Integer Constants

In R an integer constant can be created by using suffix `L` after the constant. For example to create the integer constant 12, we use `12L`.

Examples of integer constants

```
x<-12L  # Assign an integer 12 to the variable x
x     # Display the value of x on the screen
```

```
[1] 12
```

```
typeof(x) # determines the type of storage mode of object x
```

```
[1] "integer"
```

```
mode(x)
```

```
[1] "numeric"
```

```
y<-0x11  # Assign an integer 17 from the hexadecimal representation to variable y
y
```

```
[1] 17
```

```
z<-1e4L  # Assigne 10000 as an integer to variable z
z
```

```
[1] 10000
```

Numeric Constants

Numeric constants consists of an integer part with zero or more digits followed by decimal point (optional) `.` and a fractional part with zero or more digits (optional) followed by an exponent part consisting of an `E` or `e`, an optional sign and an integer constant with zero or more digits. e.g., `2`, `35`, `0.45`, `2e-6`, `.34`, `2.456e+3` are valid numeric constants.

Examples of numeric constants

```
y<-0.45  # Assign the constant 0.45 to the variable y
y
```

```
[1] 0.45
```

```
typeof(y) #  determines the type of storage mode of object x
```

```
[1] "double"
```

```
mode(y)
```

```
[1] "numeric"
```

```
z<-2e-06
z
```

```
[1] 2e-06
```

Logical Constants

Logical constants are either `TRUE` or `FALSE`. Single character can also be used for logical constants i.e., `T` or `F` (no quotes).

Examples of logical constants

```
Result1<-T  # Assign the logical constant 'T' or 'TRUE' to Result1
Result1
```

```
[1] TRUE
```

```
Result2<-FALSE # Assign the logical constant 'FALSE' to Result2
Result2
```

```
[1] FALSE
```

Complex Constants

Complex constants are similar to the numeric constants but they are followed by `i`. Only pure imaginary numbers are complex constants. e.g. `1i`, `0i`, `2.3e-1i` are valid complex constants.

Examples of complex constants

```
z1<-1i  # Assign 1i to the variable z1
z1
```

```
[1] 0+1i
```

```
z2<-0i # Assign 0i to the variable z2
z2
```

```
[1] 0+0i
```

```
z3<-2.3e-1i # Assign 0+0.23i to the variable z3
z3
```

```
[1] 0+0.23i
```

String Constants

String constants are delimited by a pair of single (') or double quotes (") and can contain all other printable characters. e.g., `male`, `Strongly Agree`, `Pre-test`, `T` are valid string constants.

Examples of String constants

```
gender<-"male" # Assign the string "male" to the variable gender
gender
```

```
[1] "male"
```

```
Text1<-'Strongly Agree' # Assign the string "Strongly Agree" to the variable Text1
Text1
```

```
[1] "Strongly Agree"
```

```
Text2<-"Pre-test" # Assign the string "Pre-test" to the variable Text2
Text2
```

```
[1] "Pre-test"
```

```
Text3<-"T" # Assign the string "T" to the variable Text3
Text3
```

```
[1] "T"
```

Special Constants/Values

In addition to the above constants, there are four **special constants** in R. They are

- `NULL`,
- `Inf` or `-Inf`,
- `NaN` and
- `NA`.

NULL :

The constant NULL is used to indicate an empty object in R.

Examples of NULL

```
A<-NULL  # Assign NULL to the variable A
A     # Display the value of variable A
```

```
NULL
```

```
class(A) # display class of A
```

```
[1] "NULL"
```

```
typeof(A) # display  storage mode of A
```

```
[1] "NULL"
```

Infinity (Inf and -Inf)

If a computation in R results in a number that is too big, R will return positive infinity `Inf` or negative infinity `-Inf` depending upon the result. Also a non-zero (positive or negative) number divided by zero results infinity (∞ or $-\infty$). R denotes $-\infty$ by `-Inf` and ∞ by `Inf`.

Examples of Infinity

```
2^1024
```

```
[1] Inf
```

```
-2^1024
```

```
[1] -Inf
```

```
1/0
```

```
[1] Inf
```

```
-1/0
```

```
[1] -Inf
```

```
1/Inf
```

```
[1] 0
```

```
50/0
```

```
[1] Inf
```

```
-7.6/0
```

```
[1] -Inf
```

Not a Number `NaN`

R supports a special value, called `NaN`, i.e., Not a Number, which indicates that a numerical result is undefined.

Examples of `NaN`

```
0/0
```

```
[1] NaN
```

```
Inf- Inf
```

```
[1] NaN
```

```
Inf/Inf
```

```
[1] NaN
```

All above gives `NaN`, since the result cannot be defined sensibly.

Not Available `NA`

R has a particular symbol to indicate the missing value or the value which is not available. R indicate such a value by `NA`. The result of any arithmetic expression containing `NA` will produce `NA`.

Examples of `NA`

```
log(NA)  # log is built in function in R
```

```
[1] NA
```

```
NA+10 # This is an arithmetic expression
```

```
[1] NA
```

Vectors

A basic data structure in R is vector. It is a atomic data structure, where all the stored objects are of same type. Vector can be defined as a set of constants or scalars of **same type** arranged in a one-dimensional array. In R there are four type of vectors, namely,

- numeric vector (integer or double)
- character vector
- logical vector
- complex vector

Vectors are usually created with `c()` function. The `c()` function is known as **concatenation** or **combine**.

Vectors have four basic properties. They are type, length, mode and structure. Knowledge about these properties are very important while manipulating objects. These properties can be examined using the functions `typeof()`, `length()`, `mode()` and `str()`.

Property	Command	Meaning
Type	`typeof()`	Type of Storage mode of object
Length	`length()`	Number of elements in object
Mode	`mode()`	Storage mode of object
Structure	`str()`	Compactly display the structure of object

The `str()` function check the internal structure of the R object and print a preview of its content.

