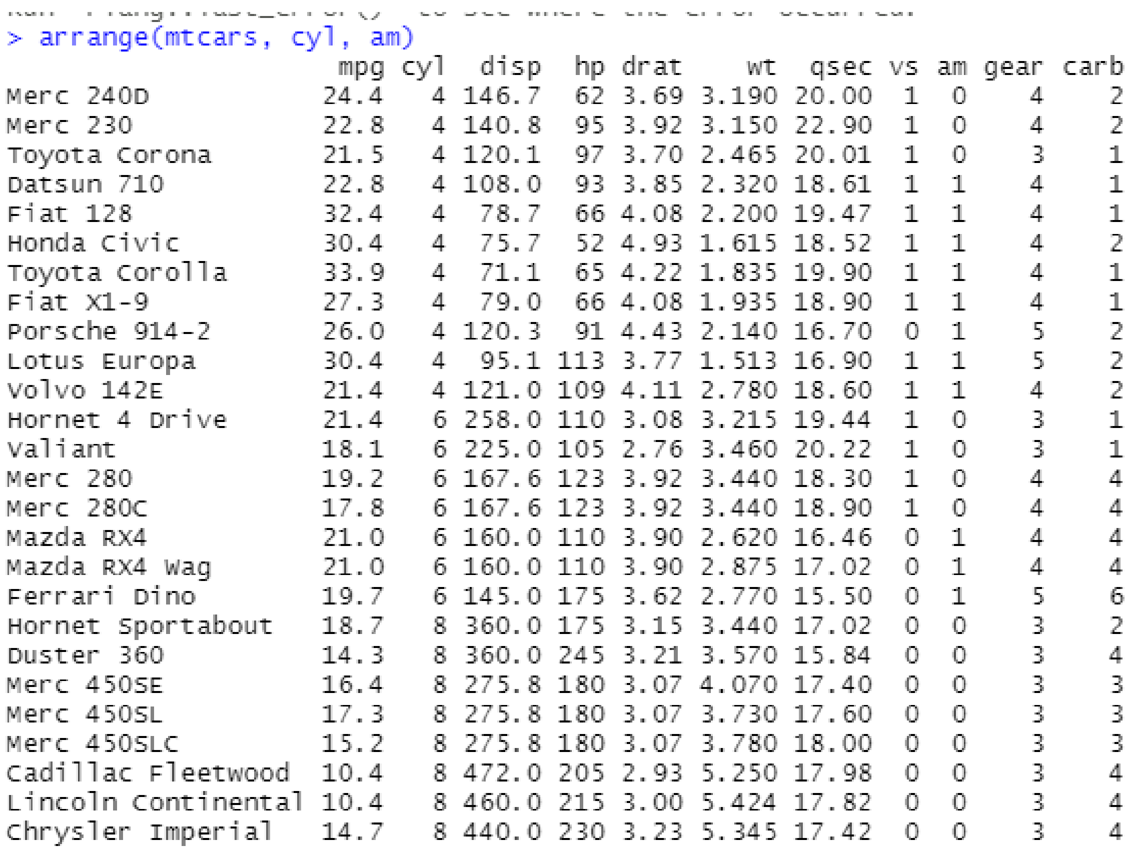
```
> # Install the complete tidyverse with:
> install.packages("tidyverse", dependencies = TRUE)
```

Ex:



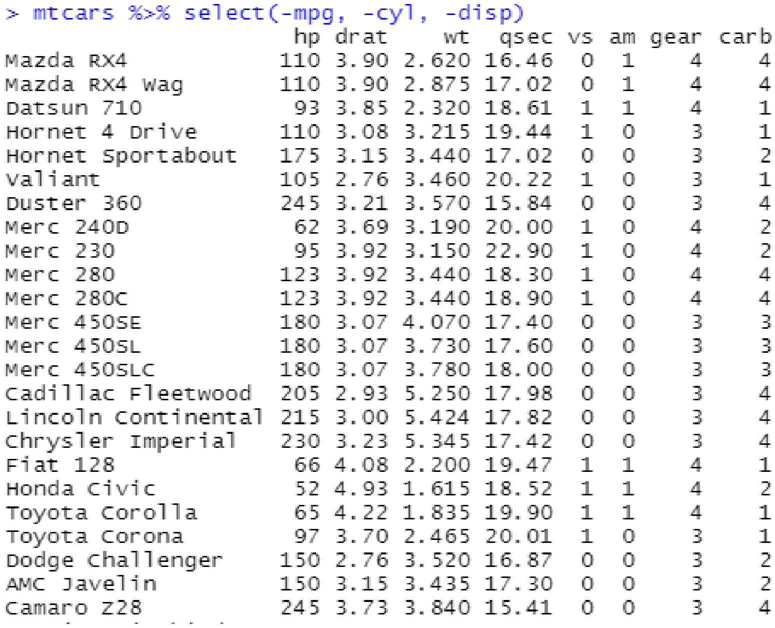
```
> mtcars
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

Ex: arrange () function to arrange the data in ascending order

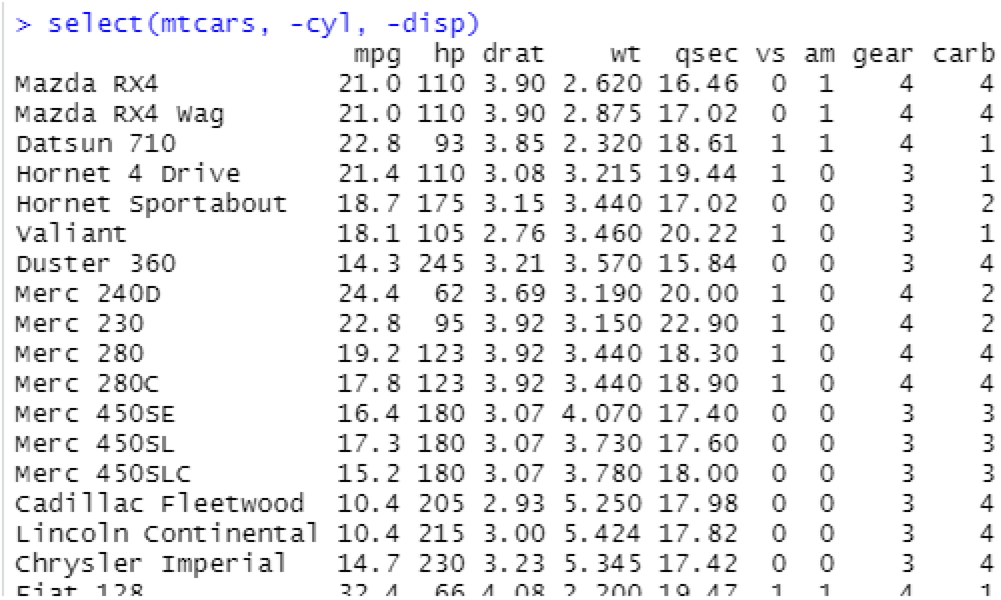


```
Run `rlang::last_error()` to see where the error occurred.
> arrange(mtcars, cyl, am)
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
```

Ex: to remove columns by using select()

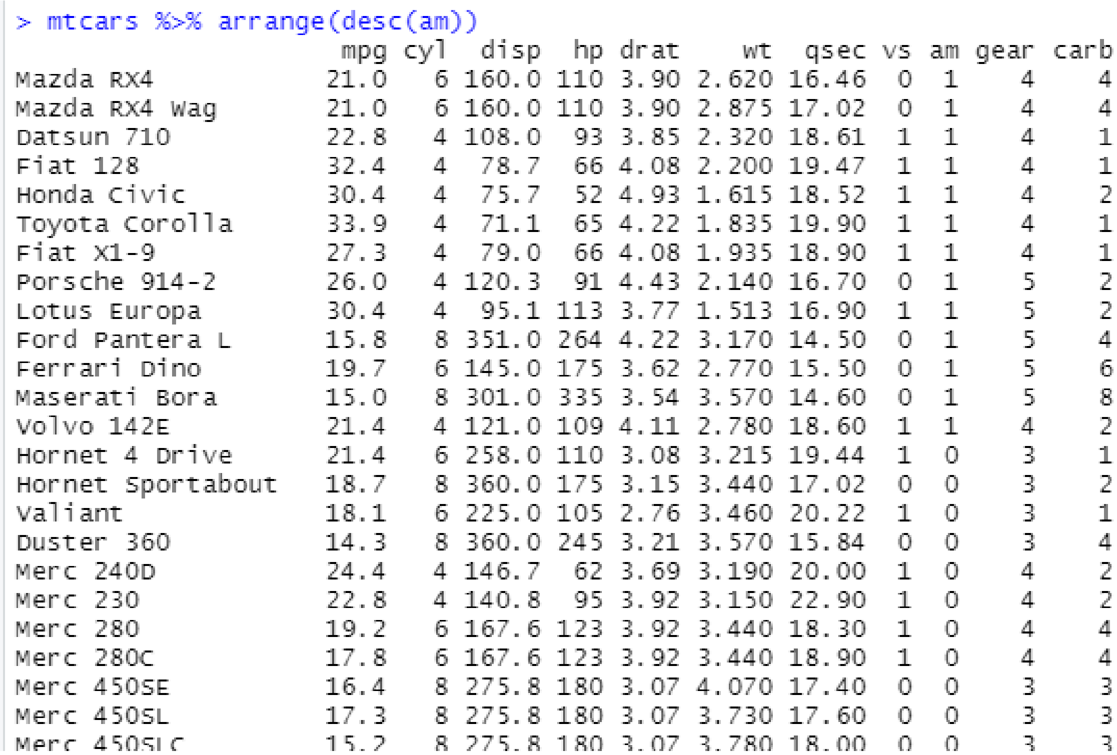


```
> mtcars %>% select(-mpg, -cyl, -disp)
