R Programming

UNIT - 2

Functions

Syntax:

```
func_name <- function (argument) {
statement
}
```

Example:

```
 pow <- function(x, y) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
```

Sample run:

```
>pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, 8)
[1] "2 raised to the power 8 is 256"
```

Named Arguments

Sample run:

```
> pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(x = 8, y = 2)
[1] "8 raised to the power 2 is 64"
> pow(y = 2, x = 8)
[1] "8 raised to the power 2 is 64"
```

Sample run:

```
> pow(x=8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, x=8)
[1] "8 raised to the power 2 is 64"
```

Default values for Arguments

Example:

```
pow <- function(x, y = 2) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
```

Sample run:

```
> pow(3)
[1] "3 raised to the power 2 is 9"
> pow(3,1)
[1] "3 raised to the power 1 is 3"
```

Return Value from Function

Syntax:

return(expression)

Example:

```
check <- function(x) {
if (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
else {
result <- "Zero"
}
return(result)
}
```

Sample run:

```
> check(1)
[1] "Positive"
> check(-10)
[1] "Negative"
> check(0)
[1] "Zero"
```

Functions without return()

Example:

```
check <- function(x) {
if (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
else {
result <- "Zero"
}
result
}
```

Example:

```
check <- function(x) {
if (x>0) {
return("Positive")
}
else if (x<0) {
return("Negative")
}
else {
return("Zero")
}
}
```

Multiple Returns

Example:

```
multi_return <- function() {
my_list <- list("color" = "red", "size" = 20, "shape" = "round")
return(my_list)
}
```

Sample run:

```
multi_return()
$color
[1] "red"

$size
[1] 20

$shape
[1] "round"
```

R Programming Environment

- R environment can be considered as a place to store and manage variables.
- Whenever an object or a function is created in R, an entry is added to the environment.
- Environment can be thought of as a collection of objects (functions, variables etc.)
- An environment is created when we first fire up the R interpreter. Any variable we define, is now in this environment.
- By default, the top-level environment is the R_GlobalEnv global environment
- Global environment can be referred to as .GlobalEnv in R codes as well.
- ls() function can be used to show what variables and functions are defined in the current environment.
- The environment() can also be used to get the current environment.

Example:

```
> a <- 2
> b <- 5
> f <- function(x) x<-0
> ls()
[1] "a" "b" "f "

> environment()
<environment: R_GlobalEnv>

> .GlobalEnv
<environment: R_GlobalEnv>
```

Cascading of environments

Example:

```
f <- function(f_x){
g <- function(g_x){
print("Inside g")
print(environment())
print(ls())
}
g(5)
print("Inside f")
print(environment())
print(ls())
}
```

Sample run:

```
> f(6)
[1] "Inside g"
<environment: 0x0000000010c2bdc8>
[1] "g_x"
[1] "Inside f"
<environment: 0x0000000010c2a870>
[1] "f_x" "g"

> environment()
<environment: R_GlobalEnv>

> ls() # On the console
[1] "f"
```

R Programming Scope

Example:

```
outer_func <- function(){
a <- 20
inner_func <- function(){
a <- 30
print(a)
}
inner_func()
print(a)
}
```

Sample Run:

```
> a <- 10
> outer_func()
[1] 30
[1] 20

> print(a)
[1] 10
```

R Programming Scope – Cont.

Example:

```
outer_func <- function(){
a<-20
inner_func <- function(){
a <<- 30 #Global variable assignment
print(a)
}
inner_func()
print(a)
}
```

Sample Run:

```
> outer_func()
[1] 30
[1] 30
> print(a)
[1] 30
```

Recursive Function

Example:

```
# Recursive function to find factorial
recursive.factorial <- function(x) {
if (x == 0)    return (1)
else  return (x * recursive.factorial(x-1))
}
```

Sample run:

```
> recursive.factorial(0)
[1] 1
> recursive.factorial(5)
[1] 120
> recursive.factorial(7)
[1] 5040
```

Infix Operator

- Most of the operators that we use in R are binary operators (having two operands). Hence, they are infix operators, used between the operands. Actually, these operators do a function call in the background.
- For example, the expression a+b is actually calling the function `+`() with the arguments a and b, as `+`(a, b).