Creating an empty vector

An empty vector can be created with the `vector()` function. The function `vector(mode,length)`produces a vector of the given mode and length.

```
vec1<-vector(mode="numeric",length=3)
vec1
```

```
[1] 0 0 0
```

```
vec2<-vector(mode="logical",length=4)
vec2
```

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

```
vec3<-vector(mode="complex",length=2)
vec3
```

```
[1] 0+0i 0+0i
```

```
vec4<-vector(mode="character",length=5)
vec4
```

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

Numeric Vector

Numeric vectors have elements which are numeric (integer or double) values.

```
data1<-c(2L,3L,5L,7L) # store the values as integer to data1
str(data1) # display structure
```

```
 int [1:4] 2 3 5 7
```

```
length(data1)  # display length
```

```
[1] 4
```

In the above code first line create an object `data1`, as a vector, containing the entries 2, 3, 5, 7.

```r
mode(data1)  # display mode
```

```
[1] "numeric"
```

```r
typeof(data1) # display storage mode
```

```
[1] "integer"
```

```r
is.vector(data1)
```

```
[1] TRUE
```

```r
is.vector(data1,mode="logical")
```

```
[1] FALSE
```

```r
data2<-c(2,3,5,7) # Store the values as double to data2
length(data2) # display length
```

```
[1] 4
```

```r
mode(data2)   # display mode
```

```
[1] "numeric"
```

```r
typeof(data2) # display storage mode
```

```
[1] "double"
```

```r
str(data2)   # display structure
```

```
 num [1:4] 2 3 5 7
```

We can see the difference between the results of `typeof()` command for `data1` and `data2`.

Character Vectors

Character vectors have elements which are character strings, where strings are sequence of characters enclosed in double quotes, `"pre-test"`, or single quotes, `'pre-test'`. The function `nchar()` returns the length of the character strings in a character vector.

```r
result<-c("First","Second","Third","Pass","Fail","Second","Pass")
result
```

```
[1] "First"  "Second" "Third"  "Pass"   "Fail"   "Second" "Pass"
```

```r
nchar(result) # Display the length of the each character string
```

```
[1] 5 6 5 4 4 6 4
```

```r
length(result) # Display the number of element in result
```

```
[1] 7
```

```r
mode(result) # Display the type of data stored in result
```

```
[1] "character"
```

```r
typeof(result)
```

```
[1] "character"
```

```r
str(result)
```

```
 chr [1:7] "First" "Second" "Third" "Pass" "Fail" "Second" ...
```

Logical Vectors

Logical vectors have elements which are logical character strings `T` or `TRUE` and `F` or `FALSE`.

```r
x<-c(T,F,T,F,T,F)
x
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

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

```
[1] 4
```

```r
mode(y)
```

```
[1] "logical"
```

```r
typeof(y)
```

```
[1] "logical"
```

```r
str(y)
```

```
 logi [1:4] TRUE FALSE TRUE TRUE
```

Logical vectors can also be created using a logical expression on numeric vector.

```
a<-c(1,2,3,5)
mode(a)
```

```
[1] "numeric"
```

```
z<-a<2.5   # logical expression
z
```

```
[1]  TRUE  TRUE FALSE FALSE
```

```
mode(z)
```

```
[1] "logical"
```

Complex Vectors

Complex vectors have elements which are complex constants.

```
z1<-c(1i,0i,2+3i,3-4i)
z1
```

```
[1] 0+1i 0+0i 2+3i 3-4i
```

```
z2<-c(0,0,1+3i,-4i)
z2
```

```
[1] 0+0i 0+0i 1+3i 0-4i
```

```
length(z2)
```

```
[1] 4
```

```
mode(z2)
```

```
[1] "complex"
```

```
typeof(z2)
```

```
[1] "complex"
```

```
str(z2)
```

```
 cplx [1:4] 0+0i 0+0i 1+3i ...
```

Some functions associated with complex vectors are

```r
Re(z2) # gives real part of z2
```

```
[1] 0 0 1 0
```

```r
Im(z2) # gives imaginary part of z2
```

```
[1]  0  0  3 -4
```

```r
Mod(z2) # gives modulus of z2
```

```
[1] 0.000000 0.000000 3.162278 4.000000
```

```r
Conj(z2) #  gives conjugate of z2
```

```
[1] 0+0i 0+0i 1-3i 0+4i
```

Coercion in vectors

R follows two basic rules of coercion

- If a character is present, R will coerce everything else to characters
- If a vector contains logicals and numbers, R will convert the logicals to numbers (Like TRUE to 1 and FALSE to 0)

R provides a set of explicit coercion functions that allows us to convert one type of data into another

- `as.character()`
- `as.numeric()`
- `as.logical()`
- `as.complex()`

When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same type.

```r
xx<-c(1.8,"Bob")  ## xx is a character vector
xx
```

```
[1] "1.8" "Bob"
```

```r
as.numeric(xx)
```

```
Warning: NAs introduced by coercion
```

```
[1] 1.8  NA
```

```r
yy<-c(TRUE, 1.8,-1.2,FALSE,T) ## yy is a numeric vector
yy
```

```
[1]  1.0  1.8 -1.2  0.0  1.0
```

```r
as.logical(yy)
```

```
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

```r
zz<-c("Bob",TRUE) ## zz is a character vector
zz
```

```
[1] "Bob"  "TRUE"
```

```r
as.logical(zz)
```

```
[1]   NA TRUE
```

Vectors operations

Generating vectors using : operator

R provides a very useful way of generating a regular sequence of increasing or decreasing values using sequence operator `:`.

The general expression is `n1:n2`, which generates sequence of integers ranging from `n1` to `n2`.

It generates a sequence of values `n1`, `n1 + 1`, ... , up to the sequence value less than or equal to `n2`.

Examples of : operator

```r
x<-2:6   # generate sequence of numbers from 2 to 6
x
```

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

```r
y<-2.5:6
y
```

```
[1] 2.5 3.5 4.5 5.5
```

In the first case, since the starting value is integer, the sequence operator `:` generate a sequence of integers from 2 to 6, whereas in the second case the starting value is float, the sequence operator `:` generate a sequence of floats ranging from 2.5 to 6.

```r
z<-2.5:6.9
z
```

```
[1] 2.5 3.5 4.5 5.5 6.5
```

It will generate a sequence of real numbers starting with 2.5 to 6.5.

```
x<--3:5  # generate a sequence of integers in increasing order
x
```

```
[1] -3 -2 -1  0  1  2  3  4  5
```

```
x<-5:-3  # generates a sequence of integers in decreasing order
x
```

```
[1]  5  4  3  2  1  0 -1 -2 -3
```

(as precedence of -(unary minus) is higher than that of :).

Generating Vectors using seq() function

The operator : generate simple sequence of numbers in increasing or decreasing orders. The sequence of numbers with specific pattern can be generated using seq() function.

The general syntax of seq() function is

- seq(from, to) or
- seq(from, to, by=) or
- seq(from, to, length=).

Examples of seq() function

```
seq(1,8)  # generates a regular sequence from 1 to 8
```

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

```
# generates a regular sequence from 1 to 8 with increment 2
seq(1,8,by=2)
```

```
[1] 1 3 5 7
```

```
# generates a regular sequence from 8 to 1 with decrement 2
seq(8,1,by=-2)
```

```
[1] 8 6 4 2
```

```
# generates a sequence of 3 numbers from 1 to 8 with equal spacing
seq(1,8,length=3)
```

```
[1] 1.0 4.5 8.0
```

```
seq(0,2,length=5)
```

```
[1] 0.0 0.5 1.0 1.5 2.0
```

```
seq(from=0,to =2,length=10)
```

```
 [1] 0.0000000 0.2222222 0.4444444 0.6666667 0.8888889 1.1111111 1.3333333
 [8] 1.5555556 1.7777778 2.0000000
```

It will calculate `by` value as $\frac{\text{to}-\text{from}}{\text{length}-1} = 0.2222222$.

Repeating vectors using `rep()` function

A very useful function to create a vector by repeating a given number/vector with specified number of times is the `rep()` function.

The general structure of `rep()` function is

`rep(v1,n1)`.

Here vector `v1` is repeated `n1` times.

The forms of `rep()` function are

- `rep(v1, times=)`,
- `rep(v1, each=)`,
- `rep(v1, length=)`.

Examples of `rep()` function

```
rep(0,5)  # replicates 0 five times
```

```
[1] 0 0 0 0 0
```

The constant 0 is repeated 5 times.

```
rep(1:3,times=3) # replicates 1 to 3 numbers 3 times
```

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

The sequence `1:3` that is 1,2,3 is repeated 3 times.

```
rep(1:3,each=3) #each number 1 to 3 is replicated three times
```

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

Each element from the sequence 1,2,3 is repeated 3 times.

```
# generate a vector 1,2,3
x<-1:3
# vector x is replicated such that the length is five.
rep(x, length=5)
```

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

Here the vector `x` is recycled until you get the specific length.

```
# replicate each element of x twice until you get length of 4
rep(x,each=2,length=4)
```

```
[1] 1 1 2 2
```

```
# replicate each element of x twice until you get length of 8 (two elements recycled)
rep(x,each=2,length=8)
```

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

```
rep(x,each=2,length=5)  # one element recycled
```

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

```
rep(x,each=2,times=6)
```

```
 [1] 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3
[36] 3
```

```
rep(x,times=6,each=2)
```

```
 [1] 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3
[36] 3
```

```
rep(x,times=6,length=10)
```

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

```
rep(x,each=6,length=10)
```

```
 [1] 1 1 1 1 1 1 2 2 2 2
```

```
x
```

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

```
rep(x,c(2,1,3))  # 1 is replicated 2 times, and so on
```

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

Matrix

Matrix

A matrix is a two-dimensional data structure in R. It is a set of elements appearing in rows and columns. All the elements of the matrix must be of same mode (logical, numeric, complex or character). A matrix can be created with the function matrix:

`matrix(data=NA, nrow=r, ncol=c, byrow=FALSE, dimnames=NULL)`

- By default matrices are created with their values running down successive column.
- It is also possible to specify that the matrix be filled by rows using `byrow=TRUE`.
- The option dimnames allows to give names to the rows and columns.

Some attributes related to matrix (or data)

Function	Description
`class(x)`	Gives the class of object x as matrix
`typeof(x)`	Gives the type of data stored in matrix
`dim(x)`	Sets or changes the dimension of `x`
`str(x)`	Gives the structure of a matrix
`nrow(x)`	Gives the number of rows of a matrix
`ncol(x)`	Gives the number of columns of a matrix
`rownames(x)`	Gives row names of a matrix
`colnames(x)`	Gives column names of a matrix

Creating an empty matrix

An empty matrix can be created by specifying the number of rows and number of columns of a matrix. The `dim(x)` function either sets or changes the dimension of object `x`.

```
AE<-matrix(nrow=3,ncol=4)
AE
```

```
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

```
dim(AE) # dim function display the dimension of matrix
```

```
[1] 3 4
```

```
ncol(AE)  # gives the number of columns of given matrix
```

```
[1] 4
```

```
class(AE)
```

```
[1] "matrix"
```

```
typeof(AE) # In R NA are logical constant of length 1
```

```
[1] "logical"
```

```
str(AE)
```

```
 logi [1:3, 1:4] NA NA NA NA NA NA ...
```

Zero matrix can be created by specifying the element 0 and the number of rows and columns of a matrix.

```r
A0<-matrix(0,nrow=3,ncol=4)
A0
```

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

```r
dim(A0) # dim function display the dimension of matrix
```

```
[1] 3 4
```

```r
nrow(A0) # gives the number of rows of given matrix
```

```
[1] 3
```

Creating a matrix from a sequence of numbers

By default, elements of sequence can be filled in a matrix column-wise.

```r
B0<-matrix(1:9,nrow=3,ncol=3)
B0
```

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

Note: Number of elements should be in multiples of number of rows or columns, otherwise R display warning message.

Elements of sequence can also be filled in a matrix row-wise by specifying `byrow=T`.

```r
B01<-matrix(1:9,nrow=3,byrow=T)
B01
```

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

Creating a matrix from vector

Another way to create a matrix in R is to convert a vector into a matrix by specifying number of rows `nrow` or number of column `ncol`.

```r
v1<-c(1,2,3,4,5,6)
A1 <- matrix(v1, nrow = 3)
A1
```

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

```
str(A1)
```

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

Specifying `byrow=T` argument.

```
B1 <- matrix(v1, nrow = 3, byrow = T)
B1
```

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

```
nrow(B1)   # Display number of rows
```

```
[1] 3
```

```
typeof(B1)  # Display storage mode
```

```
[1] "double"
```

Creating a Matrix from vector using a dimension attributes

A matrix can also be constructed from a vector by setting the dimension attribute of given vector. Assigning a dimension to a vector treats it as a matrix of specified dimension.

```
da1<-c(1, 2, 3, 4, 5, 6) # define a vector
da1
```

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

```
dim(da1)<-c(2,3) # set dimension of vector da1 as 2 rows and 3 columns
da1
```

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

Creating a matrix with dimension names

Rows and columns of a matrix can be labelled using `dimnames()` command.

```
C1 <- matrix(c(20, 15, 10, 40), nrow = 2,
             dimnames = list(c("Male","Female"),
            c("Smoker", "Non-smoker")))
C1
```

```
       Smoker Non-smoker
Male       20         10
Female     15         40
```

```
dim(C1)
```

```
[1] 2 2
```

```
dimnames(C1)
```

```
[[1]]
[1] "Male"   "Female"

[[2]]
[1] "Smoker"     "Non-smoker"
```

Row names and column names of a matrix can also be assigned using `rownames()` and `colnames()` function.

```
da1
```

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

```
rownames(da1)<-c("R1","R2");colnames(da1)<-c("C1","C2","C3")
dimnames(da1)
```

```
[[1]]
[1] "R1" "R2"

[[2]]
[1] "C1" "C2" "C3"
```

Creating a matrix using `cbind()` and `rbind()` function

Matrix can also be created using `cbind()` (column-binding) or `rbind()` (row-binding) functions.

```
C3 <- cbind(c(1, 2), c(3, 4)) # combine coulmns
C3
```

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

```
C4 <- rbind(c(1, 2), c(3, 4))  # combine rows
C4
```

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

`cbind()` and `rbind()` can also be use to combine two or more matrices.

```
C5<-cbind(C3,C4) # combine two matrices using cbind
C5
```

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

```
C6<-rbind(C3,C4) # combine two matrices using rbind
C6
```

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

Accessing elements of matrix

Elements of a matrix can be accessed by specifying row number(s) and/or column number(s). Like

- `mat1[i,]` returns i^{th} row,
- `mat[,j]` returns j^{th} column and
- `mat1[i,j]` returns $(i, j)^{th}$ element of matrix.

```
mat1<-matrix(1:9,nrow=3,byrow=TRUE)
mat1
```

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

```
mat1[2,3]  # returns value from mat1 in the 2nd row and 3rd column
```

```
[1] 6
```

```
mat1[1,]   # returns 1st row of matrix mat1
```

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

```r
mat1[,3]   # returns 3rd column of matrix mat1
```

```
[1] 3 6 9
```

```r
mat1[1:2,3] # returns the value from mat1 in the first 2 rows ad 3rd column
```

```
[1] 3 6
```

Matrix Operations in R

- Arithmetic operations on matrices are element-wise: +, -, *,/
- Matrix multiplication: `%*%`
- Transpose of a matrix : `t()`
- To solve the system of linear equations `Ax = b`: `solve(A,b)`
- To find the inverse of a matrix A^{-1} : `solve(A)`
- To computes eigen values and eigen vectors: `eigen()`
- To compute determinant of a matrix: `det(A)`

Addition/subtraction of matrices

```r
A <- matrix(c(1, 2, 3, 4), nrow = 2)
B <-matrix(c(4, 5, 5, 3), nrow = 2)
A+B   # Addition of two matrices
```

```
     [,1] [,2]
[1,]    5    8
[2,]    7    7
```

```r
A-B   # Difference of matrix B from A
```

```
     [,1] [,2]
[1,]   -3   -2
[2,]   -3    1
```

Matrix multiplication

`A*A` gives a new matrix with element-wise multiplication, where as `A%*% A` gives actual matrix multiplication.

```r
A
```

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

```r
A*A   # Elementwise multiplication
```

```
     [,1] [,2]
[1,]    1    9
[2,]    4   16
```

```r
A
```

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

```r
A%*%A # Matrix Multiplication
```

```
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

Transpose of a matrix

```r
A <- matrix(c(1, 2, 3, 4), nrow = 2)
A
```

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

```r
t(A)   # transpose of a matrix A
```

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

Inverse of a matrix

The arithmetic operations on matrices are element-wise. Hence if we use `A^(-1)`, R will calculate a new matrix as a reciprocal of each element of matrix A.

Inverse of a square matrix can be computed using `solve()` function.

```r
A^(-1)   # Matrix of reciprocal of each elements
```

```
     [,1]      [,2]
[1,]  1.0 0.3333333
[2,]  0.5 0.2500000
```

```r
solve(A)  # Inverse of a matrix A
```

```
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Determinant of a matrix

```r
A
```

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

```r
det(A)   # determinant of Matrix A
```

```
[1] -2
```

Eigen values and eigen vectors of a matrix

```r
eigen(A)  # eigen values and eigen vectors of A
```

```
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

```r
eigen(A)$vectors # gives only eigen vectors of A
```

```
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

Array

Matrix is two dimensional object. When there are more than two dimensions, we use array to store such a data. Thus arrays are similar to matrices, but have more than two dimensions.

The general structure of `array()` function is

`array(data=NA,dim=length(data),dimnames=NULL)`

- **data:** a vector giving data to fill the array.
- **dim:** the dimension attributes for the array to be created.
- **dimnames:** dimension names.

Creating an array