                     hp drat    wt  qsec vs am gear carb
Mazda RX4           110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       110 3.90 2.875 17.02  0  1    4    4
Datsun 710           93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   175 3.15 3.440 17.02  0  0    3    2
Valiant             105 2.76 3.460 20.22  1  0    3    1
Duster 360          245 3.21 3.570 15.84  0  0    3    4
Merc 240D            62 3.69 3.190 20.00  1  0    4    2
Merc 230             95 3.92 3.150 22.90  1  0    4    2
Merc 280            123 3.92 3.440 18.30  1  0    4    4
Merc 280C           123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   230 3.23 5.345 17.42  0  0    3    4
Fiat 128             66 4.08 2.200 19.47  1  1    4    1
Honda Civic          52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla       65 4.22 1.835 19.90  1  1    4    1
Toyota Corona        97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          245 3.73 3.840 15.41  0  0    3    4
```



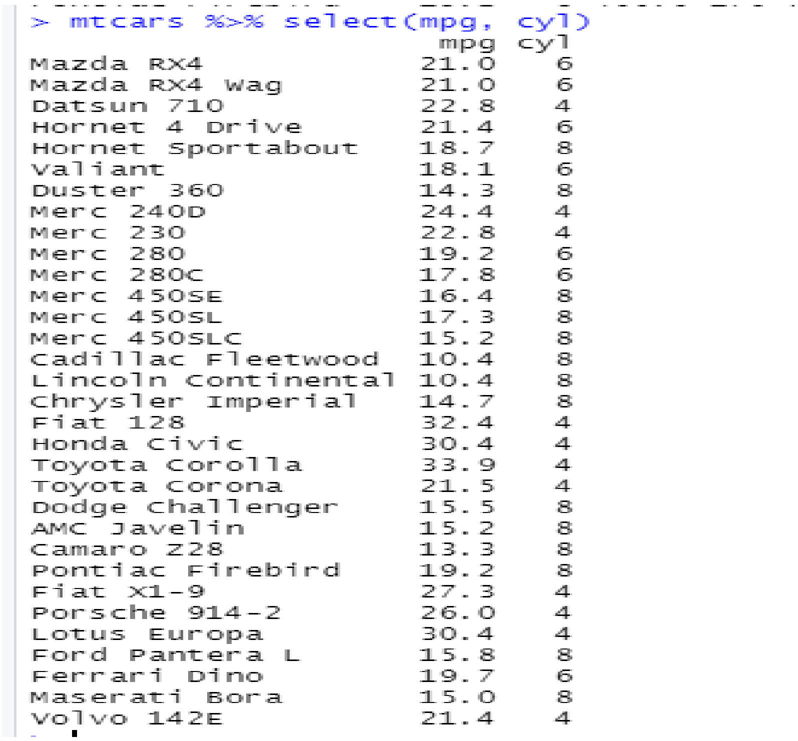
```
> select(mtcars, -cyl, -disp)
                     mpg  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4  66 4.08 2.200 19.47  1  1    4    1
```

Ex: to arrange columns in a descending order.