Example:

```
> 5+3
[1] 8
> `+`(5,3) #operator within backtick or backquote
[1] 8
> 5-3
[1] 2
> `-`(5,3)
[1] 2
> 5*3-1
[1] 14
> `-`(`*`(5,3),1)
[1] 14
```

User defined Infix Operator

- It is possible to create user-defined infix operators in R. This is done by naming a function that starts and ends with %.

Example:

```
`%divisible%` <- function(x,y)
{
if (x%%y ==0) return (TRUE)
else  return (FALSE)
}
```

Sample Run:

```
> 10 %divisible% 3
[1] FALSE
> 10 %divisible% 2
[1] TRUE
> `%divisible%`(10,5)
[1] TRUE
```

Predefined infix operators

%%	Remainder operator
%/%	Integer division
%*%	Matrix multiplication
%o%	Outer product
%x%	Kronecker product
%in%	Matching operator

switch() function

Syntax:

switch (expression, list)

Example:

```
> switch(2,"red","green","blue")
[1] "green"
> switch(1,"red","green","blue")
[1] "red"
```

switch() function – Cont.

Examples:

```
> x <- switch(4,"red","green","blue")
> x
NULL

> x <- switch(0,"red","green","blue")
> x
NULL

> switch("color", "color" = "red", "shape" = "square", "length" = 5)
[1] "red"

> switch("length", "color" = "red", "shape" = "square", "length" = 5)
[1] 5
```

Vector

- Vector is a basic data structure in R. It contains element of the same type. The data types can be logical, integer, double, character, complex.
- Vectors are generally created using the c() function.

Creating Vectors

The keyword vector() is used to create a vector of a fixed type and fixed length.

```
vector ("numeric", 5) # numeric vector with 0 at every index
vector ("complex", 5) # complex vector with 0+0i at every index
vector ("logical", 5) # logical vector with FALSE at every index
vector ("character", 5) # character vector with "" at every index
```

```
[1] 0 0 0 0 0
[1] 0+0i 0+0i 0+0i 0+0i 0+0i
[1] FALSE FALSE FALSE FALSE FALSE
[1] "" "" "" "" ""
```

Creating Vectors by Concatenation

Example:

```
x<- c(1, 5, 4, 9, 0)
> typeof(x)
[1] "double"

> length(x)
[1] 5

> x <- c(1, 5.4, TRUE, "hello")
> x
[1] "1"     "5.4"   "TRUE"  "hello"
> typeof(x)
[1] "character"
```

```
> a <- 1
> is.vector(a)
[1] TRUE
```

Creating a vector using : operator

```
> x <- 1:7; x    #binary operaor
[1] 1 2 3 4 5 6 7

> y <- 2:-2; y
[1]  2  1  0 -1 -2

v <- 6.6:12.6; v
[1] 6.6 7.6 8.6 9.6 10.6 11.6 12.6
```

Creating a vector using **seq() function**

```
> seq(from=1,to=3, by=0.2)        # specify step size
[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0

> seq(1, 5, length.out=4)    # specify length of the vector
[1] 1.000000 2.333333 3.666667 5.000000
```

Create a sequence in Descending order of elements

```
additon<-function(n1,n2)
 {
  return(n1+n2)
 }

 subt<-function(n1,n2)
 {
  return(n1-n2)
 }

 mult<-function(n1,n2)
 {
  return(n1*n2)
 }

 div<-function(n1,n2)
 {
   return(n1/n2)
 }
```

```
calculator<-function()
{
   print("Enter your Choice")
   print("1.Addition 2.Subtraction 3.Multiplication 4.Division")
   choice <- as.integer(readline(prompt = ""))

   print("Enter the first number")
   n1 <- as.integer(readline(prompt = ""))
   print("Enter the second number")
   n2 <- as.integer(readline(prompt = ""))

   result<-switch(choice,addition(n1,n2),subt(n1,n2),mult(n1,n2),div(n1,n2))
   print("The result is ")
   print(result)
 }
```

```
> calculator()
[1] "Enter your Choice"
[1] "1.Addition 2.Subtraction 3.Multiplication 4.Division"
4
[1] "Enter the first number"
60
[1] "Enter the second number"
10
[1] "The result is "
[1] 6
```

Accessing the elements of a Vector

- Elements of a vector can be accessed using vector indexing. The vector used for indexing can be logical, integer or character vector.

