Creating a Matrix: Matrix is a two dimension of data structure in RP all attributes of objects can be checked with the attributes() i.e (dimnames) can be checked directly with the dim(). we can check if a variable matrix or not with the class()

eg:- > a

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

> class(a)

o/p: "matrix"

> attributes (a) (variables)

$dim o/p: 2 2

dim(a) o/p: 2 2

How to create a matrix in RP: It can be created by using matrix() dimensions of the matrix can be defined by passing appropriate values for arguments nrows & ncolumns

eg:- > matrix (1:4, nrow = 2, ncol = 2) (or)

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

matrix (1:4, nrow=2)

- Same as matrix(1:4, nrow=2) by deafult ncol=2
- Insert value row wise take bycol=TRUE. A matrix is stored in colowmn may as order internally. It is possible to name the rows and colowmns of matrix during creation by passing a two elements list to the argument.

It is also possible to change colowmns names

eg:- > x ← matrix (1:9, nrow = 3, dimnames = list (c("x","y","z"), c("A","B","C"))

> colnames (x) ← c ("c1","c2","c3")

> rownames(x) ← c("R1","R2","R3")

o/p:

	c1	c2	c3
R1	1	4	7
R2	2	5	8
R3	3	6	9

General matrix Operations: Matrix computations are same as the general matrix operations in that matrix addition, subtraction, multiplication, division and accessing elements of a matrix are covered in unit 1

Modify a matrix in R: use assignment operator to modify matrix elements

eg: > matrix (1:9, nrow=3)

o/p:

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

> m(2,2) ← 9 # to modify single element

o/p:

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 9 & 8 \\ 3 & 6 & 5 \end{bmatrix}$$

> m(x<5) ← 0 → o/p;

	[,1]	[,2]	[,3]
[1,]	0	0	7
[2,]	0	5	8
[3,]	0	6	9

Transpose matrix: Transpose matrix is interchanging rows to colomns
- A common operator with matrix is to transpose it this can be done with the function "t()"

> t(m) o/p

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

> T(m) o/p

$$\begin{bmatrix} 0 & 0 & 7 \\ 0 & 5 & 8 \\ 0 & 6 & 9 \end{bmatrix}$$

General matrix Operation :- 11/04/2022

Common operations performed with matrices, there includes

1. linear algebra operations
2. matrix indexing
3. Matrix filtering

linear algebra: To perform linear algebra operations on matrix.

we can perform various linear algebra operations on matrices such as matrix multiplication, matrix scalar multiplication & matrix addition.

Ex: y ← matrix (c(1,2,3,4), nrow = 2, ncol = 2)

> y%*%y o/p: y = $\begin{bmatrix}1&3\\2&4\end{bmatrix}$ %*% $\begin{bmatrix}1&3\\2&4\end{bmatrix}$ = $\begin{bmatrix}10&14\\14&20\end{bmatrix}$

> 3*y o/p: $\begin{bmatrix}3&9\\6&12\end{bmatrix}$

> y+y o/p: $\begin{bmatrix}2&6\\4&8\end{bmatrix}$

$\begin{bmatrix}1+9&3+12\\2+12&4+16\end{bmatrix}$

2. Matrix Indexing:

The same operations we discuss for vectors apply to matrix as well

z ← matrix (c(1,2,3,4,1,1,0,0,0,1,0), nrow=4, ncol=3)

Ex: > z

	[,1]	[,2]	[,3]
[1,]	1	1	1
[2,]	2	1	0
[3,]	3	0	1
[4,]	4	0	0

> z [2:3] o/p: $\begin{bmatrix}1&1\\1&0\\0&1\\0&0\end{bmatrix}$

> z [2:3,] o/p: $\begin{bmatrix}2&1&0\\3&0&1\end{bmatrix}$

> z [2:3, 2] o/p: [1] 1 0

we requested the submatrix of two consisting of all elements with column number 2 & 3 and any row number these extracting the second & third column from the example 1 here extracting rows instead of column from example 2 you can also assign values to submatrix is