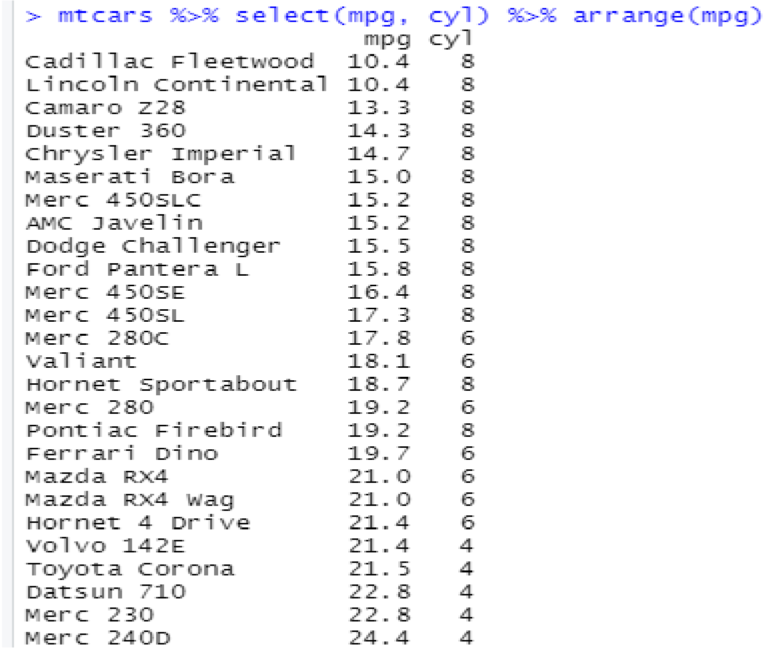
```
> mtcars %>% arrange(desc(am))
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
```

Ex: To select particular columns we use select()



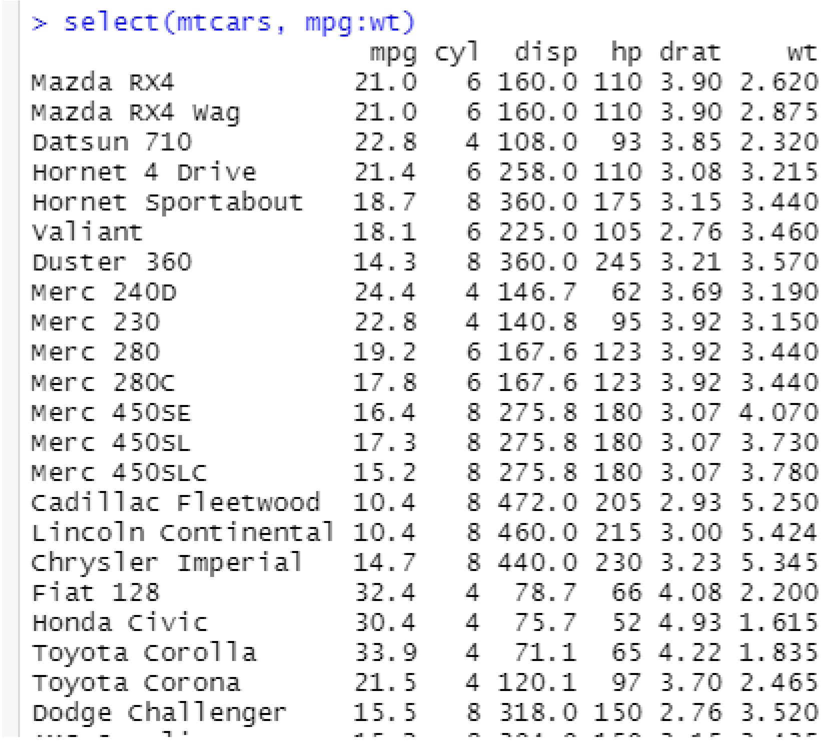
```
> mtcars %>% select(mpg, cyl)
                     mpg cyl
Mazda RX4           21.0   6
Mazda RX4 Wag       21.0   6
Datsun 710          22.8   4
Hornet 4 Drive      21.4   6
Hornet Sportabout   18.7   8
Valiant             18.1   6
Duster 360          14.3   8
Merc 240D           24.4   4
Merc 230            22.8   4
Merc 280            19.2   6
Merc 280C           17.8   6
Merc 450SE          16.4   8
Merc 450SL          17.3   8
Merc 450SLC         15.2   8
Cadillac Fleetwood  10.4   8
Lincoln Continental 10.4   8
Chrysler Imperial   14.7   8
Fiat 128            32.4   4
Honda Civic         30.4   4
Toyota Corolla      33.9   4
Toyota Corona       21.5   4
Dodge Challenger    15.5   8
AMC Javelin         15.2   8
Camaro Z28          13.3   8
Pontiac Firebird    19.2   8
Fiat X1-9           27.3   4
Porsche 914-2       26.0   4
Lotus Europa        30.4   4
Ford Pantera L      15.8   8
Ferrari Dino        19.7   6
Maserati Bora       15.0   8
Volvo 142E          21.4   4
```

Ex: Use of select() , pipe operator (%>%) and arrange()



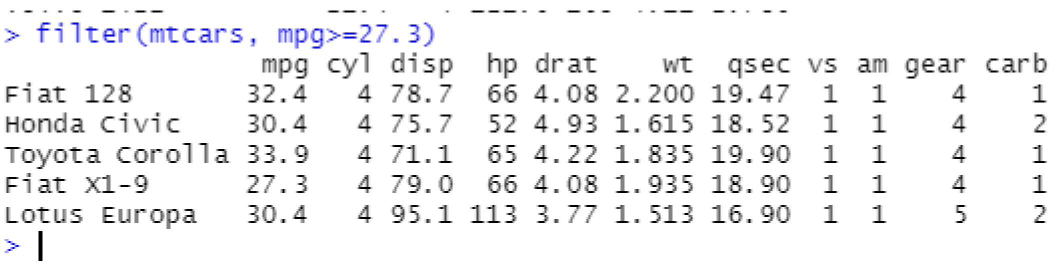
```
> mtcars %>% select(mpg, cyl) %>% arrange(mpg)
                     mpg cyl
Cadillac Fleetwood  10.4   8
Lincoln Continental 10.4   8
Camaro Z28          13.3   8
Duster 360          14.3   8
Chrysler Imperial   14.7   8
Maserati Bora       15.0   8
Merc 450SLC         15.2   8
AMC Javelin         15.2   8
Dodge Challenger    15.5   8
Ford Pantera L      15.8   8
Merc 450SE          16.4   8
Merc 450SL          17.3   8
Merc 280C           17.8   6
