UNIT - III Vectors: Creating and Naming Vectors, Vector Arithmetic, Vector sub setting, Matrices: Creating and Naming Matrices, Matrix Sub setting, Arrays, Class. Factors and Data Frames: Introduction to Factors: Factor Levels, Summarizing a Factor, Ordered Factors, Comparing Ordered Factors, Introduction to Data Frame, subsetting of Data Frames, Extending Data Frames, Sorting Data Frames. Lists: Introduction, creating a List: Creating a Named List, Accessing List Elements, Manipulating List Elements, Merging Lists, Converting Lists to Vectors

Vectors

- A vector is simply a list of items that are of the same type.

- A **vector** is a basic data structure which plays an important role in R programming.
- 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.

- To combine the list of items to a vector, use the c() function and separate the items by a comma.
- Vectors in R are the same as the arrays in C language which are used to hold multiple data values of the same type. One major key point is that in R the indexing of the vector will start from '1' and not from '0'. We can create numeric vectors and character vectors as well.
- The length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.

Types of vectors

- Vectors are of different types which are used in R.

1.Numeric vectors

Numeric vectors are those which contain numeric values such as integer, float, etc.

Ex: 1

```
# R program to create numeric Vectors
# creation of vectors using c() function.
v1 <- c(4, 5, 6, 7)
# display type of vector
typeof(v1)
# by using 'L' we can specify that we want
integer values.
v2 <- c(1L, 4L, 2L, 5L)
# display type of vector
typeof(v2)
```

O/P: [1] "double"
[1] "integer"

EX: 2

- **Character vectors**

  Character vectors contain alphanumeric values and special characters.

  - **# R program to create Character Vectors**
  - **# by default numeric values**
  - **# are converted into characters**
  - **v1 <- c('geeks', '2', 'hello', 57)**
  - **# Displaying type of vector**
  - **typeof(v1)**

  Output:

- [1] "character"

Ex :3

Logical vectors

Logical vectors contain boolean values such as TRUE, FALSE and NA for Null values.

- **# R program to create Logical Vectors**
- **# Creating logical vector**
- **# using c() function**
- **v1 <- c(TRUE, FALSE, TRUE, NA)**
- **# Displaying type of vector**
- **typeof(v1)**

Output:

- [1] "logical"

Creating and Naming Vectors

- we use c() function to create a vector. This function returns a one-dimensional array or simply vector. The c() function is a generic function which combines its argument.
- All arguments are restricted with a common data type which is the type of the returned value. There are various other ways also there

Types:

1. Using c() Function
2. Using the colon(:) operator
3. Using the seq() function
4. Using assign() function

Using c() Function

- The c function in R programming stands for 'combine. ' This function is used to get the output by giving parameters inside the function.

EX: 1

```
# R program to create Vectors
# we can use the c function
# to combine the values as a vector.
# By default the type will be double
X <- c(61, 4, 21, 67, 89, 2)
cat('using c function', X)
# print the vales option 1
X
# print the vales option 2
# print(X)
```

O/P: using c function 61 4 21 67 89 2

2. Using the colon(:) operator

- Colon operator (":") in R is **a function that generates regular sequences**. It is most commonly used in for loops, to index and to create a vector with increasing or decreasing sequence. It is a binary operator i.e. it takes two arguments.

Syntax : z<- **x:y**

EX: 1

✓ # Vector with numerical values in a sequence

```
numbers <- 1:10

numbers
```

EX: 2

```
# Vector with numerical decimals in a sequence
numbers1 <- 1.5:6.5
numbers1
```

3. **Using the seq() function**

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

Example:

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

Output

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
[1] "numeric"
```

EX: 2

```
seq_vec<-seq(1,4,length.out=10)
seq_vec
class(seq_vec)
```

Output

```
[1]  1.000000  1.333333  1.666667  2.000000  2.333333  2.666667
3.000000 3.333333
 [9] 3.666667 4.000000
[1] "numeric"
```

EX: 3

```
seq_vec<-seq(1,4,length.out=5)
seq_vec
class(seq_vec)
```

Output

```
[1] 1.00 1.75 2.50 3.25 4.00
[1] "numeric"
```

EX:4

```
# Creating a sequence from 5 to 13.
v <- 5:13
print(v)
# Creating a sequence from 6.6 to 12.6.
v <- 6.6:12.6
print(v)
```

EX:

```
# R program to illustrate
# Assigning vectors
# Assigning a vector using
# seq() function
V = seq(1, 3, by=0.2)

# Printing the vector

print(V)
```

Output

```
[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

4 Using assign() function

- The assign() function takes the following mandatory parameter values: **x : This represents the variable name that is given as a character string. value : This is the value to be assigned to the x variable**.
- EX:1

✓ **assign("vec2",c(6,7,8,9,10))**

✓ **vec2**

Output

```
[1]  6  7  8  9 10
```

Vector Arithmetic Operations

- We can perform arithmetic operations on vectors, like addition, subtraction, multiplication and division.
- Please note that the two vectors should be of same length and same type. Or one of the vectors can be an atomic value of same type.
- If the vectors are not of same length, then **Vector Recycling** happens implicitly.
- **Vector Recycling:**
- If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors u and v have different lengths, and their sum is computed by recycling values of the shorter vector u.
- > **u = c(10, 20, 30)**
  > v = c(1, 2, 3, 4, 5, 6, 7, 8, 9)
  > u + v
  [1] 11 22 33 14 25 36 17 28 39

Types:

1. Addition

- Addition operator takes two vectors as operands, and returns the result of sum of two vectors.

a + b

Example

In the following program, we create two integer vectors and add them using Addition Operator.

Ex:1

- ✓ **a <- c(10, 20, 30, 40, 50)**
- ✓ **b <- c(1, 3, 5, 7, 9)**
- ✓ **result <- a + b**

- ✓ **print(result)**

Output

[1] 11 23 35 47 59

2. Subtraction

- Subtraction operator takes two vectors as operands, and returns the result of difference of two vectors.

a - b

Example

In the following program, we create two integer vectors and find their different using Subtraction Operator.

Ex:1

- ✓ **a <- c(10, 20, 30, 40, 50)**
- ✓ **b <- c(1, 3, 5, 7, 9)**
- ✓ **result <- a - b**
- ✓ **print(result)**

Output

[1] 9 17 25 33 41

3.Multiplication

- Multiplication operator takes two vectors as operands, and returns the result of product of two vectors.

a * b

Example

- In the following program, we create two integer vectors and find their product using Multiplication Operator.

Ex:

- ✓ **a <- c(10, 20, 30, 40, 50)**
- ✓ **b <- c(1, 3, 5, 7, 9)**
- ✓ **result <- a * b**
- ✓ **print(result)**

Output

[1] 10 60 150 280 450

4.Division

- Division operator takes two vectors are operands, and returns the result of division of two vectors.

a + b

Example

In the following program, we create two integer vectors and divide them using Division Operator.

Example.R

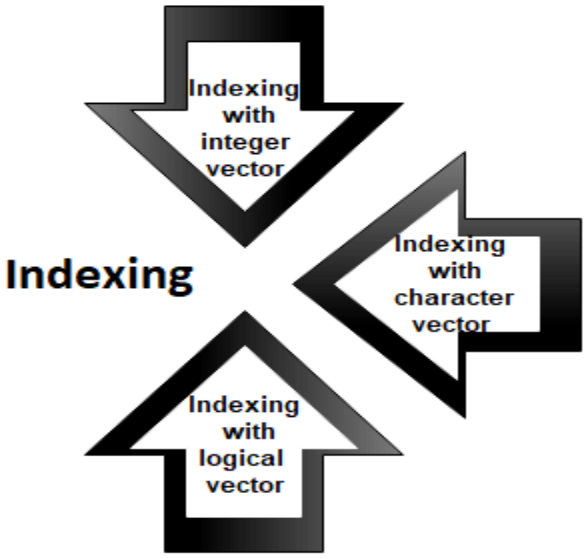
- ✓ **a <- c(10, 20, 30, 40, 50)**
- ✓ **b <- c(1, 3, 5, 7, 9)**
- ✓ **result <- a / b**
- ✓ **print(result)**