> y

1	4
2	5
3	6

> y [c(1,3),] ← matrix (c(1,1,8,12), nrow=2)

o/p: - $\begin{bmatrix}1&8\\2&5\\1&12\end{bmatrix}$

Here we assigned new values to the 1st & 3rd rows of y.

ex: > a ← matrix (nrow=3, ncol=3) > a : o/p : $\begin{bmatrix} NA & NA & NA \\ NA & NA & NA \\ NA & NA & NA \end{bmatrix}$

> y ← matrix (c(4,5,2,3), nrow=2)

> y o/p: $\begin{bmatrix} 4 & 2 \\ 5 & 3 \end{bmatrix}$

> a [2:3, 2:3] ← y

> a o/p :- $\begin{bmatrix} NA & NA & NA \\ NA & NA & NA \\ NA & NA & NA \end{bmatrix}$

ex:- > y

1 4
2 5
3 6

o/p : $\begin{bmatrix} 1 & 4 \\ 3 & 6 \end{bmatrix}$

> y [-2,]

8. Matrix Filtering:

filtering can be done with matrices just as vectors.

> a

1 4
2 5
3 6

> a [a [,2] >= 3,] o/p: $\begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$

A will print normally (vector format) 3 4 5 6

> z ← c(5, 12, 13)

> x [z %% 2 == 0] o/p: NA NA NA NA

> m

1 4
2 5
3 6

> m [m [,1] > 1 & m [,2] > 5,] o/p 3 6

> which(m>2) o/p: 3 4 5 6

12/04/2022

Applying functions to matrix rows & columns

One of the most famous used feature of R is the apply function in these family of apply functions are apply(), sapply(), tapply()

1. apply()
2. sapply()
3. tapply()
4. lapply()

1. apply:

Syntax: (m, margin, fargs)

where the arguments are as follows m, m is the matrix dimcode or margin is the dimension, equal to 0 if the function applies to rows (or) 2 for cololumns f is the function to be applied fargs is the optional set of arguments to be applied to f

Ex:- > m <- matrix (c(1:10), nrow = 5, ncol = 6)

> m o/p:

1	6	1	6	1	6
2	7	2	7	2	7
3	8	3	8	3	8
4	9	4	9	4	9
5	10	5	10	5	10

> apply (m, 2, sum)

> 15 40 15 40 15 40

> z

1 4

2 5

3 6

> apply (z, 2, mean) o/p: 2 5

> f <- function (x) x/c (2, 8)

> y <- apply (z, 1, f) o/p:

0.5	1.0	1.5
0.5	0.625	0.75

> y

we may have been suprised that the size of result is 2x3 rather than 3x2

- That first computations 0.5, 0.5 ends up at the 1st column in the output of apply() function not in the first row.

But this is the behaviour of the apply()

If the function to be applied returns a vector of k components then the result of apply() function will have k rows

- we can use the matrix transpose () T to change it

If necessary > t(apply (z, 1, f))

o/p:

0.5	0.5
1.0	0.625
1.5	0.75

```
> x
     [,1] [,2] [,3] [,4]
[1,]    1    0    1    0
[2,]    1    1    1    0
[3,]    1    0    1    1
[4,]    0    1    0    0
```

> apply (x, 2, copy maj) o/p: 1 1 1 0

> apply (x, 2, copy maj, 3) o/p: 1

Suppose, we have a matrix of 1's & 0's and want to create a vector for each row of the matrix the corresponding element of the vector will be either 0 or 1 depending on weather the majority of the first key elements in that row is 0 or 1. d will be the parameters that we may wish to vary

> apply (x, 1, copy maj, 3) o/p:- 1

> apply (x, 1, copy maj, 2) o/p:- 1

The values 3 & 2 form the actual arguments for the formal argument in copy maj function

The row consider of 1 0 1 1 0 the 1st d element of which were 1 0 1 a majority of these 3 elements were 1's So copymaj functions returns 1 and thus the first element of the output of apply() functions.

