```
x <- "dataset"
typeof(x)
```

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

```
class(x)
```

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

```
attributes(x)
```

```
## NULL
```

```
y <- 1:10
y
```

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

```
typeof(y)
```

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

```
length(y)
```

```
## [1] 10
```

```
class(y)
```

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

```
z <- as.numeric(y)
z
```

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

```
typeof(z)
```

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

```
class(z)
```

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

```
#typeof() indicates lower data type and class indicates higher data types
```

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

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

```
typeof(x)
```

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

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

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

```
typeof(x)
```

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

```
str(x)
```

```
##  int [1:3] 1 2 3
```

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

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

```
typeof(y)
```

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

```
str(y)
```

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

```
z <- c("Sarah", "Tracy", "Jon")
class(z)
```

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

```
typeof(z)
```

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

```
length(z)
```

```
## [1] 3
```

```
str(z)
```

```
##  chr [1:3] "Sarah" "Tracy" "Jon"
```

```
z <- c(z, "Annette")
z
```

```
## [1] "Sarah"   "Tracy"   "Jon"     "Annette"
```

```
z <- c("Greg", z)
z
```

```
## [1] "Greg"    "Sarah"   "Tracy"   "Jon"     "Annette"
```

```
series <- 1:10
seq(10)
```

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

```
series
```

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

```
seq(from = 1, to = 10, by = 0.1)
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
## [15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
## [29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
## [43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
## [57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
## [71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
## [85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

```
#Missing Data
#R supports missing data in vectors. They are represented as NA (Not Available) and can be used
 for all the vector types covered in this lesson:

x <- c(0.5, NA, 0.7)
x <- c(TRUE, FALSE, NA)
x <- c("a", NA, "c", "d", "e")
x <- c(1+5i, 2-3i, NA)
```

```
#The function is.na() indicates the elements of the vectors that represent missing data, and the
 function anyNA() returns TRUE if the vector contains any missing values:

x <- c("a", NA, "c", "d", NA)
y <- c("a", "b", "c", "d", "e")
is.na(x)
```

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

```
anyNA(x)
```

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

```
anyNA(y)
```

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

```
class(is.na(x))
```

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

```
typeof(is.na(x))
```

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

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

```
class(xx)
```

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

```
typeof(xx)
```

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

```
class(yy)
```

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

```
typeof(yy)
```

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

```
class(zz)
```

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

```
typeof(zz)
```

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

```
#You can also control how vectors are coerced explicitly using the as.<class_name>() functions:

p<-as.numeric("1")
q<-as.character(1:2)
class(p)
```

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

```
typeof(p)
```

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

```
class(q)
```

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

```
class(q)
```

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

```
length(1:10)
```

```
## [1] 10
```

```
nchar("Software Carpentry")
```

```
## [1] 18
```

```
#In R matrices are an extension of the numeric or character vectors.
m <- matrix(nrow = 2, ncol = 2)
m
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

```
dim(m)
```

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

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

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

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

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x, y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

```
#You can also use the byrow argument to specify how the matrix is filled. From R's own documenta
tion:

mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE)
mdat
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13
```

```
#n R lists act as containers. Unlike atomic vectors, the contents of a list are not restricted t
o a single mode and can encompass any mixture of data types. Lists are sometimes called generic
 vectors, because the elements of a list can by of any type of R object, even lists containing f
urther lists. This property makes them fundamentally different from atomic vectors.

#A list is a special type of vector. Each element can be a different type.

#Create lists using list() or coerce other objects using as.list(). An empty list of the require
d length can be created using vector()


x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

```
class(x)
```

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

```
typeof(x)
```

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

```
x <- vector("list", length = 5) ## empty list
length(x)
```

```
## [1] 5
```

```
x
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
```

```
#The content of elements of a list can be retrieved by using double square brackets.

x[[1]]
```

```
## NULL
```

```
#Vectors can be coerced to lists as follows:
x <- 1:3
x <- as.list(x)
length(x)
```