Examples:

```
> x
[1]  0  2  4  6  8 10

> x[3]          # access 3rd element
[1] 4

> x[c(2, 4)]     # access 2nd and 4th element
[1] 2 6

> x[-1]          # access all but 1st element
[1]  2  4  6  8 10

> x[c(2, -4)]    # cannot mix positive and negative integers
Error in x[c(2, -4)]

> x[c(2.4, 3.54)]    # real numbers are truncated to integers
[1] 2 4
```

```
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
u <- t[c(2,3,6)]
print(u)

# Accessing vector elements using logical indexing.
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

# Accessing vector elements using 0/1 indexing.
y <- t[c(0,0,0,0,0,0,1)]
print(y)
```

```
[1] "Mon" "Tue" "Fri"
 [1] "Sun" "Fri"
 [1] "Sun" "Tue" "Wed" "Fri" "Sat"
 [1] "Sun"
```

Using logical expression as index

Examples:

```
> x
 [1] -3 -2 -1 0 1 2

> x[x < 0]  # filtering vectors based on conditions
[1] -3 -1

> x[x > 0]
[1] 3
```

Using character as index

Examples:

```
> x <- c("first"=3, "second"=0, "third"=9)
> names(x)
[1] "first"  "second" "third"

> x["second"]
second
0

> x[c("first", "third")]
first third
3     9
```

Modifying a vector

Examples:

```
> x
[1] -3 -2 -1  0  1  2

> x[2] <- 0; x        # modify 2nd element
[1] -3  0 -1  0  1  2

> x[x<0] <- 5; x   # modify elements less than 0
[1] 5 0 5 0 1 2

> x <- x[1:4]; x      # truncate x to first 4 elements
[1] 5 0 5 0
```

Inserting Elements in a Vector

```
myVector <- c(1, 2, 3, 4)
cat("Original Vector: ")
print(myVector)

myVector <- c(0, myVector)
cat("Appending 0 at the start of the vector: ")
print(myVector)

myVector <- c(myVector, 5)
cat("Appending 5 at the end of the vector: ")
print(myVector)

tempVector <- c(6, 7, 8)
myVector <- c(myVector, tempVector)
cat("Appending another vector at the end of the original vector: ")
print(myVector)
```

```
Original Vector: [1] 1 2 3 4
Appending 0 at the start of the vector: [1] 0 1 2 3 4
Appending 5 at the end of the vector: [1] 0 1 2 3 4 5
Appending another vector at the end of the original vector: [1] 0 1 2 3 4 5 6 7 8
```

Deleting a vector

Examples:

```
> x
[1] -3 -2 -1  0  1  2

> x <- NULL
> x
NULL

> x[4]
NULL
```

Operation on Vectors

Example:

```
> x <- c(2,8,3)
> y <- c(6,4,1)
> x+y
[1]  8 12  4
> x>y
[1] FALSE  TRUE  TRUE
```

Operation on Vectors

When there is a mismatch in length (number of elements) of operand vectors, the elements in shorter one is recycled in a cyclic manner to match the length of the longer one.

Examples:

```
> x <- c(2,1,8,3)
> y <- c(9,4)
> x+y # Element of y is recycled to 9,4,9,4
[1] 11  5 17  7

> x-1 # Scalar 1 is recycled to 1,1,1,1
[1] 1 0 7 2

> x+c(1,2,3)
[1]  3  3 11  4
Warning message:
In x + c(1, 2, 3) :
longer object length is not a multiple of shorter object length
```

Matrix

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout.
- They contain elements of the same atomic types.