2. lapply():

It is useful for performing operations on list objects and returns a list of object of same length of original set

Syntax :- > lapply (x, fun)

x is a vector or object, fun is a function applied to each element of x

- In lapply() l- stands for lists -lapply() does not need margin

Ex:- > movies <- c ("x", "y", "z")

> movie - lower <- lapply (movies, tolower)

> str (movies, lower)

* Use unlist() function to convert the list into a vector

ex: > movies ← c("x", "y", "z")

> movies_lower ← unlist(lapply(movies, tolower))

> str(movies, lower)

3. sapply():

sapply() function takes list, vector or data-frame as input and gives output in vector or matrix

Syntax :- > sapply(x, fun)

x is a vector or object, fun is a function applied to each element of x

ex:- > data ← data.frame(names = c("x", "y", "z"),

age = c(20, 23, 24), gender = factor("F", "M", "M")

> data

names	age	gender
x	20	F
y	23	M
z	24	M

> sapply(data$age, data$gender, min)

o/p:

age	gender
20	F
23	M

4. tapply():

tapply computes a measure mean, min, max etc. or a function for each factor variable in a vector

Syntax : > tapply(x, index, fun=NULL)

x is an object usually a vector

index - a list containg a factor

fun is a fun applied to each element of x

> tapply(data$name, data$age, max)

o/p: name age

z 24

Adding and deleting Matrix rows & columns.

Matrices are of fixed length and dimensions so we can't add or remove rows or columns However matrices can be reassigned and thus we can achieve the same effect as if we had directly done addition or deletion

ex:-2 > x

12 5 13 16 8

> x ← c (x, 20) o/p: 12 5 13 16 8 20

> x

12 5 13 16 8 20

> x ← c (x[1:3], 20, x[4:6])

> x o/p: 12 5 13 20 16 8 20

> x ← x[-2:-4]

> x o/p: 12 16 8 20

> one

1 1 1 1

> z

1	1	
2	1	0
3	0	1
4	0	0

> cbind (one, z) o/p:

1	1	1	1
1	2	1	0
1	3	0	1
1	4	0	0

> cbind (1,2)

> q ← cbind (c(1,2), c(3,4))

> q o/p:

1	3
2	4

> m ← matrix (1:6, nrow = 3)

> m o/p:

1	4
2	5
3	6

> m ← m (c(1,3),)

> m o/p:

1	4
3	6

> x ← 1:5

> y ← 6:10

> z ← 11:15

> rbind (x,y,z)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

> cbind (x,y,z)

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

More on the vector or matrix distinction:-

A matrix is just a vector but with two dimensional attributes no of rows & no. of columns

Ex: z ← matrix (1:8, nrow = 4)

> z o/p: 1 5
2 6
3 7
4 8

As z is still a vector we can query its length.

> length (z) o/p: 8

But as a matrix z is bit more than vector

> class (z) o/p: "matrix"

> attributes (z) o/p: 1 2 3 4 5 6 7 8

> $dim (z) o/p: 4 2

The matrix class has one attribute, named dim which is a vector containing the no. of rows & columns in the matrix.

The no. of rows & columns obtained individually via the

> nrows (z) o/p: 4

> ncol (z) o/p: 2

Recall once again that objects can be printed in interactive mode by simply typing their names

> nrows
function (z)
dim (z) [1]

o/p: 1 5
2 6
3 7
4 8

These functions are useful when we are writing for general purpose whose argument is a vector

Avoiding unintended dimension reduction:-

In the world of statistics, dimension reduction is good thing with many statistical procedures aimed to do it well.

If we are working with 10 variables and can reduce that number to 3 that still captures the essence of data.

In R something else might merit the name dimension reduction that we have some time which to avoid.

Ex: z

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

```
> q<-z[2,]
> q.        o/p: 2 8
> attributes(z)
  $dim(z) o/p: 6 2
> attributes(q)
o/p: Null  $dim(q).
> str(z)
  int [1:4; 1:2] o/p: 1,2,3,4- - - 12
> str[q]
  int [1:2] 2,8 -> o/p.
```

q is a vector of length 12 rather than A 1½ matrix. there R informs that two has row and column no.s which are q doesn't similar.

STR function tells us that z has indexes ranging in 1:6 & 1:2 for rows and columns. while q's indicates. which simply randing 1 column i.e. vector in that q is vector not a matrix

R has a way to reduce the dimension using Drop argument

```
Eg: > q<-z[2,, drop=FALSE)
    > q.
        [,1] [,2]
o/p: [1,]  2    8
    > dim(q)
    o/p  1   2
```

Now q is a 1½ matrix not a two element vector for these res reason we may find it useful routine include the drop = FALSE argument in our matrix code. Because the square brace "["

"is actually a function:

eg:- > z=[3, 2] o/p: 9

> "[" (z, 3, 2] o/p: 9

> 4

[1]

If we have a vector that we wish to be treated as a matrix, we can use the as.matrix() function.

> 4

[1] 1 2 3

> v ← as.matrix (v)

> attributes (v)

$dim (v) o/p: NULL

> attributes (v)

$dim (v) o/p: -1 3

> attribute

Naming matrix rows and columns:

we can also give names to the entities (rows/columns)

> z

[,1] [,2]

[1,] 1 3

[2,] 2 4

> colnames (z) o/p: NULL

> colnames (z) ← c ("a", "b")

> z

a b

[1,] 1 3

[2,] 2 4

> z [, "a"]

o/p: [1] 1 2

21/04/2022

The function row names() the function works similarly naming rows and columns is usually is less important when writing R code for general applications but it can be useful when analysing a specific data set

Higher dimensional arrays:

The matrix is then a two dimensional data structure But suppose we have data taken at different times, one data point Per person per variable per time. time then become the 3rd dimension in addition to rows and columns.

In R Such data sets are called arrays.

eg: First test | Second test

46	30	46	43
21	25	41	35
50	48	50	56

> tests ← array (data = c (Firsttest, Secondtest), dim = c(3,2,2))

o/p: 56

> attributes (tests)

$dim · o/p: 3 2 2

> test [3,2,1] · o/p: 4 8

> tests ·

$$\begin{bmatrix} 46 & 30 \\ 21 & 25 \\ 50 & 48 \end{bmatrix} \quad \begin{bmatrix} 46 & 43 \\ 41 & 35 \\ 50 & 56 \end{bmatrix}$$

- One of the common use of arrays is calculating tables.

lists:- list is a vector in which all elements must be of the same mode R's list structure can be combined objects of different types.

Creating list: lists are refered to as recurssive vectors.

Consider an employee data base for each employee we wish to store the name salary and a boolean indicating union membership. we have three different modes here character, numeric & logical. Create a list to represent employee is x, y, z xyz

> j ← list (name = "xyz", salary = 55000, union = 5)

> j

$name

"xyz"

$salary

55000

$union

TRUE

The componet names called tages in the R segreture literature

eg:- > x ← list ("xyz", 55000, T)

```
> x
[[1]]
[1] "xyz"
[[2]]
[1] 55000
[[3]]
[1] TRUE
```

→ vector in a list

o/p: "xyz"

It is generally considered as clear and less error prone to use names instead of numeric indicates. (indexes)

Eg:- j ← list (name = "xyz", salary = 55000, union = T)

> j[1] o/p: "xyz"

> $salary - 55000

lists are vector they can be created via vector function.

Eg:- z ← vector (mode = "list")

> z ["abc"] ← c(3, 2, 1)

> z $abc

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

22/04/2022

general list operations

list Indexing: We can access an list component in several different ways

> x$salary
[1] 55000
> x[["salary"]]
[1] 55000
> j[[2]]
[1] 55000

- lst $c
- lst [["c"]]
- lst [i] where i is the index of c with in list

there lst is a variable

- lst ["c"]
- lst [i] where i is the index of c with in list

To refer we can refered to list components by their numerical components, treating the list as a vector.

