```
> rep_len(1:3, 9)
[1] 1 2 3 1 2 3 1 2 3
```

2.3. Matrices and Arrays

A matrix is a collection of data elements with the same basic type arranged in a two-dimensional rectangular layout. An array consists of multidimensional rectangular data. Matrices are special cases of two-dimensional arrays. To create an array the *array()* function can be used and a vector of values and vector of dimensions are passed to it.

```
> x <- array(1:24, dim = c(4, 3, 2),
            dimnames = list(c("a", "b", "c", "d"), c("e", "f", "g"), c("h", "i")))
> x
, , h
  e f  g
a 1 5  9
b 2 6 10
c 3 7 11
d 4 8 12
, , i
   e  f  g
a 13 17 21
b 14 18 22
c 15 19 23
d 16 20 24
```

The syntax for creating matrices is using the function *matrix()* and passing the *nrow* or *ncol* argument instead of the *dim* argument in the arrays. A matrix can also be created using the *array()* function where the dimension of the array is two.

```
> m <- matrix(1:12, nrow = 3, dimnames = list(c("a", "b", "c"), c("d", "e", "f", "g")))
> m
```

```
   d e f  g
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
> m1 <- array(1:12, dim = c(3,4),
dimnames = list(c("a", "b", "c"), c("d", "e", "f", "g")))
> m1
   d e f  g
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
```

The argument *byrow = TRUE* in the *matrix()* function assigns the elements row wise. If this argument is not specified, by default the elements are filled column wise.

```
> m <- matrix(1:12, nrow = 3, byrow = TRUE,
                              dimnames = list(c("a", "b", "c"), c("d", "e", "f", "g")))
```

The *dim()* function returns the dimensions of an array or a matrix. The functions *nrow()* and *ncol()* returns the number of rows and number of columns of a matrix respectively.

```
> dim(x)
[1] 4 3 2
> dim(m)
[1] 3 4
> nrow(m)
[1] 3
> ncol(m)
[1] 4
```

The *length()* function also works for matrices and arrays. It is also possible to assign new dimension for a matrix or an array using the *dim()* function.

```
> length(x)
[1] 24
> length(m)
[1] 12
> dim(m) <- c(6,2)
```

The functions *rownames()*, *colnames()* and *dimnames()* can be used to fetch the row names, column names and dimension names of matrices and arrays respectively.

```
> rownames(m1)
[1] "a" "b" "c"
> colnames(m1)
[1] "d" "e" "f" "g"
> dimnames(x)
[[1]]
[1] "a" "b" "c" "d"
[[2]]
[1] "e" "f" "g"
[[3]]
[1] "h" "i"
```

It is possible to extract the element at the n^{th} row and m^{th} column using the expression *M[n, m]*. The entire n^{th} row can be extracted using *M[n,]* and similarly, the m^{th} column can be extracted using *M[,m]*. Also, it is possible to extract more than one column or row.

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

```
> M[,3]
[1] 3 6 9
> M[,c(1,3)]
  [,1] [,2]
[1,]    1    3
[2,]    4    6
[3,]    7    9
> M[c(1,3),]
  [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
```

The matrix transpose is constructed by interchanging its rows and columns using the function *t()*.

```
> t(M)
  r1 r2 r3
c1  1  4  7
c2  2  5  8
c3  3  6  9
```

The columns of two matrices can be combined using the *cbind()* function and similarly the rows of two matrices can be combined using the *rbind()* function.

```
> M1 = matrix(c(2,4,6,8,10,12), nrow=3, ncol=2)
> M1
  [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
> M2 = matrix(c(3,6,9), nrow=3, ncol = 1)
> M2
```

```
  [,1]
[1,]    3
[2,]    6
[3,]    9
> cbind(M1, M2)
  [,1] [,2] [,3]
[1,]    2    8    3
[2,]    4   10    6
[3,]    6   12    9
> M3 = matrix(c(4,8), nrow=1, ncol=2)
> M3
  [,1] [,2]
[1,]    4    8
> rbind(M1, M3)
  [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
[4,]    4    8
```

A matrix can be deconstructed using the *c()* function which combines all column vectors into one.

```
> c(M1)
[1]  2  4  6  8 10 12
```

The arithmetic operators "+", "- ", "* ", "/ " work element wise on matrices and arrays. But the condition is that the matrices or arrays should be of conformable sizes. The matrix multiplication is done using the operator "%*%".

```
> M1 = matrix(c(2,4,6,8,10,12), nrow=3, ncol=2)
> M1
```

```
  [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12
> M2 = matrix(c(3,6,9,11,1,5), nrow=3, ncol = 2)
> M2
  [,1] [,2]
[1,]    3   11
[2,]    6    1
[3,]    9    5
> M1 + M2
  [,1] [,2]
[1,]    5   19
[2,]   10   11
[3,]   15   17
> M1 * M2
  [,1] [,2]
[1,]    6   88
[2,]   24   10
[3,]   54   60
> M2 = matrix(c(3,6,9,11), nrow=2, ncol = 2)
> M2
  [,1] [,2]
[1,]    3    9
[2,]    6   11
> M1 %*% M2
  [,1] [,2]
[1,]   54  106
```

```
[2,]   72  146
[3,]   90  186
```