Syntax

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

Following is the description of the parameters used −

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

Creating a matrix

Example:

```
> matrix(1:9, nrow = 3, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> # same result is obtained by providing only one dimension
> matrix(1:9, nrow = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Creating a matrix – Cont.

Example:

```
> matrix(1:9, nrow=3, byrow=TRUE)    # fill matrix row-wise
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> x <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"), c("A","B","C")))
> x
  A B C
X 1 4 7
Y 2 5 8
Z 3 6 9
```

Creating a matrix – Cont.

Example:

```
> colnames(x)
[1] "A" "B" "C"
> rownames(x)
[1] "X" "Y" "Z"

> # It is also possible to change names
> colnames(x) <- c("C1","C2","C3")
> rownames(x) <- c("R1","R2","R3")
> x
    C1 C2 C3
R1  1  4  7
R2  2  5  8
R3  3  6  9
```

Creating a matrix – Cont.

Example:

```
> cbind(c(1,2,3),c(4,5,6)) #Combine by colums
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(c(1,2,3),c(4,5,6)) #Combine by rows
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Creating a matrix – Cont.

Example:

```
> x <- c(1,2,3,4,5,6)
> x
[1] 1 2 3 4 5 6

>class(x)
[1] "numeric"

> dim(x) <- c(2,3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> class(x)
[1] "matrix"
```

Accessing the elements of a matrix

Example:

```
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> x[c(1,2),c(2,3)]    # select rows 1 & 2 and columns 2 & 3
     [,1] [,2]
[1,]    4    7
[2,]    5    8

> x[c(3,2),]    # leaving column field blank will select entire columns
     [,1] [,2] [,3]
[1,]    3    6    9
[2,]    2    5    8
```

Accessing the elements of a matrix

Example:

```
> x[,]    # leaving row as well as column field blank will select entire matrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> x[-1,]    # select all rows except first
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

Matrix

Example:

```
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> class(a)
[1] "matrix "

> attributes(a)
$dim
[1] 3 3

> dim(a)
[1] 3 3
```

Accessing the elements of a matrix

Example:

```
> x[1,]
[1] 1 4 7

> class(x[1,])
[1] "integer"

> x[1,,drop=FALSE]    # now the result is a 1X3 matrix rather than a vector
     [,1] [,2] [,3]
[1,]    1    4    7

> class(x[1,,drop=FALSE])
[1] "matrix"
```

Indexing a matrix with a single vector

Example:

```
> x
     [,1] [,2] [,3]
[1,]    4    8    3
[2,]    6    0    7
[3,]    1    2    9

> x[1:4]
[1] 4 6 1 8

> x[c(3,5,7)]
[1] 1 0 3
```

Using logical vector as index

```
> x
     [,1] [,2] [,3]
[1,]    4    8    3
[2,]    6    0    7
[3,]    1    2    9

> x[c(TRUE,FALSE,TRUE),c(TRUE,TRUE,FALSE)]
     [,1] [,2]
[1,]    4    8
[2,]    1    2

> x[c(TRUE,FALSE),c(2,3)]     # the 2 element logical vector is recycled to 3 element vector
     [,1] [,2]
[1,]    8    3
[2,]    2    9

> x[c(TRUE, FALSE)]
[1] 4 1 0 3 9

> x[x>5]     # select elements greater than 5
[1] 6 8 7 9

> x[x%%2 == 0]     # select even elements
[1] 4 6 8 0 2
```

Using character vector as index

Example:

```
> x
     A B C
[1,] 4 8 3
[2,] 6 0 7
[3,] 1 2 9

> x[,"A"]
[1] 4 6 1

> x[TRUE,c("A","C")]
     A C
[1,] 4 3
[2,] 6 7
[3,] 1 9

> x[2:3,c("A","C")]
     A C
[1,] 6 7
[2,] 1 9
```

Modifying a Matrix

Example:

```
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> x[2,2] <- 10; x    # modify a single element
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2   10    8
[3,]    3    6    9

> x[x<5] <- 0; x    # modify elements less than 5
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
[3,]    0    6    9
```

Modifying a Matrix

Example:

```
> t(x)    # transpose a matrix
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0   10    6
[3,]    7    8    9
```

```
> cbind(x, c(1, 2, 3))    # add column
     [,1] [,2] [,3] [,4]
[1,]    0    0    7    1
[2,]    0   10    8    2
[3,]    0    6    9    3
```

```
> rbind(x,c(1,2,3))    # add row
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
[3,]    0    6    9
[4,]    1    2    3
```

```
> x <- x[1:2,]; x    # remove last row
     [,1] [,2] [,3]
[1,]    0    0    7
[2,]    0   10    8
```

Modifying a Matrix

Example:

```
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> dim(x) <- c(3,2); x    # change to 3X2 matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> dim(x) <- c(1,6); x    # change to 1X6 matrix
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
```

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)

# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

```
# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

Lists

- List is a data structure having components of mixed data types.
- A vector having all elements of the same type is called atomic vector but a vector having elements of different type is called list.

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

```
[[1]]
 [1] "Shubha"

[[2]]
 [1] "Arpita "

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

 [[4]]
 [1] TRUE

 [[5]]
 [1] FALSE

 [[6]]
 [1] 22.5

 [[7]]
 [1] 12
```

```
# Creating a list containing a vector, a matrix and a list.
list_data <- list(c("Shubha","Nisha","Guna"), 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)
```

```
$Students
 [1] "Shubha" "Nisha" "Guna "

$Marks
      [,1] [,2] [,3]
 [1,]   40 60  90
 [2,]   80 70  80

 $Course

 $Course[[1]]
 [1] "BCA"

$Course[[2]]
 [1] "MCA"

$Course[[3]]
[1] "B. tech."
```

Accessing List Elements

```
myList <- list(1, 1+1i, "a", TRUE)
print(myList[1])
```

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

Each element in a list can be another list, so to obtain a single element use double square brackets[[]] instead

```
myList <- list(1, 1+1i, "a", TRUE)
print(myList[[1]])
```

```
[1] 1
```

```
x <- list(TRUE, 25, "Apple")
names(x) <- c("In Stock", "Quantity", "Product")
print(x$'In Stock')
print(x$Quantity)
print(x$Product)
```

```
[1] TRUE
[1] 25
[1] "Apple"
```

Manipulation of list elements

```
# 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 element at the end of the list.
list_data[4] <- "Bangalore "
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])
```

```
[[1]]
[1] "Bangalore"

 $<NA>
 NULL

$Course
[1] "Masters of computer applications"
```

Modifying a list

```
thislist <- list("apple", "banana", "cherry")
thislist[1] <- "blackcurrant"
thislist
```

```
[[1]]
[1] "blackcurrant"

[[2]]
[1] "banana"

[[3]]
[1] "cherry"
```

```
thislist <- list("apple", "banana", "cherry")

append(thislist, "orange", after = 2)
```

```
[[1]]
[1] "apple"

[[2]]
[1] "banana"

[[3]]
[1] "orange"

[[4]]
[1] "cherry"
```

```
thislist <- list("apple", "banana", "cherry")

newlist <- thislist[-1]

# Print the new list
newlist
```

```
[[1]]
[1] "banana"

[[2]]
[1] "cherry"
```

```
thislist <- list("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")

(thislist)[2:5]
```

```
[[1]]
[1] "banana"

[[2]]
[1] "cherry"

[[3]]
[1] "orange"

[[4]]
[1] "kiwi"
```

```
list1 <- list("a", "b", "c")
list2 <- list(1, 2, 3)
list3 <- c(list1,list2)

list3
```

```
[[1]]
[1] "a"

[[2]]
[1] "b"

[[3]]
[1] "c"

[[4]]
[1] 1

[[5]]
[1] 2

[[6]]
[1] 3
```

```
# Creating two lists.
Even_list <- list(2,4,6,8,10)
Odd_list <- list(1,3,5,7,9)

# Merging the two lists.
merged.list <- list(Even_list,Odd_list)

# Printing the merged list.
print(merged.list)
```

Data Frame

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.(Try giving the same name to two columns)
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