There are three ways to access an individual component e of a list and return it in the data type of e

> x [1:2]
$name
[1] "xyz"
$salary
[1] 55000

> x2
$salary
55000
> class (x2)
[1] "list"

> str(x2)
list of 1
$ salary : num 55000
> x [[1:2]]
error in x [[1:2]]
Outsunpt Out of bound

> x2a
55000

> x2salary
55000

> x2 Salary [ ]
55000

Adding and deleting list elements:

The operations of adding and deleting list elements arising in a no.of context eg:

```
> z <- list (a = "abc", b = 12)
> z
 $a
 [1] "abc"
 $b
 [1] 12
> z$d <- "salary"
> z
 $a          $b        $d
 [1] "abc"   [1] 12    [1] salary
> z$b <- NULL
> z[[4]] <- 28
> z[5:7] <- c(FALSE, TRUE, TRUE)
> z
```

$a	$b	$d	$[[4]]	$[5]	$[6]	$[7]
[1] "abc"	[1] 12	Salary	28	FALSE	TRUE	TRUE

we can delete a list component by setting it to NULL

```
eg: > z$b <- NULL
    > z[4:7] <- NULL
    > z
```

In sequence [[ ]] doesn't accept

$a	$d
[1] "abc"	Salary

Note: Note that upon deleting z$b, the index of the elements after it moved up by 1

```
> z <- c(list("xyz", 55000, T), list(5))
 [[1]]
 [1] "xyz"
 [[2]]
 [1] 55000
 [[3]]
 [1] TRUE
 [[4]]
 [1] 5
```

length of the list:

Syntax: length(variable name)

```
eg:- length(z)
     o/p: 4
```

Accessing list components and values:

If the components in the list do have tages.

```
> names(a)
[1] "names" "salary" "union"
```

```
>ulx ← unlist(x)
>ulx
  name   salary   union
"xyz"   "55000"  "TRUE"
>class(ulx)  o/p: character.
```

The return value of unlist() is a vector

```
>z ← list(a=5, b=15, c=13)
>y ← unlist(z)
>class(y)
  "numeric"
>y
  a  b  c
  5  15 13
>w ← list(a=5, b="xyz")
>wv ← unlist(w)
>wv   o/p: a   b
           5  "xyz"
>class(wv)  o/p: character.
```

Here R choose the common denominator is character/string

```
>names(wv) ← NULL
>wv   o/p: "5"  "xyz"
>wvn ← unname(wv)
>wvn   o/p: "5"  "xyz"
```

23/04/2022

Applying functions to list:

Two functions are familiar for applying functions to list- lapply(), sapply()

using lapply(), sapply() functions.

```
>c ← lapply(list[1:3, 2:29], median)
>c
[[1]]
[1] 2
[[2]]
[1] 28
>c ← sapply(list[1:3, 27:29], median)
>c
o/p:- [1] 2 28
```

Recursive list:

list can be recursive that means you can have list with in a list

Eg:
```
>b <- list (u=5, v=12)
>c <- list (w=13)
>a <- list (b,c)
```

O/P:
```
> a
[[1]]
[1] $u
5
[2] $v
12

[[2]] $w
[1] 13
```

length of a is 2 — (b,c) => length(a) = 2:

- This code makes 'a' into a two dimensional component list with each component itself also be in a list
- The combine() 'c' has an optional argument recursive, which controls whether errors occurs when recursive lists are combine

Eg:
```
>c (list(a=1, b=2, c=list (d=5, e=9)))
```

O/P:
```
$a
[1] 1
$b
[1] 2

$c
$c.$d
[1] 5
$c $e
[1] 9
```

```
>c (list (a=1, b=2, c=list (d=5, e=9)), recursive=T)
```

```
$a
[1] 1
$b
[1] 2

$c
$c.$d
[1] 5
$c $e
[1] 9
```