The power operator "^" also works element wise on matrices. To find the inverse of a matrix the function *solve()* can be used.

```
> M2
  [,1] [,2]
[1,]    3    9
[2,]    6   11
> M2^-1
    [,1]      [,2]
[1,] 0.3333333 0.11111111
[2,] 0.1666667 0.09090909
> solve(M2)
    [,1]      [,2]
[1,] -0.5238095  0.4285714
[2,]  0.2857143 -0.1428571
```

2.4. Lists

Lists allow us to combine different data types in a single variable. Lists can be created using the *list()* function. This function is similar to the *c()* function. The contents of a list are just listed within the *list()* function as arguments separated by a comma. List elements can be a vector, matrix or a function. It is possible to name the elements of the list while creation or later using the *names()* function.

```
> L <- list(c(9,1, 4, 7, 0), matrix(c(1,2,3,4,5,6), nrow = 3))
> L
[[1]]
[1] 9 1 4 7 0
[[2]]
```

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

> names(L) <- c("Num", "Mat")

> L
$Num
[1] 9 1 4 7 0
$Mat
  [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> L <- list(Num = c(9,1, 4, 7, 0), Mat = matrix(c(1,2,3,4,5,6), nrow = 3))
> L
$Num
[1] 9 1 4 7 0
$Mat
  [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Lists can be nested. That is a list can be an element of another list. But, vectors, arrays and matrices are not recursive/nested. They are atomic. The functions *is.recursive()* and *is.atomic()* shows if a variable type is recursive or atomic respectively.

```
> is.atomic(list())
[1] FALSE
```

```
> is.recursive(list())
[1] TRUE
> is.atomic(L)
[1] FALSE
> is.recursive(L)
[1] TRUE
> is.atomic(matrix())
[1] TRUE
> is.recursive(matrix())
[1] FALSE
```

The *length()* function works on list like in vectors and matrices. But, the *dim()*, *nrow()* and *ncol()* functions returns only *NULL*.

```
> length(L)
[1] 2
> dim(L)
NULL
> nrow(L)
NULL
> ncol(L)
NULL
```

Arithmetic operations in list are possible only if the elements of the list are of the same data type. Generally, it is not recommended. As in vectors the elements of the list can be accessed by indexing them using the square brackets. The index can be a positive number, or a negative number, or element names or logical values.

```
> L1 <- list(l1 = c(8, 9, 1), l2 = matrix(c(1,2,3,4), nrow = 2),
                                l3 = list( l31 = c("a", "b"), l32 = c(TRUE, FALSE) ))
> L1
$l1
```

```
[1] 8 9 1
$l2
  [,1] [,2]
[1,]    1    3
[2,]    2    4
$l3
$l3$l31
[1] "a" "b"
$l3$l32
[1]  TRUE FALSE

> L1[1:2]
$l1
[1] 8 9 1
$l2
  [,1] [,2]
[1,]    1    3
[2,]    2    4

> L1[-3]
$l1
[1] 8 9 1
$l2
  [,1] [,2]
[1,]    1    3
[2,]    2    4

> L1[c("l1", "l2")]
$l1
[1] 8 9 1
```

```
$l2
  [,1] [,2]
[1,]    1    3
[2,]    2    4

> L1[c(TRUE, TRUE, FALSE)]
$l1
[1] 8 9 1
$l2
  [,1] [,2]
[1,]    1    3
[2,]    2    4
```

A list is a generic vector containing other objects.

```
> a = c(4,8,12)
> b = c("abc", "def", "ghi", "jkl", "mno")
> d = c(TRUE, FALSE)
> t = list(a, b, d, 5)
```

The list *t* contains copies of the vectors a, b and d. A list slice is retrieved using single square brackets *[]*. In the below, *t[2]* contains a slice and a copy of *b*. Slice can also be retrieved with multiple members.

```
> t[2]
[[1]]
[1] "abc" "def" "ghi" "jkl" "mno"
> t[c(2,4)]
[[1]]
[1] "abc" "def" "ghi" "jkl" "mno"
[[2]]
[1] 5
```

To reference a list member directly double square bracket *[[]]* is used. Thus *t[[2]]* retrieves the second member of the list *t*. This results in a copy of *b*, but not a slice of *b*. It is also possible to modify the contents of the elements directly, but the contents of *b* are unaffected.

```
> t[[2]]
[1] "abc" "def" "ghi" "jkl" "mno"
> t[[2]][1] = "qqq"
> t[[2]]
[1] "qqq" "def" "ghi" "jkl" "mno"
> b
[1] "abc" "def" "ghi" "jkl" "mno"
```

We can assign names to the list members and reference lists by names instead of numeric indexes. A list of two members is given as example below with the member names as "*first*" and "*second*". The list slice containing the member "*first*" can be retrieved using the square brackets *[]* as shown below.

```
> l = list(first=c(1,2,3), second=c("a","b", "c"))
> l
$first
[1] 1 2 3
$second
[1] "a" "b" "c"
> l["first"]
$first
[1] 1 2 3
```

The named list member can also be directly referenced with the $ operator or double square brackets *[[]]* as below.

```
> l$first
[1] 1 2 3
```

```
> l[["first"]]
[1] 1 2 3
```

A vector can be converted to a list using the function *as.list()*. Similarly, a list can be converted into a vector, provided the list contains scalar elements of the same type. This is done using the conversion functions such as *as.numeric()*, *as.character()* and so on. If a list consists of non-scalar elements, but if they are of the same type, then it can be converted into a vector using the function *unlist()*.

```
> v <- c(7, 3, 9, 2, 6)

> as.list(v)
[[1]]
[1] 7
[[2]]
[1] 3
[[3]]
[1] 9
[[4]]
[1] 2
[[5]]
[1] 6

> L <- list(3, 7, 8, 12, 14)
> as.numeric(L)
[1]  3  7  8 12 14
> L1 <- list("aaa", "bbb", "ccc")
> L1
[[1]]
[1] "aaa"
[[2]]
[1] "bbb"
```

```
[[3]]
[1] "ccc"

> as.character(L1)
[1] "aaa" "bbb" "ccc"
> L1 <- list(l1 = c(78, 90, 21), l2 = c(11,22,33,44,55))
> L1
$l1
[1] 78 90 21
$l2
[1] 11 22 33 44 55

> unlist(L1)
l11 l12 l13 l21 l22 l23 l24 l25
 78  90  21  11  22  33  44  55
```

The *c()* function can also be used to combine lists as we do for vectors.

```
> L1 <- list(l1 = c(78, 90, 21), l2 = c(11,22,33,44,55))
> L2 <- list("aaa", "bbb", "ccc")
> c(L1, L2)
$l1
[1] 78 90 21
$l2
[1] 11 22 33 44 55
[[3]]
[1] "aaa"
[[4]]
[1] "bbb"
[[5]]
[1] "ccc"
```

2.5. Data Frames

A data frame is used for storing data tables. They store spread-sheet like data. It is a list of vectors of equal length (not necessarily of the same basic data type). Consider a data frame *df1* consisting of three vectors a, b, and d.

```
> a = c(1, 2, 3)
> b = c("a", "b", "c")
> d = c(TRUE, FALSE, TRUE)
> df1 = data.frame(a, b, d)
> df1
      a  b     d
1     1  a  TRUE
2     2  b FALSE
3     3  c  TRUE
```

By default the row names are automatically numbered from 1 to the number of rows in the data frame. It is also possible to provide row names manually using the row.names argument as below.

```
> df1 = data.frame(a, b, d, row.names = c("one", "two", "three"))
> df1
      a  b     d
one   1  a  TRUE
two   2  b FALSE
three 3  c  TRUE
```

The functions rownames(), colnames(), dimnames(), nrow(), ncol() and dim() can be applied on the data frames as below. The length() and names() function, returns the same result as that of ncol() and colnames() respectively.

```
> rownames(df1)
[1] "one"   "two"   "three"
> colnames(df1)
```

```
[1] "a" "b" "d"
> dimnames(df1)
[[1]]
[1] "one"   "two"   "three"
[[2]]
[1] "a" "b" "d"

> nrow(df1)
[1] 3
> ncol(df1)
[1] 3
> dim(df1)
[1] 3 3
> length(df1)
[1] 3
> colnames(df1)
[1] "a" "b" "d"
```

It is possible to create data frames with different length of vectors as long as the shorter ones can be recycled to match that of the longer ones. Otherwise, an error will be thrown.

```
> df2 <- data.frame(x = 1, y = 2:3, y = 4:7)
> df2
  x y y.1
1 1 2   4
2 1 3   5
3 1 2   6
4 1 3   7
```

The argument check.names can be set as FALSE so that a data frame will not look for valid column names.

```
> df3 <- data.frame("BaD col" = c(1:5), "!@#$%^&*" = c("aaa"))
> df3
  BaD.col X.....
1       1   aaa
2       2   aaa
3       3   aaa
4       4   aaa
5       5   aaa
```

There are many built-in data frames available in R (example – mtcars). When this data frame is invoked in R tool, it produces the below result.

```
> mtcars
                   mpg  cyl  disp  hp   drat  wt ...
Mazda RX4          21.0 6    160   110  3.90  2.62  ...
Mazda RX4 Wag      21.0 6    160   110  3.90  2.88  ...
Datsun 710         22.8 4    108   93   3.85  2.32  ...
.....
```

The top line contains the header or the column names. Each row denotes a record or a row in the table. A row begins with the name of the row. Each data member of a row is called a cell. To retrieve a cell value, we enter the row and the column number of the cell in square brackets [] separated by a comma. The cell value of the second row and third column is retrieved as below. The row and the column names can also be used inside the square brackets [] instead of the row and column numbers.

```
> mtcars[2, 3]
[1] 160
> mtcars["Mazda RX4 Wag", "disp"]
[1] 160
```

The *nrow()* function gives the number of rows in a data frame and the *ncol()* function gives the number of columns in a data frame. To get the preview or the first few records of a data frame along with the header the *head()* function can be used.

```
> nrow(mtcars)
[1] 32
> ncol(mtcars)
[1] 11
> head(mtcars)
                   mpg  cyl  disp  hp   drat  wt    ...
Mazda RX4          21.0 6    160   110  3.90  2.62 ...
.....
```

To retrieve a column from a data frame we use double square brackets *[[]]* and the column name or the column number inside the *[[]]*. The same can be achieved by making use of the $ symbol as well. This same result can also be achieved by using single brackets *[]* by mentioning a comma instead of the row name / number and using the column name / number as the second index inside the *[]*.

```
> mtcars[["hp"]]
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 ....
> mtcars[[4]]
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 ....
> mtcars$hp
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 ....
> mtcars[,"hp"]
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 ....
> mtcars[,4]
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 ....
```

Similarly, if we use the column name or the column number inside a single square bracket *[]*, we get the below result.

```
> mtcars[4]
                   hp
Mazda RX4          110
```

```
Mazda RX4 Wag          110
Datsun 710             93
....
> mtcars[c("mpg","hp")]
                       mpg   hp
Mazda RX4              21.0  110
Mazda RX4 Wag          21.0  110
Datsun 710             22.8  93
....
```

To retrieve a row from a data frame we use the single square brackets [] only by mentioning the row name / number as the first index inside [] and a comma instead of the column name / number.

```
> mtcars[6,]
                mpg   cyl   disp   hp   drat   wt....
Valiant         18.1  6     225    105  2.76   3.46....

> mtcars[c(6,18),]
                mpg   cyl   disp   hp   drat   wt....
Valiant         18.1  6     225    105  2.76   3.46....
Fiat 128        32.4  4     78.7   66   4.08   2.20....
> mtcars["Valiant",]
                mpg   cyl   disp   hp   drat   wt....
Valiant         18.1  6     225    105  2.76   3.46....

> mtcars[c("Valiant","Fiat 128"),]
                mpg   cyl   disp   hp   drat   wt....
Valiant         18.1  6     225    105  2.76   3.46....
Fiat 128        32.4  4     78.7   66   4.08   2.20....
```

If we need to fetch a subset of a data frame by selecting few columns and specifying conditions on the rows, we can use the subset() function to do this. This function takes the arguments, the data frame, the condition to be applied on the rows and the columns to be fetched.

```
> x <- c("a", "b", "c", "d", "e", "f")
> y <- c(3, 4, 7, 8, 12, 15)
> z <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE)
> D <- data.frame(x, y, z)
> D
```

	x	y	z
1	a	3	TRUE
2	b	4	TRUE
3	c	7	FALSE
4	d	8	TRUE
5	e	12	FALSE
6	f	15	TRUE

```
> subset(D, y<10 & z, x)
```

	x
1	a
2	b
4	d

As we have for matrices the transpose of a data frame can be obtained using the t() function as below.

```
> t(D)
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]
x	"a"	"b"	"c"	"d"	"e"	"f"
y	" 3"	" 4"	" 7"	" 8"	"12"	"15"
z	" TRUE"	" TRUE"	"FALSE"	" TRUE"	"FALSE"	" TRUE"

The functions *rbind()* and *cbind()* can also be applied on the data frames as we do for the matrices. The only condition for *rbind()* is that the column names should match, but for *cbind()* it does not check even if the column names are duplicated.

```
> x1 <- c("aaa", "bbb", "ccc", "ddd", "eee", "fff")
> y1 <- c(9, 12, 17, 18, 23, 32)
> z1 <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
> E <- data.frame(x1, y1, z1)
> E
  x1  y1  z1
1 aaa  9 TRUE
2 bbb 12 FALSE
3 ccc 17 TRUE
4 ddd 18 FALSE
5 eee 23 TRUE
6 fff 32 FALSE
> cbind(D, E)
  x y     z  x1 y1    z1
1 a 3  TRUE aaa  9  TRUE
2 b 4  TRUE bbb 12 FALSE
3 c 7 FALSE ccc 17  TRUE
4 d 8  TRUE ddd 18 FALSE
5 e 12 FALSE eee 23  TRUE
6 f 15  TRUE fff 32 FALSE
> F <- data.frame(x, y, z)
> F
```

```
    x  y     z
1   a  9  TRUE
2   b 12 FALSE
3   c 17  TRUE
4   d 18 FALSE
5   e 23  TRUE
6   f 32 FALSE
> rbind(D, F)
    x  y     z
1   a  3  TRUE
2   b  4  TRUE
3   c  7 FALSE
4   d  8  TRUE
5   e 12 FALSE
6   f 15  TRUE
7   a  9  TRUE
8   b 12 FALSE
9   c 17  TRUE
10  d 18 FALSE
11  e 23  TRUE
12  f 32 FALSE
```

The merge() function can be applied to merge two data frames provided they have common column names. By default, the merge() function does the merging based on all the common columns, otherwise one of the common column name has to be specified.

```
> merge(D, F, by = "x", all = TRUE)
```

	x	y.x	z.x	y.y	z.y
1	a	3	TRUE	9	TRUE
2	b	4	TRUE	12	FALSE
3	c	7	FALSE	17	TRUE
4	d	8	TRUE	18	FALSE
5	e	12	FALSE	23	TRUE
6	f	15	TRUE	32	FALSE

The functions colSums(), colMeans(), rowSums() and rowMeans() can be applied on the data frames that have numeric values as below.

```
> x <- c(5, 6, 7, 8)
> y <- c(15, 16, 17, 18)
> z <- c(25, 26, 27, 28)
> G <- data.frame(x, y, z)
> G
```

	x	y	z
1	5	15	25
2	6	16	26
3	7	17	27
4	8	18	28

```
> colSums(G[, 1:2])
```

x	y
26	66

```
> colMeans(G[, 1:3])
```

x	y	z
6.5	16.5	26.5

```
> rowSums(G[1:3, ])
  1   2   3
 45  48  51

> rowMeans(G[2:4, ])
  2   3   4
 16  17  18
```

2.6. Factors

Factors are used to store categorical data like gender ("Male" or "Female"). They behave sometimes like character vectors and sometimes like integer vectors based on the context.

Factors stores categorical data and they behave like strings sometimes and integers sometimes. Consider a data frame that stores the weight of few males and females. In this case the column that stores the gender is a factor as it stores categorical data. The choices "female" and "male" are called the levels of the factor. This can be viewed by using the *levels()* function and *nlevels()* function.

```
> weight <- data.frame(wt_kg = c(60,82,45, 49,52,75,68),
        gender = c("female","male", "female", "female", "female", "male", "male"))
> weight
  wt_kg gender
1    60 female
2    82   male
3    45 female
4    49 female
5    52 female
6    75   male
7    68   male

> weight$gender
```

```
[1] female male   female female female male   male
Levels: female male
> levels(weight$gender)
[1] "female" "male"
> nlevels(weight$gender)
[1] 2
```

At the atomic level a factor can be created using the *factor()* function, which takes a character vector as the argument.

```
> gender <- factor(c("female", "male", "female", "female", "female", "male", "male"))
> gender
[1] female male   female female female male   male
Levels: female male
```

The levels argument can be used in the *factor()* function to specify the levels of the factor. It is also possible to change the levels once the factor is created. This is done using the function *levels()* or the function *relevel()*. The function *relevel()* just mentions which level comes first.

```
> gender <- factor(c("female", "male", "female", "female", "female",
                     "male", "male"), levels = c("male", "female"))
> gender
[1] female male   female female female male   male
Levels: male female
> levels(gender) <- c("F", "M")
> gender
[1] M F M M M F F
Levels: F M
> relevel(gender, "M")
[1] M F M M M F F
Levels: M F
```

It is possible to drop a level from a factor using the function *droplevels()* when the level is not in use as in the example below. [*Note:* the function *is.na()* is used to remove the missing value].

```
> diet <- data.frame(eat = c("fruit", "fruit", "vegetable", "fruit"),
                                    type = c("apple", "mango", NA, "papaya"))
> diet
          eat       type
1       fruit      apple
2       fruit      mango
3   vegetable       <NA>
4       fruit     papaya
> diet <- subset(diet, !is.na(type))
> diet
          eat    type
1       fruit   apple
2       fruit   mango
4       fruit   papaya
> diet$eat
[1] fruit fruit fruit
Levels: fruit vegetable
> levels(diet)
NULL
> levels(diet$eat)
[1] "fruit"    "vegetable"
> unique(diet$eat)
[1] fruit
Levels: fruit vegetable
```

```
> diet$eat <- droplevels(diet$eat)
> levels(diet$eat)
[1] "fruit"
```

In some cases, the levels need to be ordered as in rating a product or course. The ratings can be "*Outstanding*", "*Excellent*", "*Very Good*", "*Good*", "*Bad*". When a factor is created with these levels, it is not necessary they are ordered. So, to order the levels in a factor, we can either use the function *ordered()* or the argument *ordered = TRUE* in the *factor()* function. Such ordering can be useful when analysing survey data.

```
> ch <- c("Outstanding", "Excellent", "Very Good", "Good", "Bad")
> val <- sample(ch, 100, replace = TRUE)
> rating <- factor(val, ch)
> rating
  [1] Outstanding Bad       Outstanding Good      Very Good   Very Good
  [7] Excellent   Outstanding Bad       Excellent   Very Good   Bad
...
Levels: Outstanding Excellent Very Good Good Bad
> is.factor(rating)
[1] TRUE
> is.ordered(rating)
[1] FALSE
> rating_ord <- ordered(val, ch)
> is.factor(rating_ord)
[1] TRUE
> is.ordered(rating_ord)
[1] TRUE
> rating_ord
```

```
 [1] Outstanding Bad       Outstanding Good      Very Good  Very Good
 [7] Excellent  Outstanding Bad       Excellent  Very Good  Bad
...
Levels: Outstanding < Excellent < Very Good < Good < Bad
```

Numeric values can be summarized into factors using the *cut()* function and the result can be viewed using the *table()* function which lists the count of numbers in each category. For example let us consider the variable *age* which has the numeric values of ages. These ages can be grouped using the *cut()* function with an interval of 10 and the result is a factor *age_group*.

```
> age <- c(18,20, 31, 32, 33, 35, 41, 38, 45, 48, 51, 27, 29, 42, 39)
> age_group <- cut(age, seq.int(15, 55, 10))
> age
 [1] 18 20 31 32 33 35 41 38 45 48 51 27 29 42 39
> age_group
 [1] (15,25] (15,25] (25,35] (25,35] (25,35] (25,35] (35,45] (35,45] (35,45] (45,55]
[11] (45,55] (25,35] (25,35] (35,45] (35,45]
Levels: (15,25] (25,35] (35,45] (45,55]
> table(age_group)
age_group
(15,25] (25,35] (35,45] (45,55]
      2       6       5       2
```

The function *gl()* can be used to create a factor, which takes the first argument that tells how many levels the factor contains and the second argument that tells how many times each level has to be repeated as value. This function can also take the argument *labels*, which lists the names of the factor levels. The function can also be made to list alternating values of the labels as below.

```
> gl(5,3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
Levels: 1 2 3 4 5
```

```
> gl(5,3, labels = c("one", "two", "three", "four", "five"))
 [1] one   one   one   two   two   two   three three three four  four  four  five
[14] five  five
Levels: one two three four five
> gl(5,1,15)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
 Levels: 1 2 3 4 5
```

The factors thus generated can be combined using the function interaction() to get a resultant combined factor.

```
> fac1 <- gl(5,3, labels = c("one", "two", "three", "four", "five"))
> fac2 <- gl(5,1,15, labels = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j",
                                "k", "l", "m", "n", "o"))
> interaction(fac1, fac2)
 [1] one.a   one.b   one.c   two.d   two.e   two.a   three.b three.c three.d four.e
[11] four.a  four.b  five.c  five.d  five.e
75 Levels: one.a two.a three.a four.a five.a one.b two.b three.b four.b ... five.o
```

2.7. Strings

Strings are stored in character vectors. Most string manipulation functions act on character vectors. Character vectors can be created using the *c()* function by enclosing the string in double or single quotes. (Generally we follow only double quotes). The *paste()* function can be used to concatenate two strings with a space in between. If the space need not be shown, we use the function *paste0()*. To have specified separator between the two concatenated string, we use the argument sep in the *paste()* function. The result can be collapsed into one string using the collapse argument.

```
> c("String 1", 'String 2')
[1] "String 1" "String 2"
> paste(c("Pine", "Red"), "Apple")
```

```
[1] "Pine Apple" "Red Apple"
> paste0(c("Pine", "Red"), "Apple")
[1] "PineApple" "RedApple"
> paste(c("Pine", "Red"), "Apple", sep = "-")
[1] "Pine-Apple" "Red-Apple"
> paste(c("Pine", "Red"), "Apple", sep = "-", collapse = ", ")
[1] "Pine-Apple, Red-Apple"
```

The *to String()* function can be used to convert a number vector into a character vector, with the elements separated by a comma and a space. It is possible to specify the width of the print string in this function.

```
> x <- c(1:10) ^ 3
> x
 [1]    1    8   27   64  125  216  343  512  729 1000
> toString(x)
[1] "1, 8, 27, 64, 125, 216, 343, 512, 729, 1000"
> toString(x, 18)
[1] "1, 8, 27, 64, ...."
```

The *cat()* function is also similar to the *paste()* function, but there is little difference in it as shown below.

```
> cat(c("Red", "Pine"), "Apple")
Red Pine Apple
```

The *noquote()* function forces the string outputs not to be displayed with quotes.

```
> a <- c("I", "am", "a", "data", "scientist")
> a
[1] "I"       "am"      "a"       "data"    "scientist"
> noquote(a)
[1] I       am      a       data    scientist
```

The *formatC()* function is used to format the numbers and display them as strings. This function has the arguments *digits*, *width*, *format*, *flag* etc which can be used as below. A slight variation of the function *formatC()* is the function *format()* whose usage is as shown below.

```
> h <- c(4.567, 8.981, 27.772)
> h
[1]  4.567  8.981 27.772
> formatC(h)
[1] "4.567" "8.981" "27.77"
> formatC(h, digits = 3)
[1] "4.57" "8.98" "27.8"
> formatC(h, digits = 3, width = 5)
[1] " 4.57" " 8.98" " 27.8"
> formatC(h, digits = 3, format = "e")
[1] "4.567e+00" "8.981e+00" "2.777e+01"
> formatC(h, digits = 3, flag = "+")
[1] "+4.57" "+8.98" "+27.8"

> format(h)
[1] " 4.567" " 8.981" "27.772"
> format(h, digits = 3)
[1] " 4.57" " 8.98" "27.77"
> format(h, digits = 3, trim = TRUE)
[1] "4.57"  "8.98"  "27.77"
```

The *sprint()* function is also used for formatting strings and passing number values in between the strings. The argument *%s* in this function stands for a string to be passed. The argument *%d* and argument *%f* stands for integer and floating-point number. The usage of this function can be understood by the below example.

```
> x <- c(1, 2, 3)
> sprintf("The number %d in the list is = %f", x, h)
[1] "The number 1 in the list is = 4.567000"
[2] "The number 2 in the list is = 8.981000"
[3] "The number 3 in the list is = 27.772000"
```

To print a tab in between text, we can use the *cat()* function with the special character "\t" included in between the text as below. Similarly, if we need to insert a new line in between the text, we use "\n". In this *cat()* function the argument *fill = TRUE* means that after printing the text, the cursor is placed in the next line. Suppose if a back slash has to be used in between the text, it is preceded by another back slash. If we enclose the text in double quotes and if the text contains a double quote in between, it is also preceded by a back slash. Similarly, if we enclose the text in single quotes and if the text contains a single quote in between, it is also preceded by a back slash. If we enclose the text in double quotes and if the text contains a single quote in between, or if we enclose the text in single quotes and if the text contains a double quote in between, it is not a problem (No need for back slash).

```
> cat("Black\tBerry", fill = TRUE)
Black    Berry
> cat("Black\nBerry", fill = TRUE)
Black
Berry
> cat("Black\\Berry", fill = TRUE)
Black\Berry
> cat("Black\"Berry", fill = TRUE)
Black"Berry
> cat('Black\'Berry', fill = TRUE)
Black'Berry
> cat('Black"Berry', fill = TRUE)
Black"Berry
```

> cat("Black'Berry", fill = TRUE)

Black'Berry

The function *toupper()* and *tolower()* are used to convert a string into upper case or lower case respectively. The *substring()* or the *substr()* function is used to cut a part of the string from the given text. Its arguments are the text, starting position and ending position. Both these functions produce the same result.

> toupper("The cat is on the Wall")

[1] "THE CAT IS ON THE WALL"

> tolower("The cat is on the Wall")

[1] "the cat is on the wall"

> substring("The cat is on the wall", 3, 10)

[1] "e cat is"

> substr("The cat is on the wall", 3, 10)

[1] "e cat is"

> substr("The cat is on the wall", 5, 10)

[1] "cat is"

The function *strsplit()* does the splitting of a text into many strings based on the splitting character mentioned as argument. In the below example the splitting is done when a space is encountered. It is important to note that this function returns a list and not a character vector as a result.

> strsplit("I like Bannana, Orange and Pineapple", " ")

[[1]]

[1] "I" "like" "Bannana," "Orange" "and" "Pineapple"

In this same example if the text has to be split when a comma or space is encountered it is mentioned as ",?". This means that the comma is optional and space is mandatory for splitting the given text.

```
> strsplit("I like Bannana, Orange and Pineapple", ",? ")
[[1]]
[1] "I"        "like"     "Bannana"  "Orange"   "and"      "Pineapple"
```

The default R's working directory can be obtained using the function *getwd()* and this default directory can be changed using the function *setwd()*. The directory path mentioned in the *setwd()* function should have the forward slash instead of backward slash as in the example below.

```
> getwd()
[1] "C:/Users/admin/Documents"
> setwd("C:/Program Files/R")
> getwd()
[1] "C:/Program Files/R"
```

It is also possible to construct the file paths using the *file.path()* function which automatically inserts the forward slash between the directory names. The function *R.home()* list the home directory where R is installed.

```
> file.path("C:", "Program Files", "R", "R-3.3.0")
[1] "C:/Program Files/R/R-3.3.0"
> R.home()
[1] "C:/PROGRA~1/R/R-33~1.0"
```

Paths can also be specified by relative terms such as "." denotes current directory, ".." denotes parent directory and "~" denotes home directory. The function *path.expand()* converts relative paths to absolute paths.

```
> path.expand(".")
[1] "."
> path.expand("..")
[1] ".."
> path.expand("~")
[1] "C:/Users/admin/Documents"
```

The function *basename()* returns only the file name leaving its directory if specified. On the other hand the function *dirname()* returns only the directory name leaving the file name.

```
> filename <- "C:/Program Files/R/R-3.3.0/bin/R.exe"
> basename(filename)
[1] "R.exe"
> dirname(filename)
[1] "C:/Program Files/R/R-3.3.0/bin"
```

2.8. Dates and Times

Dates and Times are common in data analysis and R has a wide range of capabilities for dealing with dates and times.

2.8.1. Date and Time Classes

R has three date and time base classes and they are *POSIXct*, *POSIXlt* and *Date*. *POSIX* is a set of standards that defines how dates and times should be specified and "*ct*" stands for "calendar time". *POSIXlt* stores dates as a list of seconds, minutes, hours, day of month etc. For storing and calculating with dates, we can use *POSIXct* and for extracting parts of dates, we can use *POSXlt*.

The function *Sys.time()* is used to return the current date and time. This returned value is by default in the *POSIXct* form. But, this can be converted to *POSIXlt* form using the function *as.POSIXlt()*. When printed both forms of date and time are displayed in the same manner, but their internal storage mechanism differs. We can also access individual components of a *POSIXlt* date using the dollar symbol or the double brackets as shown below.

```
> Sys.time()
[1] "2017-05-11 14:31:29 IST"
> t <- Sys.time()
> t1 <- Sys.time()
> t2 <- as.POSIXlt(t1)
```

```
> t1
[1] "2017-05-11 14:39:39 IST"
> t2
[1] "2017-05-11 14:39:39 IST"
> class(t1)
[1] "POSIXct" "POSIXt"
> class(t2)
[1] "POSIXlt" "POSIXt"
> t2$sec
[1] 39.20794
> t2[["min"]]
[1] 39
> t2$hour
[1] 14
> t2$mday
[1] 11
> t2$wday
[1] 4
```

The Date class stores the dates as number of days from start of 1970. This class is useful when time is insignificant. The *as.Date()* function can be used to convert a date in other class formats to the Date class format.

```
> t3 <- as.Date(t2)
> t3
[1] "2017-05-11"
```

There are also other add-on packages available in R to handle date and time and they are *date*, *dates*, *chron*, *yearmon*, *yearqtr*, *timeDate*, *ti* and *jul*.

2.8.2. Date Conversions

In CSV files the dates will be normally stored as strings and they have to be converted into date and time using any of the packages. For this we need to parse the strings using the function *strptime()* and this returns the date of the format *POSIXlt*. The date format is specified as a string and passed as argument to the *strptime()* function. If the given string does not match the format given in the format string, then it returns *NA*.

```
> date1 <- strptime("22:15:45 22/08/2015", "%H:%M:%S %d/%m/%Y")
> date1
[1] "2015-08-22 22:15:45 IST"

> date2 <- strptime("22:15:45 22/08/2015", "%H:%M:%S %d-%m-%Y")
> date2
[1] NA
```

In the format string "*%H*" denotes hour in 24 hour system, "*%M*" denotes minutes, "*%S*" denotes second, "*%m*" denotes the number of the month, "*%d*" denotes the day of the month as number, "*%Y*" denotes four digit year.

To convert a date into a string the function *strftime()* is used. This function also takes a date formatting string as argument like *strptime()*. In the format string "*%I*" denotes hour in 12 hours system, "*%p*" denotes AM/PM, "*%A*" denotes the string of day of the week, "*%B*" denotes the string of name of the month.

```
> strftime(Sys.Date(),"It's %I:%M%p on %A %d %B, %Y.")
[1] "It's 12:00AM on Thursday 11 May, 2017."
```

2.8.3. Time Zones

It is possible to specify the time zone when parsing a date string using *strptime()* or *strftime()* functions. If this is not specified, the default time zone is taken. The functions *Sys.timezone()* and *Sys.getlocale("LC_TIME")* are used to get the default time zone of the system and the operating system respectively.