```
data<-1:12
data
```

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

```
myarray<-array(data,dim=c(2,3,2))
class(myarray)
```

```
[1] "array"
```

```
mode(myarray)
```

```
[1] "numeric"
```

```
myarray
```

```
, , 1

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

, , 2

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

```
dimnames(myarray)  # display dimension names of myarray
```

```
NULL
```

Arrays with dimension names

```
d1<-c("A1","A2","A3")  # row names
d2<-c("B1","B2")       # column names
d3<-c("C1","C2")       # strata names
myarray<-array(data,c(3,2,2),dimnames=list(d1,d2,d3))
dimnames(myarray)  # display dimension names of myarray
```

```
[[1]]
[1] "A1" "A2" "A3"

[[2]]
[1] "B1" "B2"

[[3]]
[1] "C1" "C2"
```

```
myarray
```

```
, , C1

   B1 B2
A1  1  4
A2  2  5
A3  3  6

, , C2

   B1 B2
A1  7 10
A2  8 11
A3  9 12
```

```
data<-1:10
data
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Suppose we want to create an array of dimension `(2,3,2)` from vector `data`. So total $2 \times 3 \times 2 = 12$ elements are required.

The length of data is shorter than the specified dimension. So the values of data are recycled from the beginning again to make it an array of desired dimension.

```
myarray<-array(data,dim=c(2,3,2))
myarray
```

```
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9    1
[2,]    8   10    2
```

```
data<-1:13
data
```

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

If the length of data is larger than the specified dimension, then additional data values are omitted while creating an array.

```
myarray<-array(data,dim=c(2,3,2))
myarray
```

```
, , 1

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

, , 2

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

Creating an array from a vector using `dim`

An array can also be created from a vector by assigning the dimension to a given vector.

```
vec1<-1:12
is.array(vec1)      # check whether vec1 is array
```

```
[1] FALSE
```

```
dim(vec1)<-c(3,2,2) # convert vec1 to array of specified dimension
is.array(vec1)    # check whether vec1 is array
```

```
[1] TRUE
```

```
vec1
```

```
, , 1

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

, , 2

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

Array Indexing

Elements of a array can be accessed by specifying row number(s) and/or column number(s) and/or strata number. Like

- `myarray[i, ,]` returns i^{th} row,
- `myarray[,j,]` returns j^{th} column and
- `myarray[i,j,]` returns elements from i^{th} row and j^{th} column and
- `myarray[i,j,k]` returns $(i, j, k)^{th}$ element of an array.

That is, if any index position is given as empty, then the full range of that subscript is displayed.

```
myarray<-array(1:12,dim=c(3,2,2))
myarray
```

```
, , 1

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

, , 2

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

```
myarray[2,,]  # returns 2nd row
```

```
     [,1] [,2]
[1,]    2    8
[2,]    5   11
```

```
myarray[,2,]  # returns 2nd column
```

```
     [,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12
```

```
myarray[2,2,]  # returns 2nd row and 2nd column
```

```
[1]  5 11
```

```
myarray[,2,2]  # returns 2nd column and 2nd strata
```

```
[1] 10 11 12
```

```
myarray[2,2,1] # returns 2nd row, 2nd column and 1st strata
```

```
[1] 5
```

```
myarray[1:2,2,]
```

```
     [,1] [,2]
[1,]    4   10
[2,]    5   11
```

```
myarray[2,1:2,]
```

```
     [,1] [,2]
[1,]    2    8
[2,]    5   11
```

List

List

Lists in R are a generic vectors where each element can be any type of object; e.g., a vector (of any mode), a matrix, a data frame or a function. Because of this flexibility, lists are the basis for most complex objects in R.

The function to create `list` is

`list(name_1=object_1, name_2=object_2, ...,name_m=object_m)`.

If names are omitted, the component of lists are numbered only.

```
list1<-list(1.34,c("Bob","John"),TRUE,1+2i)
list1
```

```
[[1]]
[1] 1.34

[[2]]
[1] "Bob"  "John"

[[3]]
[1] TRUE

[[4]]
[1] 1+2i
```

```
length(list1)  # display the no. of top level components
```

```
[1] 4
```

```
mode(list1)  # display the mode of list1
```

```
[1] "list"
```

```r
is.list(list1)
```

```
[1] TRUE
```

Every component of a list is a vector. The double bracket `[[1]]` is the first component of the list having one element. The `[[2]]` is the second component of a list having two elements.

```r
list2<-list(value=1.34,names=c("Bob","John"),TRUE,1+2i)
list2
```

```
$value
[1] 1.34

$names
[1] "Bob"  "John"

[[3]]
[1] TRUE

[[4]]
[1] 1+2i
```

Referencing elements of list

The components in a list are given numbers or given names. Hence the components may be referred by its number or name.

The component of a list is referred by

- its number like `list2[[2]]` or
- its name like `list2$name` or
- its name like `list2[["name"]]`

The element of component is referred by

- `list2[[2]][1]` or
- `list2$name[1]`.

```r
list2[[1]]     # display 1st component of list2
```

```
[1] 1.34
```

```r
list2[["value"]]
```

```
[1] 1.34
```

```r
list2$names    # display component whose name is 'names'
```

```
[1] "Bob"  "John"
```

```
list2$names[1] # display 1st element of component whose name is 'names'
```

```
[1] "Bob"
```

```
list2[[2]][1]  # display 1st element from 2nd component of list2
```

```
[1] "Bob"
```

```
list2[2]    # display all elements of 2nd component
```

```
$names
[1] "Bob"  "John"
```

```
list2[[2]]  # display all elements of 2nd component
```

```
[1] "Bob"  "John"
```

Factors

Factors are used to represent categorical data (nominal or ordinal).