Output

[1] 10.000000 6.666667 6.000000 5.714286 5.555556

Accessing elements of vectors

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



1) Indexing with integer vector

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

Example:

```
1. seq_vec<-seq(1,4,length.out=6)
2. seq_vec
3. seq_vec[2]
```

Output

```
[1] 1.0 1.6 2.2 2.8 3.4 4.0
[1] 1.6
```

2) Indexing with a character vector

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

Example:

```
1. char_vec<-c("shubham"=22,"arpita"=23,"vaishali"=25)
2. char_vec
3. char_vec["arpita"]
```

Output

```
shubham   arpita vaishali
   22      23     25
arpita
     23
```

3) Indexing with a logical vector

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

Example:

```
1. a<-c(1,2,3,4,5,6)
2. a[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE)]
```

Output

```
[1] 1 3 4 6
```

Vector sub setting

- Vectors are basic objects in R and they can be subsetted using the **[operator**.
- EX:

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]    ## Extract the first element
[1] "a"
```

- EX

```
> x[2]    ## Extract the second element
[1] "b"
```

- **The [operator** can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a" "b" "c" "c"
```

- The sequence does not have to be in order; you can specify any arbitrary integer vector.

- EX

```
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

Matrices: Creating and Naming Matrices

- In R, a two-dimensional rectangular data set is known as a matrix. A matrix is created with the help of the vector input to the matrix function. On R matrices, we can perform addition, subtraction, multiplication, and division operation.
- In the R matrix, elements are arranged in a fixed number of rows and columns. The matrix elements are the real numbers. In R, we use matrix function, which can easily reproduce the memory representation of the matrix. In the R matrix, all the elements must share a common basic type.

- **To create a matrix in R you need to use the function called matrix()**. The arguments to this matrix() are the set of elements in the vector.
- You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix. Note: By default, matrices are in column-wise order

- 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. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.
- A Matrix is created using the matrix() function.

  Syntax
- The basic syntax for creating a matrix in R is −
- **matrix(data, nrow, ncol, byrow, dimnames)**

Note:

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.(byrow is a logical variable. Matrices are by default column-wise. By setting byrow as TRUE, we can arrange the data row-wise in the matrix)
- **dimname** is the names assigned to the rows and columns(takes two character arrays as input for row names and column names).

How to create a matrix in R:

- Like vector and list, R provides a function which creates a matrix. R provides the matrix() function to create a matrix. This function plays an important role in data analysis. There is the following syntax of the matrix in R:

1. matrix(data, nrow, ncol, byrow, dim_name)

data

The first argument in matrix function is data. It is the input vector which is the data elements of the matrix.

nrow

The second argument is the number of rows which we want to create in the matrix.

ncol

The third argument is the number of columns which we want to create in the matrix.

byrow

The byrow parameter is a logical clue. If its value is true, then the input vector elements are arranged by row.

dim_name

The dim_name parameter is the name assigned to the rows and columns.

Example to understand how matrix function is used to create a matrix and arrange the elements sequentially by row or column.

EX:

- ✓ **P <- matrix(c(5:16), nrow = 4, byrow = TRUE)**
- ✓ **print(P)**
- ✓ **# Arranging elements sequentially by column.**
- ✓ **Q <- matrix(c(3:14), nrow = 4, byrow = FALSE)**
- ✓ **print(Q)**
- ✓ **# Defining the column and row names.**
- ✓ **row_names = c("row1", "row2", "row3", "row4")**
- ✓ **col_names = c("col1", "col2", "col3")**
- ✓ **R <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))**
- ✓ **print(R)**

Output

```
     [,1] [,2] [,3]
[1,]    5    6    7
[2,]    8    9   10
[3,]   11   12   13
```

```
[4,]   14   15   16
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
```

Accessing matrix elements in R

There are three ways to access the elements from the matrix.

1. We can access the element which presents on nth row and mth column.
2. We can access all the elements of the matrix which are present on the nth row.
3. We can also access all the elements of the matrix which are present on the mth column.

Example