	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

Creating a Data Frame

```
# Create a, b, c, d variables
a <- c(10,20,30,40)
b <- c('book', 'pen', 'textbook', 'pencil_case')
c <- c(TRUE,FALSE,TRUE,FALSE)
d <- c(2.5, 8, 10, 7)
# Join the variables to create a data frame
df <- data.frame(a,b,c,d)
df
# Name the data frame
names(df) <- c('ID', 'items', 'store', 'price')
df
# Print the structure
str(df)
```

```
   a           b       c     d
1  10         book    TRUE   2.5
2  20          pen   FALSE   8.0
3  30     textbook    TRUE  10.0
4  40 pencil_case    ALSE   7.0
```

```
  ID       items  store price
1 10        book   TRUE   2.5
2 20         pen  FALSE   8.0
3 30    textbook   TRUE  10.0
4 40 pencil_case  FALSE   7.0
```

```
'data.frame':	4 obs. of  4 variables:
 $ ID   : num  10 20 30 40
 $ items: chr  "book" "pen" "textbook" "pencil_case"
 $ store: logi  TRUE FALSE TRUE FALSE
 $ price: num  2.5 8 10 7
```

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame
```

```
   Training  Pulse  Duration
1  Strength    100        60
2   Stamina    150        30
3     Other    120        45
```

Summarize the Data

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame

summary(Data_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
```

Accessing the items of a Data Frame

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame[1]

Data_Frame[["Training"]]

Data_Frame$Training
```

```
  Training
1 Strength
2 Stamina
3    Other
```

```
[1] Strength Stamina  Other
```

```
[1] Strength Stamina  Other
```

Add Rows

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new row
New_row_DF <- rbind(Data_Frame, c("Strength", 110, 110))

# Print the new row
New_row_DF
```

	Training	Pulse	Duration
1	Strength	100	60
2	Stamina	150	30
3	Other	120	45
4	Strength	110	110

Add Columns

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new column
New_col_DF <- cbind(Data_Frame, Steps = c(1000, 6000, 2000))

# Print the new column
New_col_DF
```

	Training	Pulse	Duration	Steps
1	Strength	100	60	1000
2	Stamina	150	30	6000
3	Other	120	45	2000

Remove Rows and Columns

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Remove the first row and column
Data_Frame_New <- Data_Frame[c(-1), c(-1)]

# Print the new data frame
Data_Frame_New
```

	Pulse	Duration
2	150	30
3	120	45

Dimension of Dataframe

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

dim(Data_Frame)
```

```
[1] 3 3
```

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

ncol(Data_Frame)
nrow(Data_Frame)
```

Try using length() function

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

Combining Data Frames

```
Data_Frame1 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame2 <- data.frame (
  Training = c("Stamina", "Stamina", "Strength"),
  Pulse = c(140, 150, 160),
  Duration = c(30, 30, 20)
)

New_Data_Frame <- rbind(Data_Frame1, Data_Frame2)
New_Data_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

```
Data_Frame3 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)

New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4)
New_Data_Frame1
```

	Training	Pulse	Duration	Steps	Calories
1	Strength	100	60	3000	300
2	Stamina	150	30	6000	400
3	Other	120	45	2000	300

Data Slicing

```
   ID       items  store price
1  10        book   TRUE   2.5
2  20         pen  FALSE   8.0
3  30    textbook   TRUE  10.0
4  40 pencil_case  FALSE   7.0
```

Select row 1 in column 2
```
df[1,2]
```

```
[1] book
```

Select Rows 1 to 2
```
df[1:2,]
```

```
  ID items  store  price
1 10  book   TRUE    2.5
2 20   pen  FALSE    8.0
```

Select Columns 1
```
df[,1]
```

```
[1] 10 20 30 40
```

#Select Rows 1 to 3 and columns 3 to 4
```
df[1:3, 3:4]
```

```
   store  price
1   TRUE    2.5
2  FALSE    8.0
3   TRUE   10.0
```

#Slice with column name
```
df[, c('ID', 'store')]
```

```
  ID  store
1 10  TRUE
2 20 FALSE
3 30  TRUE
4 40 FALSE
```

```
# Select price above 5
subset(df, subset = price > 5)
```

	ID	items	store	price
2	20	pen	FALSE	8
3	30	textbook	TRUE	10
4	40	pencil_case	FALSE	7

Factors

- Factors are used to represent categorical data.
- Useful in the columns which have a limited number of unique values. Ex: "Male, "Female" and True, False etc.
- Useful in data analysis for statistical modeling.