Valiant             18.1   6
Hornet Sportabout   18.7   8
Merc 280            19.2   6
Pontiac Firebird    19.2   8
Ferrari Dino        19.7   6
Mazda RX4           21.0   6
Mazda RX4 Wag       21.0   6
Hornet 4 Drive      21.4   6
Volvo 142E          21.4   4
Toyota Corona       21.5   4
Datsun 710          22.8   4
Merc 230            22.8   4
Merc 240D           24.4   4
```

Ex:range operator in within select()



```
> select(mtcars, mpg:wt)
                     mpg cyl  disp  hp drat    wt
Mazda RX4           21.0   6 160.0 110 3.90 2.620
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875
Datsun 710          22.8   4 108.0  93 3.85 2.320
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440
Valiant             18.1   6 225.0 105 2.76 3.460
Duster 360          14.3   8 360.0 245 3.21 3.570
Merc 240D           24.4   4 146.7  62 3.69 3.190
Merc 230            22.8   4 140.8  95 3.92 3.150
Merc 280            19.2   6 167.6 123 3.92 3.440
Merc 280C           17.8   6 167.6 123 3.92 3.440
Merc 450SE          16.4   8 275.8 180 3.07 4.070
Merc 450SL          17.3   8 275.8 180 3.07 3.730
Merc 450SLC         15.2   8 275.8 180 3.07 3.780
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250
Lincoln Continental 10.4   8 460.0 215 3.00 5.424
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345
Fiat 128            32.4   4  78.7  66 4.08 2.200
Honda Civic         30.4   4  75.7  52 4.93 1.615
Toyota Corolla      33.9   4  71.1  65 4.22 1.835
Toyota Corona       21.5   4 120.1  97 3.70 2.465
Dodge Challenger    15.5   8 318.0 150 2.76 3.520
```

Ex: use of filter()



```
> filter(mtcars, mpg>=27.3)
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4 79.0  66 4.08 1.935 18.90  1  1    4    1
Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
> 
```

Ex: