Unit 3 & 4 Notes.

Sub: Introduction to Data Science.

STATISTICS WITH R PROGRAMMING

Understanding basic data types in R

- To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.
- **Very Important** to understand because these are the things you will manipulate on a day-to-day basis in R. Most common source of frustration among beginners.
- Everything in Ris an object.

Rhas 5 basic atomic classes

- logical (e.g., TRUE, FALSE)
- integer (e.g,, 2L, as.integer(3))
- numeric (real or decimal) (e.g, 2, 2.0, pi)
- complex (e.g, 1 + 0i, 1 + 4i)
- character (e.g, "a", "swc")

```
typeof() # what is it?
class() # what is it?
storage.mode() # what is it?
length() # how long is it? What about two dimensional objects?
attributes() # does it have any metadata?
```

R also has many data structures. These include

- vector
- list
- matrix
- data frame
- factors (we will avoid these, but they have their uses)
- tables

Vectors

A vector is the most common and basic data structure in Rand is pretty much the workhorse of R. Vectors can be of two types:

- atomic vectors
- lists

Atomic Vectors A vector can be a vector of characters, logical, integers or numeric. Create an empty

vector with vector()

```
x <- vector()
# with a pre-defined length
x <- vector(length = 10)
# with a length and type
vector("character", length = 10)
vector("numeric", length = 10)
vector("integer", length = 10)
vector("logical", length = 10)
```

The general pattern is `vector(class of object, length)`. You can also create vectors by concatenating them using the `c()` function.

Various examples:

```
x <- c(1, 2, 3)
```

x is a numeric vector. These are the most common kind. They are numeric objects and are treated as double precision real numbers. To explicitly create integers, add a Lat the end.

```
x1 <- c(1L, 2L, 3L)
```

You can also have logical vectors.

```
y <- c(TRUE, TRUE, FALSE, FALSE)
```

(Don't use Tand F!)

Finally you can have character vectors:

```
z <- c("Alec", "Dan", "Rob", "Rich")
```

Examine your vector

```
typeof(z)
length(z)
class(z)
str(z)
```

More examples of vectors

```
x <- c(0.5, 0.7)
x <- c(TRUE, FALSE)
x <- c("a", "b", "c", "d", "e")
x <- 9:100
x <- c(i+0i, 2+4i)
```

You can also create vectors as sequence of numbers

```
series <- 1:10
seq(10)
seq(1, 10, by = 0.1)
```

Other objects

`Inf` is infinity. You can have positive or negative infinity.

```
 1/0
# [1] Inf
 1/Inf
# [1] 0
```

`NaN` means Not a number. it's an undefined value.

```
0/0
NaN.
```

Each object has an attribute. Attributes can be part of an object of R. These include

- names
- dimnames
- length
- class
- attributes (contain metadata)

For a vector, `length(vector_name)` is just the total number of elements.

Vectors may only have one type

R will create a resulting vector that is the least common denominator. The coercion will move towards the one that's easiest to coerce to.

Guess what the following do without running them first

```
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

This is called implicit coercion.

The coersion rule goes `logical` -> `integer` -> `numeric` -> `complex` -> `character`. You

can also coerce vectors explicitly using the `as.<class_name>`.

Example

```
as.numeric()
as.character()
```

When you coerce an existing numeric vector with `as.numeric()`, it does nothing.

```
x <- 0:6
as.numeric(x)
as.logical(x)
as.character(x)
as.complex(x)
```

Sometimes coercions, especially nonsensical ones won't work.

```
x <- c("a", "b", "c")
as.numeric(x)
as.logical(x)
# both don't work
```

Sometimes there is implicit conversion

```
1 < "2"
# TRUE
"1" > 2
# FALSE
1 < "a"
# TRUE
```

Operation on Vectors

The above mentioned operators work on vectors. The variables used above were in fact single element vectors.

We can use the function `c()` (as in concatenate) to make vectors in R.

All operations are carried out in element-wise fashion. Here is an example.

```
> x <- c(2,8,3)
> y <- c(6,4,1)

> x+y
[1]  8 12  4

> x>y
[1] FALSE  TRUE  TRUE
```

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.

R will issue a ***warning*** if the length of the longer vector is not an integral multiple of the shorter vector.

```
> 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
```

How to create a Vector in R programming?

Vectors are generally created using the `c()` function.

Since, a vector must have elements of the same type, this function will try and coerce elements to the same type, if they are different.

Coercion is from lower to higher types from logical to integer to double to character.

```
> 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"
```

If we want to create a vector of consecutive numbers, the `:` operator is very helpful.

Example 1: Creating a vector using : operator

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

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

More complex sequences can be created using the `seq()` function, like defining number of points in an interval, or the step size.

Example 2: Creating a vector using seq() function

```
> seq(1, 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

> seq(from = 1, by = 2, length.out = 10)

 [1]  1  3  5  7  9 11 13 15 17 19

> seq(f = 1, b = 5, leng = 10)

 [1]  1  6 11 16 21 26 31 36 41 46

> seq(0, 1, length.out = 11)

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(1, 9, by = 2)           # matches 'end'

[1] 1 3 5 7 9

> seq(17) # same as 1:17, or even better seq_len(17)

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

```
> seq_len(17)

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17

> seq(1, 6, by = 3)

[1] 1 4

> seq(1.575, 5.125, by = 0.05)

 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025
2.075 2.125

[13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625
2.675 2.725

[25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225
3.275 3.325

[37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725 3.775 3.825
3.875 3.925

[49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.325 4.375 4.425
4.475 4.525

[61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.925 4.975 5.025
5.075 5.125
```

How to access 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.

Using integer vector as index

Vector index in R starts from 1, unlike most programming languages where index start from 0.
We can use a vector of integers as index to access specific elements.
We can also use negative integers to return all elements except that those specified.

But we cannot mix positive and negative integers while indexing and real numbers, if used, are truncated to integers.

```
> 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)] : only 0's may be mixed with negative subscripts

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

Using logical vector as index

When we use a logical vector for indexing, the position where the logical vector is `TRUE` is returned.

This useful feature helps us in filtering of vector as shown below.

```
> x[c(TRUE, FALSE, FALSE, TRUE)]
[1] -3  3

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

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

In the above example, the expression `x>0` will yield a logical vector `(FALSE, FALSE, FALSE, TRUE)` which is then used for indexing.

Using character vector as index

This type of indexing is useful when dealing with named vectors. We can name each elements of a vector.

```
> 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
```

How to modify a vector in R?

We can modify a vector using the assignment operator.

We can use the techniques discussed above to access specific elements and modify them.

If we want to truncate the elements, we can use reassignments.

```
> 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
```

How to delete a Vector?

We can delete a vector by simply assigning a NULL to it.

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

> x
NULL
> x[4]
NULL
```

R Program to Sort aVector

Sorting of vectors can be done using the **sort()** function.

By default, it sorts in ascending order. To sort in descending order we can pass decreasing=TRUE.

Note that sort is not in-place. This means that the original vector is not effected (sorted).

Only a sorted version of it is returned.

Example: Sort a Vector

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

> # sort in ascending order
> sort(x)
```

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

> # sort in descending order
> sort(x, decreasing=TRUE)
[1] 8 7 6 5 4 3 2 2 2 1

> # vector x remains unaffected
> x
[1] 7 1 8 3 2 6 5 2 2 4
```

Sometimes we would want the index of the sorted vector instead of the values. In such case we can use the function order().

```
> order(x)
[1]  2  5  8  9  4 10  7  6  1  3

> order(x, decreasing=TRUE)
[1]  3  1  6  7 10  4  5  8  9  2

> x[order(x)]      # this will also sort x
[1] 1 2 2 2 3 4 5 6 7 8
```

Matrix

Matrices are a special vector in R. They are not a separate class of object but simply a vector but now with dimensions added on to it. Matrices have rows and columns.

```
m <- matrix(nrow = 2, ncol = 2)
m
dim(m)
same as
attributes(m)
```

Matrices are constructed columnwise.

```
m <- matrix(1:6, nrow=2, ncol =3)
```

Other ways to construct a matrix

```
m <- 1:10
dim(m) <- c(2,5)
```

This takes a vector and transform into a matrix with 2 rows and 5 columns. Another way is

to bind columns or rows using cbind()and rbind().

```
x <- 1:3
y <- 10:12
cbind(x,y)
# or
rbind(x,y)
```

R Matrix

In this to work with matrix in R. You will learn to create and modify matrix, and access matrix elements.

Matrix is a two dimensional data structure in R programming.

Matrix is similar to vectors but additionally contains the dimension attribute. All attributes of an object can be checked with the attributes() function (dimension can be checked directly with the dim() function).

We can check if a variable is a matrix or not with the class() function.

```
> 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
```

Let us consider a matrix as above.

How to create a matrix in R programming?

Matrix can be created using the matrix() function.

Dimension of the matrix can be defined by passing appropriate value for arguments nrowand ncol.

Providing value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from length of the data.

```
> 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
```

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing TRUE to the argument byrow.

```
> 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
```

In all cases, however, a matrix is stored in column-major order internally as we will see in the subsequent sections.

It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument dimnames.

```
> 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
```

These names can be accessed or changed with two helpful functions colnames() and rownames().

```
> 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
```

Another way of creating a matrix is by using functions cbind() and rbind() as in column bind and row bind.

```
> cbind(c(1,2,3),c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

> rbind(c(1,2,3),c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Finally, you can also create a matrix from a vector by setting its dimension using dim().

```
> 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"
```

How to access Elements of a matrix?

We can access elements of a matrix using the square bracket [indexing method. Elements can be accessed as var[row, column]. Here rows and columns are vectors.

Using integer vector as index

We specify the row numbers and column numbers as vectors and use it for indexing. If any field

inside the bracket is left blank, it selects all.

We can use negative integers to specify rows or columns to be excluded.

```
> 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


> 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
```

One thing to notice here is that, if the matrix returned after indexing is a row matrix or column matrix, the result is given as a vector.

```
> x[1,]
[1] 1 4 7
> class(x[1,])
[1] "integer"
```

This behavior can be avoided by using the argument drop = FALSE while indexing.

```
> 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"
```

It is possible to index a matrix with a single vector.

While indexing in such a way, it acts like a vector formed by stacking columns of the matrix one after another. The result is returned as a vector.

```
> 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

Two logical vectors can be used to index a matrix. In such situation, rows and columns where the value is TRUE is returned. These indexing vectors are recycled if necessary and can be mixed with integer vectors.

```
> 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
```

It is also possible to index using a single logical vector where recycling takes place if necessary.

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

In the above example, the matrix x is treated as vector formed by stacking columns of the matrix one after another, i.e., (4,6,1,8,0,2,3,7,9).

The indexing logical vector is also recycled and thus alternating elements are selected. This property is utilized for filtering of matrix elements as shown below.

```
> 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

Indexing with character vector is possible for matrix with named row or column. This can be mixed with integer or logical indexing.

```
> 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
```

How to modify a matrix in R?

We can combine assignment operator with the above learned methods for accessing elements of a matrix to modify it.

```
> 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
```

A common operation with matrix is to transpose it. This can be done with the function `t()`.

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

We can add row or column using `rbind()` and `cbind()` function respectively. Similarly, it can be removed through reassignment.

```
> 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
```

Dimension of matrix can be modified as well, using the dim() function.

```
> 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
```

R Data Frame

Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length.

Each component form the column and contents of the component form therows.

Check if a variable is a data frame using class()

We can check if a variable is a data frame or not using the class() function.

```
> x
  SN Age Name
1  1  21 John
2  2  15 Dora

> typeof(x)      # data frame is a special case of list
[1] "list"

> class(x)
[1] "data.frame"
```

In this example, x can be considered as a list of 3 components with each component having a two element vector. Some useful functions to know more about a data frame are given below.

Functions of data frame

```
> names(x)
[1] "SN"    "Age"   "Name"

> ncol(x)
[1] 3

> nrow(x)
[1] 2
```

```
> length(x)      # returns length of the list, same as ncol()
[1] 3
```

How to create a Data Frame in R?

We can create a data frame using the data.frame() function.

For example, the above shown data frame can be created as follows.

```
> x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" =
c("John","Dora"))


> str(x)      # structure of x
'data.frame':    2 obs. of  3 variables:
$ SN   : int  1 2
$ Age  : num  21 15
$ Name: Factor w/ 2 levels "Dora","John": 2 1
```

Notice above that the third column, Name is of type factor, instead of a character vector.

By default, data.frame() function converts character vector into factor.

To suppress this behavior, we can pass the argument stringsAsFactors=FALSE.

```
> x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" = c("John",
"Dora"), stringsAsFactors = FALSE)


> str(x)      # now the third column is a character vector
'data.frame':    2 obs. of  3 variables:
$ SN   : int  1 2
$ Age  : num  21 15
$ Name: chr  "John" "Dora"
```

Many data input functions of R like, read.table(), read.csv(), read.delim(), read.fwf() also read data into a data frame.

How to access Components of a Data Frame?

Components of data frame can be accessed like a list or like a matrix.

Accessing like a list

We can use either `[`, `[[` or `$` operator to access columns of data frame.

```
>x["Name"]
  Name
1 John
2 Dora

> x$Name
[1] "John" "Dora"

> x[["Name"]]
[1] "John" "Dora"

> x[[3]]
[1] "John" "Dora"
```

Accessing with `[[` or `$` is similar. However, it differs for `[` in that, indexing with `[` will return us a data frame but the other two will reduce it into a vector.

Accessing like a matrix

Data frames can be accessed like a matrix by providing index for row and column.

To illustrate this, we use datasets already available in R. Datasets that are available can be listed with the command `library(help = "datasets")`.

We will use the `trees` dataset which contains `Girth`, `Height` and `Volume` for Black Cherry Trees.

A data frame can be examined using functions like `str()` and `head()`.

```
> str(trees)
```

```
'data.frame':    31 obs. of  3 variables:
$ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
$ Height: num  70 65 63 72 81 83 66 75 80 75 ...
$ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

```
> head(trees,n=3)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
```

We can see that trees is a data frame with 31 rows and 3 columns. We also display the first 3 rows of the data frame.

Now we proceed to access the data frame like a matrix.

```
> trees[2:3,]     # select 2nd and 3rd row
  Girth Height Volume
2   8.6     65   10.3
3   8.8     63   10.2


> trees[trees$Height > 82,]     # selects rows with Height greater than 82
   Girth Height Volume
6   10.8     83   19.7
17  12.9     85   33.8
18  13.3     86   27.4
31  20.6     87   77.0


> trees[10:12,2]
[1] 75 79 76
```

We can see in the last case that the returned type is a vector since we extracted data from a single column.

This behavior can be avoided by passing the argument drop=FALSE as follows

```
> trees[10:12,2, drop = FALSE]
   Height
10      75
11      79
12      76
```

How to modify a Data Frame in R?

Data frames can be modified like we modified matrices through reassignment.

```
> x
 SN Age Name
1  1  21 John
2  2  15 Dora

> x[1,"Age"] <- 20; x
 SN Age Name
1  1  20 John
2  2  15 Dora
```

Adding Components

Rows can be added to a data frame using the `rbind()` function.

```
>rbind(x,list(1,16,"Paul"))
 SN Age  Name
1  1  20 John
2  2  15 Dora
3  1  16 Paul
```

Similarly, we can add columns using `cbind()`.

```
>cbind(x,State=c("NY","FL"))
  SN Age Name State
1  1  20 John    NY
2  2  15 Dora    FL
```

Since data frames are implemented as list, we can also add new columns through simple list-like assignments.

```
> x
  SN Age Name
1  1  20 John
2  2  15 Dora

> x$State <- c("NY","FL"); x
  SN Age Name State
1  1  20 John    NY
2  2  15 Dora    FL
```

Deleting Component

Data frame columns can be deleted by assigning `NULL` to it.

```
> x$State <- NULL
> x
  SN Age Name
1  1  20 John
2  2  15 Dora
```

Similarly, rows can be deleted through reassignments.

```
> x <- x[-1,]
> x
  SN Age Name
2  2  15 Dora
```

R Factors

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data).

For example, a data field such as marital status may contain only values from single, married, separated, divorced, or widowed.

In such case, we know the possible values beforehand and these predefined, distinct values are called levels. Following is an example of factor in R.

```
> x
[1] single  married married single
Levels: married single
```

Here, we can see that factor x has four elements and two levels. We can check if a variable is a factor or not using class() function.

Similarly, levels of a factor can be checked using the levels() function.

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

> levels(x)
[1] "married" "single"
```

How to create a factor in R?

We can create a factor using the function **factor()**. Levels of a factor are inferred from the data if not provided.

```
> x <- factor(c("single", "married", "married", "single"));
> x
[1] single  married married single
Levels: married single

> x <- factor(c("single", "married", "married", "single"), levels =
c("single", "married", "divorced"));
> x
[1] single  married married single
Levels: single married divorced
```

We can see from the above example that levels may be predefined even if not used.

Factors are closely related with vectors. In fact, factors are stored as integer vectors. This is clearly seen from its structure.

```
> x <- factor(c("single","married","married","single"))
> str(x)
Factor w/ 2 levels "married","single": 2 1 1 2
```

We see that levels are stored in a character vector and the individual elements are actually stored as indices.

Factors are also created when we read non-numerical columns into a data frame.

By default, `data.frame()` function converts character vector into factor. To suppress this behavior, we have to pass the argument `stringsAsFactors = FALSE`.

How to access compoments of a factor?

Accessing components of a factor is very much similar to that of vectors.

```
> x
[1] single  married married single
Levels: married single

> x[3]              # access 3rd element
[1] married
Levels: married single

>  x[c(2, 4)]       # access 2nd and 4th element
[1] married single
Levels: married single

> x[-1]             # access all but 1st element
[1] married married single
Levels: married single

> x[c(TRUE, FALSE, FALSE, TRUE)]  # using logical vector
[1] single single
Levels: married single
```

How to modify a factor?

Components of a factor can be modified using simple assignments. However, we cannot choose values outside of its predefined levels.

```
> x
[1] single   married  married  single
Levels: single married  divorced

> x[2] <- "divorced"      # modify second element;   x
[1] single    divorced  married   single
Levels: single married  divorced

> x[3] <- "widowed"      # cannot assign values outside levels
Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "widowed") :
 invalid factor level, NA generated

> x
[1] single    divorced <NA>       single
Levels: single married divorced
```

A workaround to this is to add the value to the level first.

```
> levels(x) <- c(levels(x), "widowed")      # add new level

> x[3] <- "widowed"

> x
[1] single    divorced widowed   single
Levels: single married divorced widowed
```

LISTS

INTRODUCTION

A list in R constitutes of different objects such as strings, numbers, and vectors. Further, it can include another list within it. Lists in R can also include matrices and functions making it functionally richer than other existing list utilities. Lists can be obtained as return values from many modelling functions such as t.test() for t tests , lm() for liner models, etc. Customized lists can be created to store useful information such as grocery list, shopping list, list of books, and so on. Most of these lists are simple with nominal and ordinal values, but in many cases there is a need for storing complex information such as the configuration details of a phone; in such cases, the list function of R can be used.

CREATING A LIST

A list can be created using the *list*() function, which takes in different R objects and stores the values in the database.

Figure 3.1 demonstrates the creation of an anonymous list *list_data*, using the *list*() function, containing strings, numbers, vectors, and logical values as data objects. After creating the list it can be viewed with the help of the *print* statement.

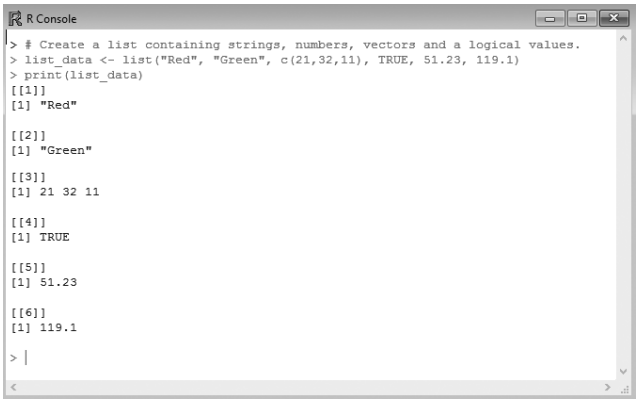


Fig. 3.1 Creation of lists

Creating a Named List

Lists in R have a special property wherein names can be assigned to the previously defined set of elements in the list. These forms of named lists provide correct information about a particular category of data with many different attributes. Named lists can be created using the *names*() function.

Example 3.1 Write the command in R console to create a list containing a vector, a matrix, and a list. Also give names to the elements in the list and display the list.

Solution:

Figure 3.2 creates a named list using the *names*() function. First a list called *list_data* containing a vector and a matrix and another list called *list* are created. Later using the *names*() function, a vector with the names to these lists is created. The created list is viewed using the *print* statement.

```
R Console
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
>
> # Show the list.
> print(list_data)
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3
```

The elements present in a normal list can be accessed using the index value of that element. For a named list, the elements are accessed based on their names.

In Fig. 3.3, a list called *list_data* is created using the *list*() function and later the list elements are given names using the *names*() function. So to access the first element from list_data, the index value of the list is accessed; here the list elements are indexed starting from 1 and not from 0. Therefore printing *list_data*[1] gives the names of the first and later elements in the list. If the third element in the list is considered, which is another list containing two different elements, printing *list_data*[3] gives the list name and the two elements. The elements in the list can also be accessed with their names using the '$' sign appended with the name of

```
R Console
> # Create a list containing a vector, a matrix and a list.
> list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
+    list("green",12.3))
>
> # Give names to the elements in the list.
> names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")
>
> # Access the first element of the list.
> print(list_data[1])
$`1st Quarter`
[1] "Jan" "Feb" "Mar"

>
> # Access the thrid element. As it is also a list, all its elements will be printed.
> print(list_data[3])
$`A Inner list`
$`A Inner list`[[1]]
[1] "green"

$`A Inner list`[[2]]
[1] 12.3


>
> # Access the list element using the name of the element.
> print(list_data$A_Matrix)
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8
>
```

Fig. 3.3 Accessing elements from a list

the element in the list. Hence printing *list_data$A_Matrix* gives the output of the third element of the list which is a matrix.

MANIPULATING LIST ELEMENTS

Addition, deletion, and updating operations can be performed on the list to manipulate the elements. Elements can be added and deleted only at the end of the list, whereas, the update operation can be performed on any element in the list irrespective of its position.

Example 3.2 Write the command in R console to add a new element at the end of the list and display the same.

Solution:

Figure 3.4 depicts the addition of an element to a list. Consider the example list created in Fig. 3.2, which consists of only three elements. To add the fourth element to this list, append the element (which is a string here) "New element" to the 4th indexed position of the list, that is, *list_data*[4]. To view the added element in the list, print the list and the four elements are displayed.

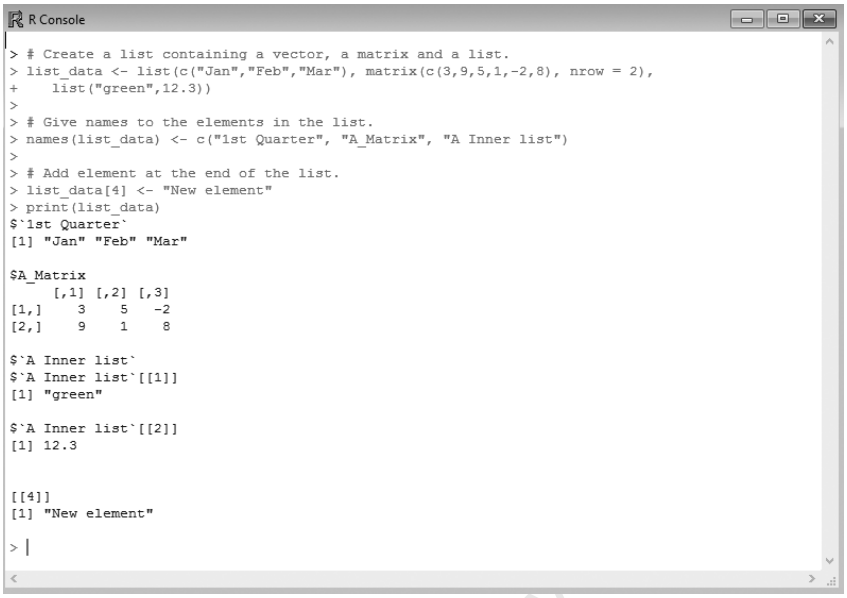


Fig. 3.4 Adding an element to the list

Example 3.3 Write the command in R console to delete the fourth element from a list and display theresultant list.

Solution:

Here, in Fig. 3.5, the fourth element, "New element" can be deleted by setting its value to NULL. Printing *list_data* shows only three elements in the list and when the fourth element is tried to be printed, it shows NULL.

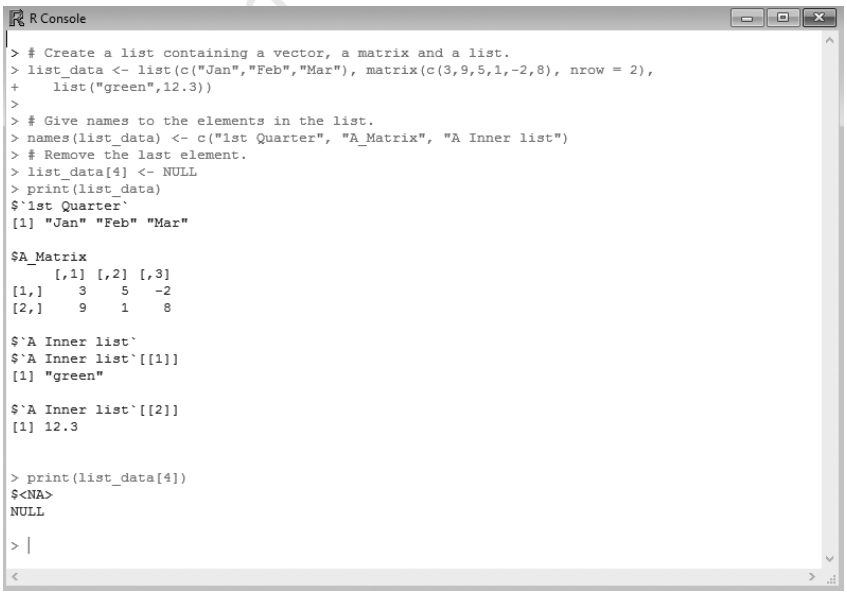


Fig. 3.5 Deleting an element from a list

Example 3.4 Write the command in R console to update the third element of the list and display theresultant list.

Solution:

In Fig. 3.6, we update the third element of *list_data* as "updated element". This is then checked by printingthe list.

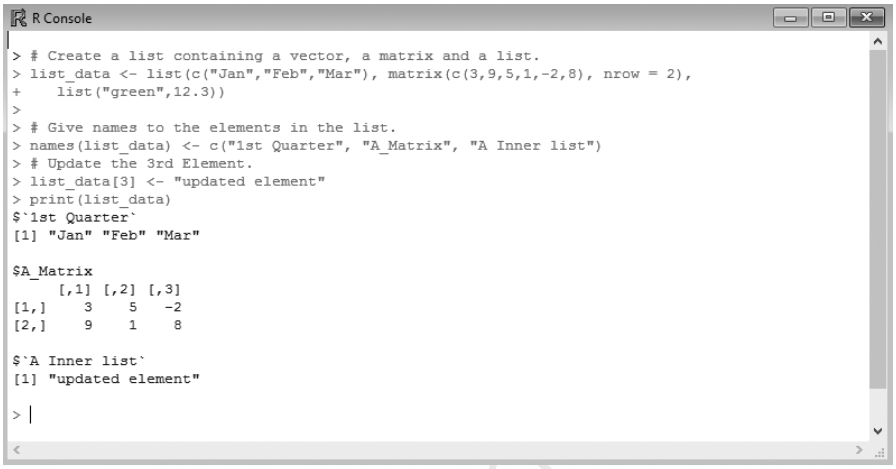


Fig. 3.6 Updating a list element

MERGING LISTS

Multiple lists can be merged by placing all those lists in a single *list*() function.

Figure 3.7 shows two lists *list1* and *list2* created using the *list*() function. Now placing them in a single *list*() function gives a merged list called *merged.list*, which contains the elements of both the lists placed sequentially.

Fig. 3.7 Merging two lists

CONVERTING LISTS TO VECTORS

There are some special cases where lists are converted to vectors for further manipulation of the data elements. This is done because in addition to the basic operations of add, delete, and update, there are other arithmetic operations that can be easily applied when the operands are in vector form. For this conversion, a special function called *unlist*() is used; this function takes the input as list and produces vectors as depicted in detailin Example 3.5.

Example 3.5 Write the command in R console to create two lists, each containing 5 elements. Convert thelists into vectors and perform addition on the two vectors. Display the resultant vector.

Solution:

As shown in Fig. 3.8, two lists namely, *list1* and *list2* are initially created. Later with the help of the *unlist*() function, both the lists are converted to vectors named v1 and v2. Addition, subtraction, and multiplication oftwo vectors can be performed, which was not possible if the data was in the form of a list.

Fig. 3.8 Converting lists to vectors

R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division

An example run

```
> x <- 5
> y <- 16

> x+y
[1] 21

> x-y
[1] -11

> x*y
[1] 80

> y/x
[1] 3.2
```

```
> y%/%x
[1] 3

> y%%x
[1] 1

> y^x
[1] 1048576
```

R Relational Operators

Relational operators are used to compare between values. Here is a list of relational operators available in R.

Relational Operators in R

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

An example run

```
> x <- 5
> y <- 16
```

```
> x<y
[1] TRUE

> x>y
[1] FALSE

> x<=5
[1] TRUE

> y>=20
[1] FALSE

> y == 16
[1] TRUE

> x != 5
[1] FALSE
```

Operation on Vectors

The above mentioned operators work on vectors. The variables used above were in fact single element vectors.

We can use the function `c()` (as in concatenate) to make vectors in R.

All operations are carried out in element-wise fashion. Here is an example.

```
> x <- c(2,8,3)
> y <- c(6,4,1)

> x+y
[1]  8 12  4

> x>y
[1] FALSE  TRUE  TRUE
```

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.

R will issue a ***warning*** if the length of the longer vector is not an integral multiple of the shorter vector.

```
> 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
```

R Logical Operators

Logical operators are used to carry out Boolean operations like AND, OR etc.

Logical Operators in R	
Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Operators & and | perform element-wise operation producing result having length of the longer operand.

But && and || examines only the first element of the operands resulting into a single length logical vector.

Zero is considered FALSE and non-zero numbers are taken as TRUE. An example run.

```
> x <- c(TRUE,FALSE,0,6)
> y <- c(FALSE,TRUE,FALSE,TRUE)

> !x
[1] FALSE  TRUE  TRUE FALSE

> x&y
[1] FALSE FALSE FALSE  TRUE

> x&&y
[1] FALSE

> x|y
[1]  TRUE  TRUE FALSE  TRUE

> x||y
[1] TRUE
```

R Assignment Operators

These operators are used to assign values to variables.

Assignment Operators in R

Operator	Description
<-, <<-, =	Leftwards assignment
->, ->>	Rightwards assignment

The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment.

The <<- operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

```
> x <- 5
> x
[1] 5

> x = 9
> x
[1] 9

> 10 -> x
> x
[1] 10
```

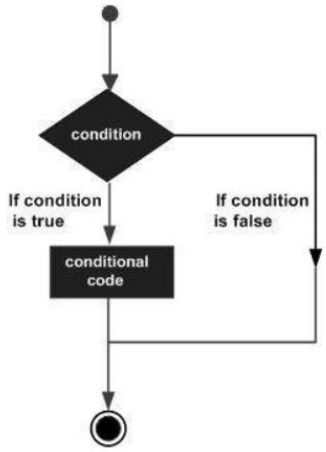
Example: Hello World Program

```
> # We can use the print() function
> print("Hello World!")
[1] "Hello World!"

> # Quotes can be suppressed in the output
> print("Hello World!", quote = FALSE)
[1] Hello World!

> # If there are more than 1 item, we can concatenate using paste()
> print(paste("How","are","you?"))
[1] "How are you?"
```

R - Decision making

Decision making structures require the programmer to specify one or more conditions to be evaluatedDecision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition isor tested by the program, along with a statement or statements to be executed if the condition is determined to bedetermined to be truetrue, and optionally, other statements to be executed if the condition is determined to, and optionally, other statements to be executed if the condition is determined to bebe falsefalse..

Following is the general form of a typical decision making structure found in most of the programmingFollowing is the general form of a typical decision making structure found in most of the programming languages −languages –



R provides the following types of decision making statements. Click the following links to check theirR provides the following types of decision making statements. Click the following links to check their detail.detail.

Sr.No.	Statement & Description
1	if statement An **if** statement consists of a Boolean expression followed by one or more statements.
2	if...else statement An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.
3	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.

Loops in R

Mastering programming languages require a programmer to master the essential challenge of learning loops. Learning an advanced concept in any language is not possible without learning loops. And things are no different,even in the relatively simple R language. The R language has those traditional looping structures like the for loop and while loop, but that’s notall.

Additionally, R has an unconventional Repeat Loop and a family of loops related to the Apply loop, which contains apply, sapply, lapply, and tapply.Use of traditional loops like for and while is for applying conventional repetitive transactions. However, the apply loop family has been designed to handle a few specific cases, i.e. tapply loop has been designed to handle the cases of ragged arrays. As in the other languages, it is quite possible fora user to use different loops inside each other in R language. I.e. a while loop can be used inside a for loop.

For Loop in R

If you are a programmer who develops software applications using differentlanguages, you will have an idea about how a for loop works. For loop implementation of R language is not any different from the other language for loops.

Just like other programming languages, for loop in R language has been used to execute repetitive code statements for a particular number of times.Generally, a for loop construction looks something like:

```
for (vector counter)

{

Statements

}
```

To explain the concept of the for loop in detail, let us take an example of creating a for loop to square all the elements of a dataset sqr, which has all odd numbers in the range of 1 to 100. A notable thing of this example is thefaster vectorizing.

```
sqr = seq(1, 100 by=2)

sqr.squared = NULL

for (n in 1:50)

{

sqr.squared[n] = sqr[n]^2

}
```

At any point of time, if your goal is to create a new vector, the first processing step shall be of setting up a vector to store member variables before creating a loop. sqr.squared = NULL statement basically depicts aset up of a vector.

Hence, we set up a vector to add the date in it at a later stage. Even thoughthat's neither an efficient nor speedy way, it is a full-proof way for sure.

Once the vector is set up, the main task for setting up a for loop starts. A forloop in example is programmed to run 50 iterations, set up by variable 'n' inthe example (You can program this iteration variable to anything you want). Square of nth (where n = 1, n =2 … n = 50) number will be stored at the nth (where n = 1, n =2 … n = 50) location in the new vector sqr.squared.Hence, for loop will run 50 iterations to store the squares of odd numbers in the range of 1 to 100 at 1 to 50th location in vector sqr.squared.

However, it may happen that you cannot handle the straight difference between the numbersof loops run against nth number of element of vectoroperated upon. I.e., once 7 loop iterations have been completed, the next iteration number would be 8 (since n = 8). However, 8th element of sqr vector will be sqr[8], which is 15. Hence, sqr.squared[8] should have the value 225, which is square of 15.

Debugging for loops in R Language

During programming of for loop in R language, if you come across anyproblem in the loop, you can consider below given points:

1. You may have reset a vector inside your loop. I.e. It is quite possible that you might have used a statement like "sqr.vector = NULL" within a loop instead of resetting it before starting your loop. This is a possible debuggingpoint. Believe me, I spent more than an hour to figure out this mistake.

2. Did you miss out on subscribing a new vector? I.e. you may have put astatement like:

```
{sqr.squared = sqr[n]^2}
```

Didn't' understand your mistake? Let me explain it.

If you compare this statement with our example loop, you would understand that you have forgotten to put in your square brackets with a counter inside, at the left-hand operand in a statement. Hence, your new vector sqr.squared will always be filled with the last calculated value (sinceyour new vector can only store one value).

Stop Aspect of For Loop

Let us take yet another example of for loop in R to understand the stopaspect of the loop.

```
n <- 1:5

r <- NULL

for (i in seq (along=n)) {

   if (n[i] <3) {

   r <- c(r, n[i]-1)

   } else {

   stop ("values shall be <3")

   }

}
```

Here is the output of the example for loop given above:

In the above example, for loop has been processed to store the value in vector n for 4 times. However, the limit of n has been set to 2, which allowsstoring the value only twice. In such conditions, stop statements can be used to display error messages inside a for loop. The above example gives agood explanation of use of stop statement inside for loop.

```
Error: values shall be <3

z

[1] 0 1
```

Setting up Counter inside for loop:

Let's say, if you are creating a for loop inside a large program, number of iterations you want for loop to run may depend upon a vector length, whichmay have dependency on other factors as well. R language provides the facility to set up a loop counter inside the vector part of for loop, as shown below:

```
for (n in 0:length (sqr))

{

Statements to execute the loop sqr number of times

}
```

Well Constructed For Loop

As we discussed earlier, there are few performance related issues with for loops in R. If you are running in speed related troubles, you must follow theconcept of well-constructed loops to get rid of such issues.

1. Try and get rid of as much code as possible inside the loop and take thatout of the loop.

If you assess the vector operations possible prior to for loop, try to fetchthem outside for loop

Repeat Loop In R Language

Just like for and while loop, repeat loop is one of the frequently used loopsin the R language. As the name suggests, repeat loop is majorly used to execute the statement repetitively until a constraint condition is met like while loop. However, break statement is the only way to come out of the repeat loop (it is the only way to terminate the repeat loop).

A repeat loop looks something like this in general:

```
repeat

{

 //statements

 if (constraint condition)

 {

    break //breaks repeat loop

 }

}
```


Let us understand how repeat loop works:

1. A repeat statement at the start in above code is a repeat keyword thatmarks the start of a repeat loop.
2. //statements inside the curly brackets depicts the executionstatements written inside the repeat loop.
3. If (constraint condition) defines if loop with the constraint condition.

4. Break statement inside if condition is used to break the repeat loop.
5. With the satisfaction of constraint condition, code control goes insideif condition has the break statement. Execution of the break statement breaks the repeat loop to terminate the loop.

Let us take an example for further understanding of repeat loop:

```
{

sum <- sum + 2;

print(sum);

if (sum > 11)

break;

}
```

Output:

3

5

7

9

11

13

A program given in the example is designed to repetitively add 2 in the self-number until the resultant number reached 11.

Here, sum > 11 is a constraint condition which is used with if loop. If loopcontains a break statement to break the repeat loop once the constraint condition is satisfied.

Usage Of Repeat Loop

Repeat loop has almost the same functionality as that of for loop and whileloop. However, it is being used generally when constraint conditions are known to the programmer and the programmer does not want to go into the complexity of while and for loop.

In fact, repeat loop is used as a do-while loop at many places in R language.If you observe the general definition of repeat loop, it reflects the same meaning as of do-while loop, which says “execute statements till constraint condition is satisfied”. **Learning Data Analysis in R** language would make learning loops even easier for you.

Points to Ponder for Repeat Loop implementation:

1. Ensure that constraint condition variable used to break repeat loop isbeing processed inside the repeat loop.
2. Break statement implemented inside constraint conditioned if loop isthe only way out from the repeat loop.
3. Repeat loop can be used as do-while loop in the R languageprograms.

While Loop In R:

While loop is one of the most simple loops used for repeated executions of statements. In R language, while loop is often used by the programmers. The structure of the while loop in R language is as simple as in other programming languages. The skeleton structure of a while loop consists of aconstraint condition to start with. This condition is given after the keyword “while”. Programming statements are written inside while loop brackets, for which repetitive execution is to be carried out till it meets the constraint condition.

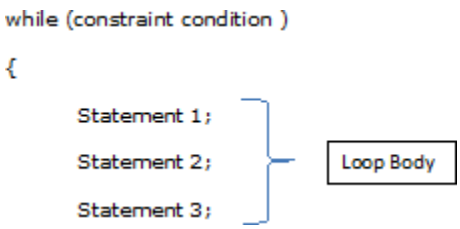
Let us look at the syntax used for while loop in R language:The syntax is as:

```
while (constraint condition) // while is a keyword

  //returns bool (true/false) value

{ //opening curly brackets

//Statements

} // closing curly brackets
```

The output or the evaluation result of the given condition is a Boolean valuethat is either true or false. Based on this Boolean value, code control enters or quit the loop to execute the next statement. If the constraint condition is not fulfilled, code control goes inside the loop and executes the statements written in the while loop. An execution of a while loop is known as an iteration.

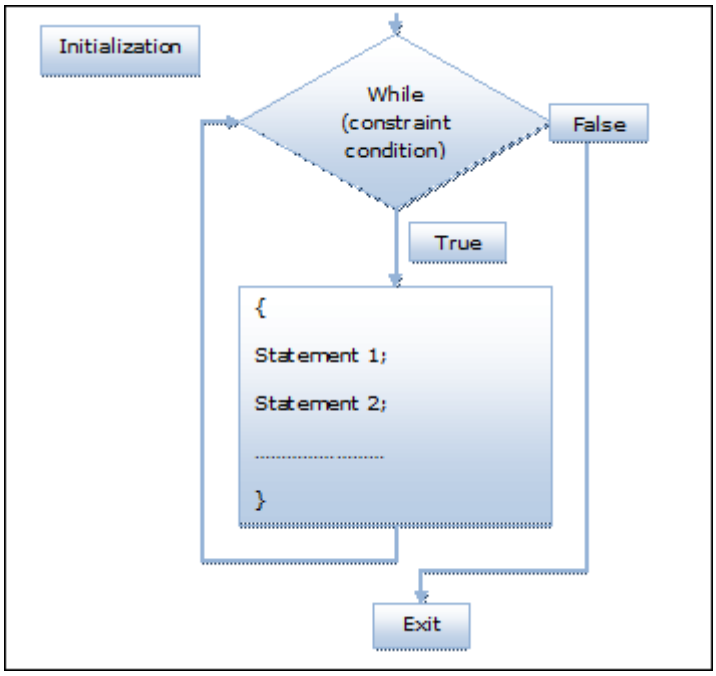
Once the iteration is completed, the control again goes back to the while keyword and checks the constraint condition. Based on the result, it jumpsout the while loop or goes inside it to execute the statements in the loop.

Code snippet of while loop:



}

A visual representation of while loop:



1. Initialization constraint variable:

Initialization is the first point from where the loop structure starts. In the above diagram, “Initialization” depicts the initialization of constraint condition. It is very much important to initialize a constraint variable before using it in a while loop. If not, it can even cause a program to crash or there can be a situation where while loop goes into the definite loop sinceconstraint variable picks up a garbage value from the memory location it is defined at.

2. Condition check:

Constraint condition is a condition where the while loop is terminated. In the above diagram, the value of constraint condition is checked. If the constraint condition is not satisfied, code control enters inside the loop and executes the statement inside the code. While loop iterations are executeduntil constraint condition is satisfied.

3. Termination of loop:

Termination is a state, where the constraint condition given is satisfied. Once the constraint condition is satisfied, the while loop terminates andcode control stops the execution of while loop.

To get the deeper understanding of while loop in R language, let us look at afew examples.

The examples below are explained in the order of ascending order ofcomplexity.

A code snippet in R using while loop to print the even numbers:

```
x<-0;

while (x < 10)

{

x<- x+4;

print (x);

}
```

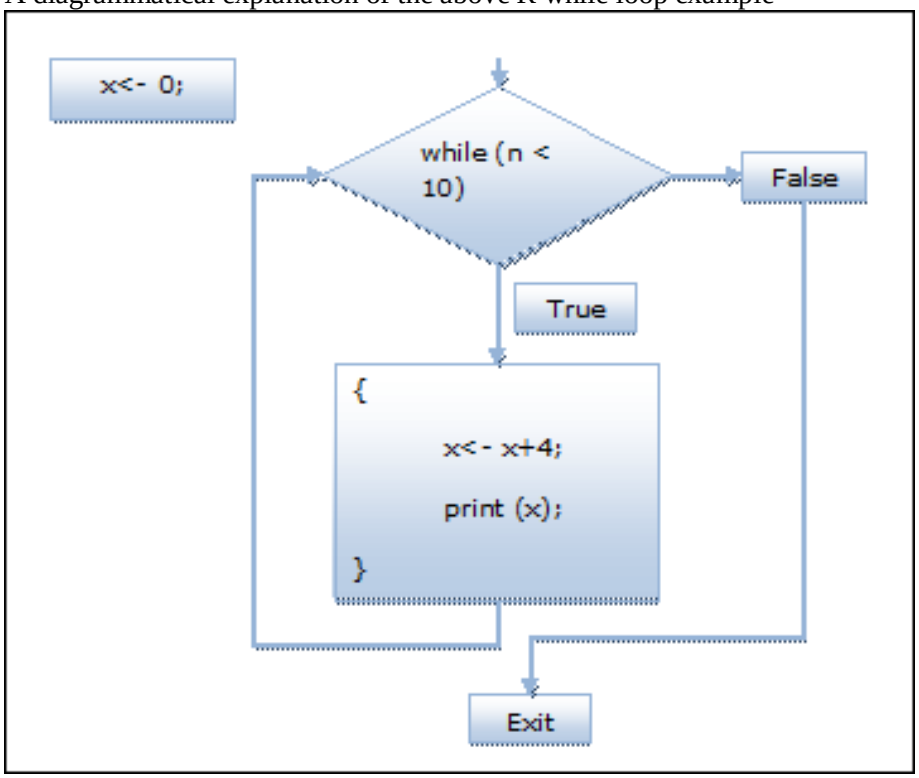
Answer: R

4

8

12

A diagrammatical explanation of the above R while loop example



Here is the explanation of the above example:

In the above diagram, a statement x<-0 depicts the initialization of constraint variable. A statement while (x < 10) shows a while loop statement with a constraint condition where x < 10 servers as a constraintcondition.

Here,

x < x+4 and

print (x) are the execution statements inside the while loop.

At the first iteration, the constraint condition will not be satisfied since x isinitialized with 0 (zero) value. Hence, code control will go inside the while loop and execute the statements inside the while loop.

At the third iteration of while loop, value of x will become 12. Hence, on thefourth iteration of while loop, code control won't go inside the loop and willterminate the while loop.

Usage of While Loop with Unknown Conditions:

Sometimes, you may come across situations where you should apply a whileloop at a place in the code. However, you will not know the exact constraint condition to apply due to the unknown nature of the result. In such cases, you can use the constraint condition "true" for the while loop. I.e. you can write your while loop as shown below:

```
While (true)

{

//Statements

}
```

While loop with constraint conditions “true” can be used in places of for loop with the dynamically allocated memory. This can in turn save the issueof for loop performance with dynamically allocated vectors.

Breaking While Loop

As we learnt, constraint condition is a way to break a while loop. However, that’s not the only way to break while loop in R. Usage of break statements or next statements can help with breaking while loops. Here is an exampleof breaking a while loop with break statement.

```
x<-0;

while (x < 10)

{

    x <- x + 4;

    print (x);

    if ( x = 8)

    {

break;

    }

}
```

In above example, code control would come out of the while loop as soon asthe value of x becomes 8, which in above condition may become a2nd whileloop iteration.

Here’s the output of the above loop. With the use of break statement in thiscase, while loop will be terminated before its constraint condition is satisfied.

Output:

4

8

Error Conditions With While Loop

As per the expert programmers, constraint conditions use in the while loopsare the very essential sources of error conditions in the while. Generally, thenature of errors related to constraint condition remains semantic, so it becomes difficult to detect such errors. Subsequently, they may produce false results as well. To avoid such

errors, R language programmers must ensure that constraint variables are being processed inside the while loop. Such errors can even lead to infinite execution of while loop.

Having said that, it is important to mention that a while loop is a concept widely accepted as an option to execute infinite conditions in the real world.A real-time example of such conditions is a data server where infinite server execution is required.

In order to learn while loops in R language, you have to learn the fundamentals of R language. One way to do that is to take a tutorial in **DataAnalysis with R.**

Important Points For While Loop Implementation

1. You must learn thebasics of loops in R before starting to learn the while loop in R language. It's an advanced concept, not the first thingyou should try.
2. It is essential to initialize a constraint condition variable beforeexecution of while loop. Otherwise, there are chances that a constraint variable will pick up a garbage value from allocated memory location and create an infinite while loop condition.
3. Please make sure that a constraint variable is being processed within the while loop to meet the constraint condition, to avoid the infinite execution of the while loop.
4. While loop is the widely accepted way of executing infinite loops inthe real world.
5. Constraint condition is not the only way to come out of a while loop.Use of break conditions can also help break the while loop.

Loop over a list

Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes_list) {
  print(p)
}

# loop version 2
for (i in 1:length(primes_list)) {
  print(primes_list[[i]])
}
```

Notice that you need double square brackets - [[ ]] - to select the list elements in loop version 2.

Suppose you have a list of all sorts of information on New York City: its population size, the names of the boroughs, and whether it is the capital of the United States. We've already defined a list nyc containing this information

A function is a set of statements organized together to perform a specific task. R has a largenumber of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, alongwith arguments that may be necessary for the function to accomplish the actions.

Thefunction in turn performs its task and returns control to the interpreter as well as anyresult which may be stored in other objects.

FunctionDefinition

An R function is created by using the keyword **function**. The basic syntax of an R functiondefinition is as follows:

```
function_name <-
   function(arg_1,arg_2,...) {
   Function body
}
```

FunctionComponents

The different parts of a function are:

- **Function Name:** This is the actual name of the function. It is stored in R environmentas an object with this name.

Arguments: An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

- **Function Body:** The function body contains a collection of statements that defineswhat the function does.

- **Return Value:** The return value of a function is the last expression in the functionbody to be evaluated.

R has many **in-built** functions which can be directly called in the program without definingthem first. We can also create and use our own functions referred as **user defined** functions.

Built-inFunction

Simple examples of in-built functions are seq(), mean(), max(), sum(x)and paste(...) etc.They are directly called by user written programs. You can refer most widely used R functions.

```
# Create a sequence of numbers from
32 to 44.print(seq(32,44))


# Find mean of numbers from 25 to
82.print(mean(25:82))


# Find sum of numbers frm 41
to 68.print(sum(41:68))
```


When we execute the above code, it produces the following result:

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-definedFunction

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function iscreated and used.

```
# Create a function to print squares of numbers in
sequence.new.function <- function(a) {
  for(i in 1:a) {
          b <-
          i^2
          print(
          b)
          }
                  }
```

CallingaFunction

```
 # Create a function to print squares of numbers in sequence.
```

```
new.function <-
  function(a) {
for(i in 1:a) {
          b <-
          i^2
          print(
          b)
          }
                  }

# Call the function new.function supplying 6 as an
argument. new.function(6)
```

When we execute the above code, it produces the following result:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an
argument.new.function <-
function() {
          for(i in
          1:5) {
          print(i^2)
          }
                 }
```

When we execute the above code, it produces the following result:

```
[1] 1
[1] 4
[1] 9
```

```
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with
arguments.new.function <-
function(a,b,c) {
          result <-
          a*b+c
          print(result
          )
                    }

# Call the function by position of
arguments.new.function(5,3,11)

# Call the function by names of the
arguments.new.function(a=11,b=5,c=3)
```

When we execute the above code, it produces the following result:

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function <- function(a = 3,b =6)
{
          result <-
          a*b
          print(resu
          lt)
                    }
```

```
# Call the function without giving
any argument.new.function()

# Call the function with giving new values of
the argument.new.function(9,5)
```

When we execute the above code, it produces the following result:

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only whenneeded by the function body.

```
# Create a function with
arguments.new.function <-
function(a, b) {
       print(a^
       2)
       print(a)
       print(b)
                    }

# Evaluate the function without supplying one of the
arguments.new.function(6)
```

When we execute the above code, it produces the following result: R

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```