```
# Defining the column and row names.
row_names = c("row1", "row2", "row3", "row4")
ccol_names = c("col1", "col2", "col3")
#Creating matrix
R <-
 matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names
, col_names))
print(R)
```

- ✓ **#Accessing element present on 3rd row and 2nd column**
- ✓ **print(R[3,2])**

- ✓ **#Accessing element present in 3rd row**
- ✓ **print(R[3,])**

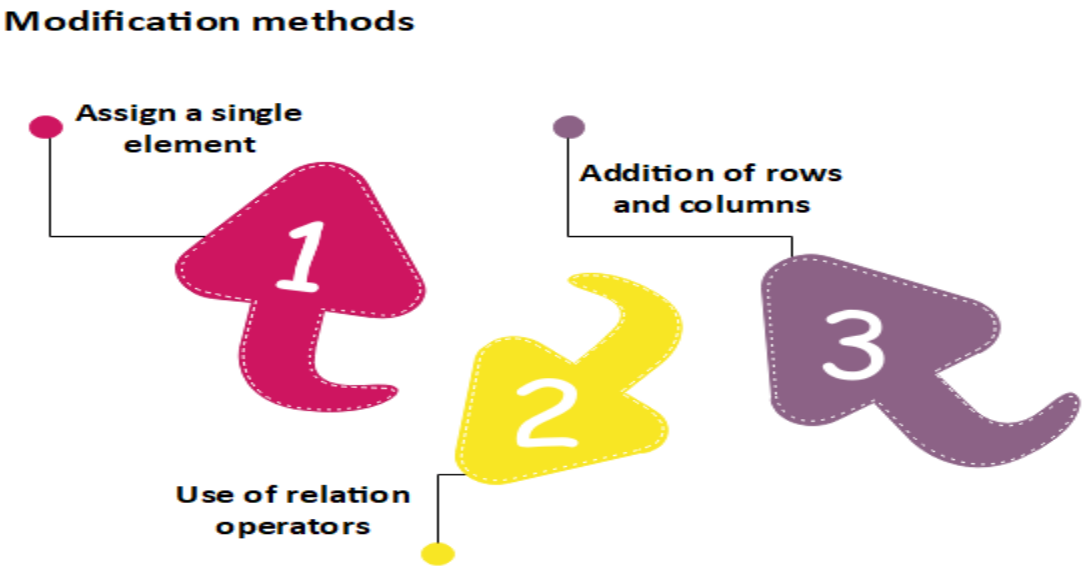
- ✓ **#Accessing element present in 2nd column**
- ✓ **print(R[,2])**

Output

```
     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16

[1] 12

col1 col2 col3
  11   12   13

row1 row2 row3 row4
   6    9   12   15
```

Modification of the matrix

R allows us to do modification in the matrix. There are several methods to do modification in the matrix, which are as follows:



Assign a single element

- In matrix modification, the first method is to assign a single element to the matrix at a particular position. By assigning a new value to that position, the old value will get replaced with the new one. This modification technique is quite simple to perform matrix modification. The basic syntax for it is as follows:

1. matrix[n, m]**<-y**

Here, n and m are the rows and columns of the element, respectively. And, y is the value which we assign to modify our matrix.

Example

- ✓ **# Defining the column and row names.**
- ✓ **row_names = c("row1", "row2", "row3", "row4")**
- ✓ **ccol_names = c("col1", "col2", "col3")**

- ✓ **R <-**
  matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names , col_names))
- ✓ **print(R)**

- ✓ **#Assigning value 20 to the element at 3d roe and 2nd column**
- ✓ **R[3,2]<-20**
- ✓ **print(R)**

Output

```
     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16

     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   20   13
row4   14   15   16
```

Use of Relational Operator

- R provides another way to perform matrix medication. In this method, we used some relational operators like >, <, ==. Like the first method, the second method is quite simple to use. Let see an example to understand how this method modifies the matrix.

Example 1

```
# Defining the column and row names.
row_names = c("row1", "row2", "row3", "row4")
ccol_names = c("col1", "col2", "col3")
R <-
 matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names
, col_names))
print(R)
#Replacing element that equal to the 12
R[R==12]<-0
print(R)
```

Output

```
     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16

     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11    0   13
row4   14   15   16
```

Example 2

```
# Defining the column and row names.
row_names = c("row1", "row2", "row3", "row4")
ccol_names = c("col1", "col2", "col3")
R <-
 matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names
, col_names))
print(R)
#Replacing elements whose values are greater than 12
R[R>12]<-0
print(R)
```

Output

```
     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16

     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12    0
row4    0    0    0
```

Addition of Rows and Columns

- The third method of matrix modification is through the addition of rows and columns using the cbind() and rbind() function. The cbind() and

rbind() function are used to add a column and a row respectively. Let see an example to understand the working of cbind() and rbind() functions.

Example 1

```
# Defining the column and row names.
row_names = c("row1", "row2", "row3", "row4")
ccol_names = c("col1", "col2", "col3")

R <-
 matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names
, col_names))
print(R)

#Adding row
rbind(R,c(17,18,19))

#Adding column
cbind(R,c(17,18,19,20))

#transpose of the matrix using the t() function:
t(R)

#Modifying the dimension of the matrix using the dim() function
dim(R)<-c(1,12)
print(R)
```

Output

```
     col1 col2 col3
row1    5    6    7
```

```
row2    8    9   10
row3   11   12   13
row4   14   15   16

     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16
          17   18   19

     col1 col2 col3
row1    5    6    7 17
row2    8    9   10 18
row3   11   12   13 19
row4   14   15   16 20

     row1 row2 row3 row4
col1    5    8   11   14
col2    6    9   12   15
col3    7   10   13   16

    [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    5    8   11   14    6    9   12   15    7    10    13    16
```

Matrix operations

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

operation on the matrix, it is required that both the matrix should have the same dimensions.

Example 1

- ✓ **R <- matrix(c(5:16), nrow = 4,ncol=3)**
- ✓ **S <- matrix(c(1:12), nrow = 4,ncol=3)**

- ✓ **#Addition**
- ✓ **sum<-R+S**
- ✓ **print(sum)**

- ✓ **#Subtraction**
- ✓ **sub<-R-S**
- ✓ **print(sub)**

- ✓ **#Multiplication**
- ✓ **mul<-R*S**
- ✓ **print(mul)**

- ✓ **#Multiplication by constant**

- ✓ **mul1<-R*12**
- ✓ **print(mul1)**

- ✓ **#Division**
- ✓ **div<-R/S**
- ✓ **print(div)**

Output

```
     [,1] [,2] [,3]
[1,]    6   14   22
[2,]    8   16   24
[3,]   10   18   26
[4,]   12   20   28

     [,1] [,2] [,3]
[1,]    4    4    4
[2,]    4    4    4
[3,]    4    4    4
[4,]    4    4    4

     [,1] [,2] [,3]
[1,]    5   45  117
[2,]   12   60  140
[3,]   21   77  165
[4,]   32   96  192

     [,1] [,2] [,3]
[1,]   60  108  156
[2,]   72  120  168
[3,]   84  132  180
```

```
[4,]   96  144  192


     [,1]     [,2]      [,3]
[1,] 5.000000 1.800000 1.444444
[2,] 3.000000 1.666667 1.400000
[3,] 2.333333 1.571429 1.363636
[4,] 2.000000 1.500000 1.333333
```

Applications of matrix

1. In geology, Matrices takes surveys and plot graphs, statistics, and used to study in different fields.
2. Matrix is the representation method which helps in plotting common survey things.
3. In robotics and automation, Matrices have the topmost elements for the robot movements.
4. Matrices are mainly used in calculating the gross domestic products in Economics, and it also helps in calculating the capability of goods and products.
5. In computer-based application, matrices play a crucial role in the creation of realistic seeming motion.

Matrix subsetting

- A matrix is subset with two arguments within single brackets, [], and separated by a comma. The first argument specifies the rows, and the second the columns.

EX:

- ✓ **a <- matrix(1:9, nrow = 3)**
- ✓ **a**
- ✓ **colnames(a) <- c("A", "B", "C")**
- ✓ **a[1:2,]**

Output

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
     A B C
[1,] 1 4 7
[2,] 2 5 8
```

Ex: 2

- ✓ **a <- matrix(1:9, nrow = 3)**
- ✓ **a**
- ✓ **colnames(a) <- c("A", "B", "C")**
- ✓ **a[1:3,]**