```
data <- c("East","West","East","North","North","East","West","West","West","East","North")

print(data)
print(is.factor(data))

# Apply the factor function.
factor_data <- factor(data)

print(factor_data)
print(is.factor(factor_data))
```

```
[1] "East"  "West"  "East"  "North" "North" "East"  "West"  "West"  "West"  "East"
"North"

[1] FALSE

 [1] East  West  East  North North East  West  West  West  East  North
Levels: East North West

[1] TRUE
```

```
# Create the vectors for data frame.
height <- c(132,151,162,139,166,147,122)
weight <- c(48,49,66,53,67,52,40)
gender <- c("male","male","female","female","male","female","male")

# Create the data frame.
input_data <- data.frame(height,weight,gender)
print(input_data)

# Test if the gender column is a factor.
print(is.factor(input_data$gender))

# Print the gender column so see the levels.
print(input_data$gender)
```

```
  height weight gender
1    132     48   male
2    151     49   male
3    162     66 female
4    139     53 female
5    166     67   male
6    147     52 female
7    122     40   male
[1] FALSE
[1] "male"   "male"   "female" "female" "male"   "female" "male"
[1] TRUE
```

```
input_data[, 'gender'] <- as.factor(input_data[, 'gender'])
print(is.factor(input_data$gender))
```

Changing the Order of Levels

```
data <- c("East","West","East","North","North","East","West","West","West","East","North")
# Create the factors
factor_data <- factor(data)
print(factor_data)
print(levels(factor_data))
print(nlevels(factor_data))

# Apply the factor function with required order of the level.
new_order_data <- factor(factor_data, levels = c("East","West","North"))
print(new_order_data)
```

```
 [1] East  West  East  North North East  West  West  West  East  North
 Levels: East North West
 [1] "East"  "North" "West"
 [1] 3
  [1] East  West  East  North North East  West  West  West  East  North
 Levels: "East" "West" "North"
```

Accessing elements of a Factor

```
gender <- factor(c("female", "male", "male", "female"));
gender[3]
```

```
[1] male
Levels: female male
```

```
gender <- factor(c("female", "male", "male", "female"));
gender[c(2, 4)]
```

```
[1] male   female
Levels: female male
```

Modifying a Factor

```
gender <- factor(c("female", "male", "male", "female"));
gender[2]<-"female"
gender
```

```
[1] female female male   female
Levels: female male
```

To add a new level

```
levels(gender) <- c(levels(gender),"other")
gender[1]<-"other"
print(gender)
```

```
[1] other  female male   female
Levels: female male other
```

Simple Calculator

```
# Program make a simple calculator that can add, subtract, multiply and divide using functions
add <- function(x, y) {
return(x + y)
}
subtract <- function(x, y) {
return(x - y)
}
multiply <- function(x, y) {
return(x * y)
}
divide <- function(x, y) {
return(x / y)
}
```

```
# take input from the user
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
choice = as.integer(readline(prompt="Enter choice[1/2/3/4]: "))
num1 = as.integer(readline(prompt="Enter first number: "))
num2 = as.integer(readline(prompt="Enter second number: "))
operator <- switch(choice,"+","-","*","/")
result <- switch(choice, add(num1, num2), subtract(num1, num2), multiply(num1, num2),
divide(num1, num2))
print(paste(num1, operator, num2, "=", result))
```

Sample Run:

```
[1] "Select operation."
[1] "1.Add"
[1] "2.Subtract"
[1] "3.Multiply"
[1] "4.Divide"
Enter choice[1/2/3/4]: 4
Enter first number: 20
Enter second number: 4
[1] "20 / 4 = 5"
```