- Nominal variables are categorical where order is not important, e.g., Gender ("Male" or "Female") of a respondent is an example of nominal data.
- Ordinal variables are also categorical but the order is important, e.g., Socio economic status (SES) of a respondent ("LES", "MES" or "HES") is an example of ordinal data. Respondent with "LES" is having less earning than respondent with "MES"

The function `factor()` is used to store the variable as factor variable. R internally stores the categorical values as a vector of integers in the range $1, 2, \cdots, k$, where k is the number of unique values in the variable and map these values to the categorical values.

```
gender<-c("Male","Male","Female","Male","Female")
gender<-factor(gender) # Store this as (2,2,1,2,1)
gender
```

```
[1] Male   Male   Female Male   Female
Levels: Female Male
```

```
str(gender) # display the structure of gender
```

```
 Factor w/ 2 levels "Female","Male": 2 2 1 2 1
```

The statement `gender<-factor(gender)` store gender as vector (2,2,1,2,1) and associate it with `1= Female` and `2 = Male`.

```
mode(gender)
```

```
[1] "numeric"
```

```
class(gender)  # display the class of gender
```

```
[1] "factor"
```

```
levels(gender) # display the levels of gender
```

```
[1] "Female" "Male"
```

If the levels are not specified, R assigns the levels to the factor variable alphabetically.

The sequence of levels can be set using the `levels` argument to `factor()` function.

```
gender<-c("Male","Male","Female","Male","Female")
gender<-factor(gender,levels=c("Male","Female"))
gender
```

```
[1] Male   Male   Female Male   Female
Levels: Male Female
```

```
str(gender) # display the structure of gender
```

```
 Factor w/ 2 levels "Male","Female": 1 1 2 1 2
```

```
class(gender)  # display the class of gender
```

```
[1] "factor"
```

```
attributes(gender)  # display attributes of gender
```

```
$levels
[1] "Male"   "Female"

$class
[1] "factor"
```

Sometime the categorical variable is coded as numeric. For example we coded `male` as 0 and `female` as 1 for the data `("M","M","F","M","F")` as `(0,0,1,0,1)`. Then such coded data can also be converted to factor using `factor()` function by specifying the levels.

```
gen<-c(0,0,1,0,1)
fgen<-factor(gen,levels=0:1)
fgen
```

```
[1] 0 0 1 0 1
Levels: 0 1
```

Levels can also be assigned using `level()` function.

```
levels(fgen)<-c("M","F") # set levels
fgen
```

```
[1] M M F M F
Levels: M F
```

```
as.numeric(fgen)
```

```
[1] 1 1 2 1 2
```

The function `as.numeric()` extract the numerical coding as numbers 1 and 2. Note that the original input values are 0 and 1.

Ordered factor

Ordered factor is used when we have a qualitative data but the levels are assumed to be ordered. Like socio economic status of students.

```
ses<-c("MES","LES","HES","MES","LES","LES")
ses.factor<-factor(ses,ordered=TRUE)
ses.factor
```

```
[1] MES LES HES MES LES LES
Levels: HES < LES < MES
```

Since the levels are not specified, R assigns alphabetically.

In order to get proper ordered factors, we have to specify the levels in proper order.

```
ses<-c("MES","LES","HES","MES","LES","LES")
ses.factor<-factor(ses,levels=c("LES","MES","HES"),
                   ordered=TRUE)
ses.factor
```

```
[1] MES LES HES MES LES LES
Levels: LES < MES < HES
```

Some additional functions for factors

- `is.factor(x)` : Check whether `x` is factor
- `as.factor(x)` : Convert `x` to factor
- `is.ordered(x)`: Check whether `x` is ordered factor
- `as.ordered(x)`: Convert `x` to ordered factor

```
is.ordered(ses.factor)
```

```
[1] TRUE
```

```r
is.factor(ses.factor)
```

```
[1] TRUE
```

```r
xnew<-c("Male","Male","Female","Male","Female")
is.factor(xnew) # check whether xnew is factor
```

```
[1] FALSE
```

```r
class(xnew)   # display the class of xnew
```

```
[1] "character"
```

```r
xnew<-as.factor(xnew)  #  convert xnew to factors
xnew
```

```
[1] Male   Male   Female Male   Female
Levels: Female Male
```

```r
is.factor(xnew)
```

```
[1] TRUE
```

```r
is.ordered(xnew)
```

```
[1] FALSE
```

Creating factor from numerical data

Numerical variable can be converted to factor by using `cut()` function. It divides the range of values of variable into intervals and codes the values in the variable according to which interval they fall.

```r
age<-c(10,20,25,12,30,45,50,26,24,13,26,47,48,50)
ageGroup<-cut(age,breaks=c(0,20,30,50))
ageGroup
```

```
 [1] (0,20]  (0,20]  (20,30] (0,20]  (20,30] (30,50] (30,50] (20,30]
 [9] (20,30] (0,20]  (20,30] (30,50] (30,50] (30,50]
Levels: (0,20] (20,30] (30,50]
```

```r
str(ageGroup)
```

```
 Factor w/ 3 levels "(0,20]","(20,30]",..: 1 1 2 1 2 3 3 2 2 1 ...
```

```r
is.factor(ageGroup)
```

```
[1] TRUE
```

```r
is.ordered(ageGroup)
```

```
[1] FALSE
```

```r
attributes(ageGroup)
```

```
$levels
[1] "(0,20]"  "(20,30]" "(30,50]"

$class
[1] "factor"
```

```r
class(ageGroup)
```

```
[1] "factor"
```

```r
table(ageGroup) # Tabulate the variable ageGroup
```

```
ageGroup
 (0,20] (20,30] (30,50]
      4       5       5
```

Generate Factor Levels

The function `gl()` generates regular series of factors.

The general structure of `gl()` is

```
gl(n, k, length = n*k, labels = seq_len(n), ordered = FALSE)
```

- **`n :`** an integer giving the number of levels,
- **`k :`** an integer giving the number of replications,
- **`length :`** an integer giving the length of the result,
- **`labels :`** vector of labels for the resulting factor,
- **`ordered :`** whether the result should be ordred or not.

```r
gl(2,6)
```

```
 [1] 1 1 1 1 1 1 2 2 2 2 2 2
Levels: 1 2
```

It generates two levels 1 and 2, each level six times.

```r
gl(3,4,length=14)
```

```
 [1] 1 1 1 1 2 2 2 2 3 3 3 3 1 1
Levels: 1 2 3
```

```r
gl(2,4,label=c("Smoker","Non-smoker"))
```

```
[1] Smoker     Smoker     Smoker     Smoker     Non-smoker Non-smoker
[7] Non-smoker Non-smoker
Levels: Smoker Non-smoker
```

Generates two levels `Smoker` and `Non-smoker`, each replicated six times.

Data Frame

Data frames are like matrices except that the columns are allowed to be of different types. Data frames can be constructed using the function `data.frame()`. It converts collection of vectors or a matrix into a data frame.

Each row in the data frame corresponds to different observational units and each column in the data frame corresponds to different variables.

Function	Output
`names(dataframe)`	Display the names of the variables of the data frame
`str(dataframe)`	Explore the data structure of a data frame
`dim(dataframe)`	Display the dimension of a data frame
`class(dataframe)`	Display the class of a data frame
`is.data.frame(dataframe)`	Check whether the argument is data frame
`as.data.frame(x)`	Convert argument x to data frame
`attributes(dataframe)`	access attributes of data frame

Creating a data frame

Suppose we have some data about the students as follows:

name	sex	age	weight
A	M	10	26
B	F	20	35
C	F	12	28
D	M	14	30
E	M	16	31
F	F	15	29
G	M	17	34

Creating a data frame directly using `data.frame()` function

```r
student<-data.frame(
  name=c("A","B","C","D","E","F","G"),
  sex=c("M","F","F","M","M","F","M"),
  age=c(10,20,12,14,16,15,17),
  weight=c(26,35,28,30,31,29,34))
str(student) # display structure of a data frame
```

```
'data.frame':   7 obs. of  4 variables:
 $ name  : Factor w/ 7 levels "A","B","C","D",..: 1 2 3 4 5 6 7
 $ sex   : Factor w/ 2 levels "F","M": 2 1 1 2 2 1 2
 $ age   : num  10 20 12 14 16 15 17
 $ weight: num  26 35 28 30 31 29 34
```

```
attributes(student)
```

```
$names
[1] "name"   "sex"    "age"    "weight"

$row.names
[1] 1 2 3 4 5 6 7

$class
[1] "data.frame"
```

```
head(student,3) # display top 3 rows of a data frame
```

```
  name sex age weight
1    A   M  10     26
2    B   F  20     35
3    C   F  12     28
```

Creating a data frame from vectors

Data frame can also be created from vectors. To construct a data frame from the above data, begin by constructing four vectors corresponding to each column of the data.

```
name<-c("A","B","C","D","E","F","G")
sex<-c("M","F","F","M","M","F","M")
age<-c(10,20,12,14,16,15,17)
weight<-c(26,35,28,30,31,29,34)
```

Use a `data.frame()` function to combine all the four vectors into a single data frame entity.

The `data.frame` function creates an object called `student` and within that it stores values of the four variables *name*, *sex*, *age* and *weight*.

```
student<-data.frame(name,sex,age,weight)
class(student) # display class of a data frame
```

```
[1] "data.frame"
```

The function `names()` display the name of each variable in a data frame. The `str()` function display status of each variable in a data frame.

```
names(student)  # display the names of the variable from the data frame
```

```
[1] "name"   "sex"    "age"    "weight"
```

```
str(student)  # Display the structure of a data frame
```

```
'data.frame':   7 obs. of  4 variables:
 $ name  : Factor w/ 7 levels "A","B","C","D",..: 1 2 3 4 5 6 7
 $ sex   : Factor w/ 2 levels "F","M": 2 1 1 2 2 1 2
 $ age   : num  10 20 12 14 16 15 17
 $ weight: num  26 35 28 30 31 29 34
```

Creating a data frame from `list()`

Create a list of students using all the four vectors defined above. Then create a data frame from a list.

```
# Make a list from a vector
student.list<-list(name=name,sex=sex,age=age,weight=weight)
class(student.list)
```

```
[1] "list"
```

```
student<-data.frame(student.list) # make a data frame from list
str(student) # display the structure of a data frame
```

```
'data.frame':   7 obs. of  4 variables:
 $ name  : Factor w/ 7 levels "A","B","C","D",..: 1 2 3 4 5 6 7
 $ sex   : Factor w/ 2 levels "F","M": 2 1 1 2 2 1 2
 $ age   : num  10 20 12 14 16 15 17
 $ weight: num  26 35 28 30 31 29 34
```

```
dim(student) # display dimension of a data frame
```

```
[1] 7 4
```

```
attributes(student)
```

```
$names
[1] "name"   "sex"    "age"    "weight"

$row.names
[1] 1 2 3 4 5 6 7

$class
[1] "data.frame"
```

Operators in R

Arithmetic Operators

The following symbols are used as arithmetic operators in R language:

Operator Symbols	Arithmetic Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
^	Raise to power (exponentiation)
%/%	Integer division
%%	Remainder from integer division

Examples of Arithmetic Operations

```
x<-c(1,2,3,4)
x
```

```
[1] 1 2 3 4
```

```
x+5 # 5 is added to each element of x
```

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

```
x-2 # 2 is subtracted from each element of x
```

```
[1] -1  0  1  2
```

```
x*2 # each element of x is multiplied by 2
```

```
[1] 2 4 6 8
```

```
x/2  #  each element of x is divided by 2
```

```
[1] 0.5 1.0 1.5 2.0
```

```
x%/%2 # integer division on dividing each element by 2
```

```
[1] 0 1 1 2
```

```
x%%2 # Remainder from integer division
```

```
[1] 1 0 1 0
```

```
x<-c(1,2,3,4); y<-c(1,2,3)
x+y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

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

```
x-y
```

```
Warning in x - y: longer object length is not a multiple of shorter object
length
```

```
[1] 0 0 0 3
```

`sum()` and `mean()` are built in function to compute sum and mean respectively.