Output

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

```
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

Note:

The only difference between vectors, matrices, and arrays are

- **Vectors are uni-dimensional arrays**
- **Matrices are two-dimensional arrays**
- **Arrays can have more than two dimensions**

R Arrays

- In R, arrays are the data objects which allow us to store data in more than two dimensions. In R, an array is created with the help of the **array()** function. This array() function takes a vector as an input and to create an array it uses vectors values in the **dim** parameter.
- **For example**- if we will create an array of dimension (2, 3, 4) then it will create 4 rectangular matrices of 2 row and 3 columns.

R Array Syntax

There is the following syntax of R arrays:

1. **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**

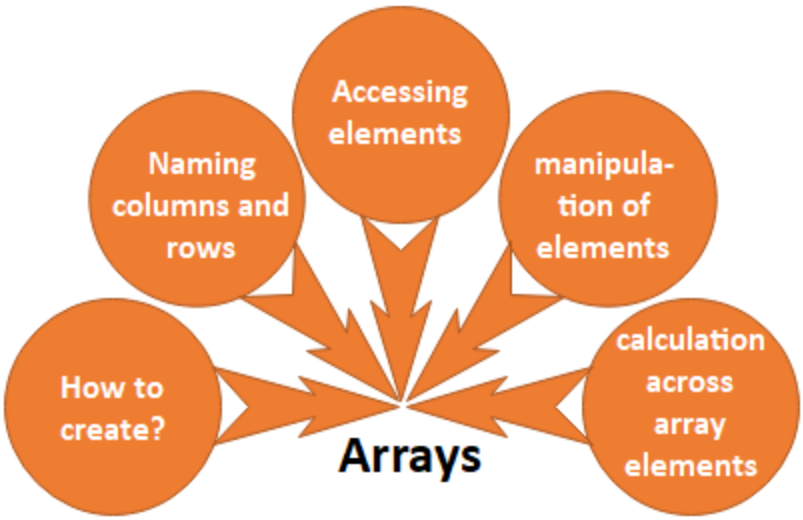
This parameter defines the number of row elements which an array can store.

- **column_size**

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

- **dim_names**

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



TYPES

Uni-Dimensional Array

- A vector is a uni-dimensional array, which is specified by a single dimension, length. A Vector can be created using '**c()**' function. A list of values is passed to the c() function to create a vector.
- EX:

```
vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)

print (vec1)

 # cat is used to concatenate

# strings and print it.

cat ("Length of vector : ", length(vec1))
```

Output

```
[1] 1 2 3 4 5 6 7 8 9
Length of vector :  9
```

Multi-Dimensional Array

- A two-dimensional matrix is an array specified by a fixed number of rows and columns, each containing the same data type. A matrix is created by using **array()** function to which the values and the dimensions are passed.
- EX:1

```
# arranges data from 2 to 13

# in two matrices of dimensions 2x3

arr = array(2:13, dim = c(2, 3, 2))

print(arr)
```

Output

```
, , 1


     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    5    7


, , 2
```

```
     [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13
```

Creation of arrays

- In R, array creation is quite simple. We can easily create an array using 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
vec1 <-c(1,3,5)
vec2 <-c(10,11,12,13,14,15)
 #Taking these vectors as input to the array
res <- array(c(vec1,vec2))
res[1]
print(res)
```

Output

```
[1] 1
[1]  1  3  5 10 11 12 13 14 15
```

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.
- ✓ It is not necessary to give the name to the rows and columns. It is only used to differentiate the row and column for better understanding.
- ✓ EX:1

```
#Creating two vectors of different lengths

vec1 <-c(1,3,5)

vec2 <-c(10,11,12,13,14,15)

#Initializing names for rows, columns and matrices

col_names <- c("Col1","Col2","Col3")

row_names <- c("Row1","Row2","Row3")

matrix_names <- c("Matrix1","Matrix2")

#Taking the vectors as input to the array
```

```
Res <-
array(c(vec1,vec2),dim=c(3,3,2),dimnames=list(row_names,col_names,matrix_names))

print(res)
```

Output

```
, , 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 arrays

- The arrays can be accessed by using indices for different dimensions separated by commas. Different components can be specified by any combination of elements' names or positions.

Accessing Uni-Dimensional Array

The elements can be accessed by using indexes of the corresponding elements.

EX:1

```
vec <- c(1:10)

 # accessing entire vector

cat ("Vector is : ", vec)

 # accessing elements

cat ("Third element of vector is : ", vec[3])
```

Output

Vector is : 1 2 3 4 5 6 7 8 9 10Third element of vector is : 3

Access Entire Row or Column

EX: 1

```
# create a two 2 by 3 matrix
array1 <- array(c(1:12), dim = c(2,3,2))

print(array1)
```

```
# access entire elements at 2nd column of 1st matrix
cat("\n2nd Column Elements of 1st matrix:", array1[,c(2),1])

# access entire elements at 1st row of 2nd matrix
cat("\n1st Row Elements of 2nd Matrix:", array1[c(1),,2])
```

Output

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

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


2nd Column Elements of 1st matrix: 3 4
1st Row Elements of 2nd Matrix: 7 9 11
```

Manipulating Array Elements

- As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.
- EX: 1

```
# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.

array1 <- array(c(vector1,vector2),dim = c(3,3,2))

array1

# Create two vectors of different lengths.

vector3 <- c(9,1,0)

vector4 <- c(6,0,11,3,14,1,2,6,9)

array2 <- array(c(vector1,vector2),dim = c(3,3,2))

array2

# create matrices from these arrays.
```

```
matrix1 <- array1[,,2]

matrix2 <- array2[,,2]

# Add the matrices.

result <- matrix1+matrix2

print(result)
```

Output

```
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 1
```

```
     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

     [,1] [,2] [,3]
[1,]   10   20   26
[2,]   18   22   28
[3,]    6   24   30
```

Calculations across array elements

- For calculation purpose, r provides **apply()** function. This apply function contains three parameters i.e., x, margin, and function.
- This function takes the array on which we have to perform the calculations.

Using apply() function

- 'apply()' is one of the R packages which have several functions that helps to write code in an easier and efficient way. You'll see the example below where it can be used to calculate the sum of two different arrays.

The syntax for apply() is :

apply(x, margin, function)

The argument above indicates that:
x**: An array or two-dimensional data as matrices**.
margin**: Indicates a function to be applied as margin value to be c(1) for rows, c(2) for columns, and c(1,2) for both rows and columns.**
function: Indicates the R- built-in or user-defined function to be applied over the given data.

EX: 1

```
#Creating two vectors of different lengths

vec1 <-c(1,3,5)

vec2 <-c(10,11,12,13,14,15)
```

```
#Taking the vectors as input to the array1

res1 <- array(c(vec1,vec2),dim=c(3,3,2))

print(res1)


#using apply function

result <- apply(res1,c(1),sum)

print(result)
```

Output

```
, , 1


     [,1] [,2] [,3]
[1,]    1   10   13
[2,]    3   11   14
[3,]    5   12   15

, , 2


     [,1] [,2] [,3]
[1,]    1   10   13
[2,]    3   11   14
[3,]    5   12   15
```

```
[1] 48 56 64
```

Output

```
, , 1

     [,1] [,2] [,3]
[1,]    1   10   13
[2,]    3   11   14
[3,]    5   12   15

, , 2

     [,1] [,2] [,3]
[1,]    1   10   13
[2,]    3   11   14
[3,]    5   12   15

[1] 18 66 84
```

EX: 2

```
#Creating two vectors of different lengths

vec1 <-c(1,3,5)
```

```
vec2 <-c(10,11,12,13,14,15)


#Taking the vectors as input to the array1

res1 <- array(c(vec1,vec2),dim=c(3,3,2))

print(res1)

#using apply function

result <- apply(res1,c(1,2),sum)

print(result)
```

Output

```
, , 1

     [,1] [,2] [,3]
[1,]    1   10   13
[2,]    3   11   14
[3,]    5   12   15

, , 2

     [,1] [,2] [,3]
[1,]    1   10   13
```

```
[2,]    3   11   14
[3,]    5   12   15


     [,1] [,2] [,3]
[1,]    2   20   26
[2,]    6   22   28
[3,]   10   24   30
```