```
sum(x)
```

```
[1] 10
```

```
sum(x)/length(x)
```

```
[1] 2.5
```

```
sum(x^2)
```

```
[1] 30
```

```
sum((x-mean(x))^2)
```

```
[1] 5
```

Relational Operators

The following symbols are used to represent relational operations in R language:

Operator Symbols	Relational Operation
==	Equal to
! =	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

```
x<-c(12,10,8,16,6)
x<8
```

```
[1] FALSE FALSE FALSE FALSE  TRUE
```

```
x==10
```

```
[1] FALSE  TRUE FALSE FALSE FALSE
```

```
x<=8
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE
```

```
x<-c(12,10,8,16,6)
x[x>8]
```

```
[1] 12 10 16
```

```
x[x!=10]
```

```
[1] 12  8 16  6
```

```
x[x<=8]
```

```
[1] 8 6
```

```
x<-c(12,10,8,16,6)
y<-c(10,12,8,18,6)
x==y
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE
```

```
y-2
```

```
[1]  8 10  6 16  4
```

```
x<=y-2
```

```
[1] FALSE  TRUE FALSE  TRUE FALSE
```

Logical Operators

The following symbols are used to represent logical operators in R language.

Operator Symbols	Relational Operation	Example
!	logical NOT	!x
&	logical AND	x& y
&&	logical AND	x && y
|	logical OR	x | y
||	logical OR	x || y
xor	exclusive OR	xor(x,y)

The shorter form & performs element-wise comparison, whereas the longer form && evaluates left to right examining only the first element in each vector.

```
x<-c(12,10,8,16,6)
x<=8 & x >=5
```

```
[1] FALSE FALSE  TRUE FALSE  TRUE
```

```
x<=8 && x >=5
```

```
[1] FALSE
```

```
x>=8 | x <=6
```

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

```
x>=8 || x <=6
```

```
[1] TRUE
```

```
x<-c(12,10,8,16,6)
y<-c(10,12,8,18,6)
x<=y & x >=10
```

```
[1] FALSE  TRUE FALSE  TRUE FALSE
```

```
x<=y && x >=10
```

```
[1] FALSE
```

```
x>=y | x <=12
```

```
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

```
x>=y || x <=12
```

```
[1] TRUE
```

```
x<-c(12,10,8,16,6)
y<-c(10,12,8,18,6)
x[x<=10 & x >=8]
```

```
[1] 10  8
```

```
x[x!=y & x <y]
```

```
[1] 10 16
```

```
y[x!=y & x <y]
```

```
[1] 12 18
```

Built-in Functions

Built-in Mathematical functions

Function	Operation Performed
`sqrt(x)`	Square root of `x`
`abs(x)`	Absolute value of `x`
`exp(x)`	Exponential value of `x`
`log(x)`	Logarithm value of `x`
`log10(x)`	Logarithm of `x` (base 10)
`ceiling(x)`	Closest integer not less than `x`
`floor(x)`	Closest integer not greater than `x`
`round(x,n)`	Round `x` to `n` significant places
`rev(x)`	reverse the elements of `x`
`sort(x)`	sort the elements of `x` in increasing order
`rank(x)`	assign ranks to the elements of `x`

Built-in Mathematical functions

Function	Operation Performed
`sin(x)`	sine value of `x`
`cos(x)`	cosine value of `x`
`tan(x)`	tangent value of `x`
`asin(x)`	arc-sine value of `x`
`acos(x)`	arc-cosine value of `x`
`atan(x)`	arc-tangent value of `x`
`sinh(x)`	hyperbolic sine of `x`
`cosh(x)`	hyperbolic cosine of `x`
`tanh(x)`	hyperbolic tangent of `x`
`asinh(x)`	hyperbolic arc-sine of `x`
`acosh(x)`	hyperbolic arc-cosine of `x`
`atanh(x)`	hyperbolic arc-tangent of `x`

Built-in Special Functions

Function	Operation Performed
`beta(a,b)`	Beta function
`lbeta(a,b)`	Logarithm of beta function
`gamma(a)`	Gamma function
`lgamma(a)`	Logarithm of gamma function
`choose(n,k)`	Binomial coefficient
`lchoose(n,k)`	Logarithm of Binomial Coefficient

Function	Operation Performed
`factorial(x)`	Factorial of `x` i.e. `x!`
`lfactorial(x)`	Logarithm of Factorial of `x` i.e. `log(x!)`

Built-in Statistical Functions

Function	Operation Performed
`length(x)`	Length of object `x`
`sum(x)`	Sum of elements of `x`
`prod(x)`	Product of elements of `x`
`mean(x)`	Mean of `x`
`weighted.mean(x,w)`	Weighted mean of `x` with weights `w`
`median(x)`	Median of `x`
`sd(x)`	Std. Dev. of `x`
`var(x)`	Varance of `x`
`mad(x)`	Mean absolute deviation of `x`
`IQR(x)`	Inter Quartile Range of `x`

Built-in Statistical Functions

Function	Operation Performed
`max(x)`	Maximum of all the elements of `x`
`min(x)`	Minimum of all the elements of `x`
`range(x)`	Return minimum and maximum of `x`
`quantile(x,probs=)`	Quantiles of `x`
`cov(x,y)`	Covariance between `x` and `y`
`cor(x,y)`	Correlation between `x` and `y`
`fivenum(x)`	Returns five number summary of `x`
`cumsum(x)`	Cumulative sum of elements of `x`
`cumprod(x)`	Cumulative product of elements of `x`

Resources

- http://www.r-project.org
- http://cran.r-project.org/doc/contrib/short-refcard.pdf
- https://cran.r-project.org/doc/manuals/
- http://www.inside-r.org/r-doc
- http://www.rcommander.com/
- http://rseek.org/ R specific Google powered search engine
- http://www.statmethods.net/interface/help.html

Thanks

USE R!

Tell Your Friends!