```
## [1] 3
```

```
x
```

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

```
class(x)
```

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

```
print(x[1])
```

```
## [[1]]
## [1] 1
```

```
class(x[1])
```

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

```
x[[1]]
```

```
## [1] 1
```

```
class(x[[1]])
```

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

```
xlist <- list(a = "Karthik Ram", b = 1:10, data = head(iris))
xlist
```

```
## $a
## [1] "Karthik Ram"
##
## $b
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $data
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
names(xlist)
```

```
## [1] "a"    "b"    "data"
```

```
length(xlist)
```

```
## [1] 3
```

```
#IMPPPPPPPPPPPPPPPPPPPPPPPPP:Lists can be extremely useful inside functions. Because the functio
ns in R are able to return only a single object, you can "staple" together lots of different kin
ds of results into a single object that a function can return.
```

```
xlist$a
```

```
## [1] "Karthik Ram"
```

```
xlist$data
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)
dat
```

```
##    id  x  y
## 1   a  1 11
## 2   b  2 12
## 3   c  3 13
## 4   d  4 14
## 5   e  5 15
## 6   f  6 16
## 7   g  7 17
## 8   h  8 18
## 9   i  9 19
## 10  j 10 20
```

```
# head() - shows first 6 rows
# tail() - shows last 6 rows
# dim() - returns the dimensions of data frame (i.e. number of rows and number of columns)
# nrow() - number of rows
# ncol() - number of columns
# str() - structure of data frame - name, type and preview of data in each column
# names() - shows the names attribute for a data frame, which gives the column names.
# sapply(dataframe, class) - shows the class of each column in the data frame
```

```
dim(dat)
```

```
## [1] 10  3
```

```
print("######")
```

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

```
names(dat)
```

```
## [1] "id" "x"  "y"
```

```
print("######")
```

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

```
str(dat)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ id: Factor w/ 10 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10
##  $ x : int  1 2 3 4 5 6 7 8 9 10
##  $ y : int  11 12 13 14 15 16 17 18 19 20
```

```
print("######")
```

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

```
sapply(dat,class)
```

```
##        id         x         y
##  "factor" "integer" "integer"
```

```
class(dat$id)
```

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

```
typeof(dat$id)
```

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

```
is.list(dat)
```

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

```
#See that it is actually a special list:
class(dat)
```

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

```
dat[["y"]]
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
dat$y
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
#R's basic data types are character, numeric, integer, complex, and logical.

#R's basic data structures include the vector, list, matrix, data frame, and factors.

#Objects may have attributes, such as name, dimension, and class.
```

```
blood<-c("A","B","A","AB","O")
class(blood)
```

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

```
typeof(blood)
```

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

```
bloody<-c("A","B","A","AB","O")
factor_bloody <-factor(bloody)
factor_bloody
```

```
## [1] A  B  A  AB O
## Levels: A AB B O
```

```
class(factor_bloody)#factor
```

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

```
typeof(factor_bloody)#int
```

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

```
class(bloody)#char
```

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

```
str(bloody)
```

```
##  chr [1:5] "A" "B" "A" "AB" "O"
```

```
str(factor_bloody)
```

```
##  Factor w/ 4 levels "A","AB","B","O": 1 3 1 2 4
```

```
#####IMPPPPPPPPPPPPPP: BELOW indicates Factor w/ 4 levels "A","AB","B","O": 1 3 1 2 4 which mean
s 4 levels are there in factor_bloody column and index of them are : A-1 AB-2 B-3 O-4 .
#Now characters in factor_bloody is A B A AB O which are 1 3 1 2 4 as per assigned in statement
 """""Factor w/ 4 levels "A","AB","B","O": 1 3 1 2 4""""" which means first is 1(A), second is 3
(B), third is 1(A), fourth is 2(AB) and last is 4(O).
```

```
head(warpbreaks)
```

```
##   breaks wool tension
## 1     26    A       L
## 2     30    A       L
## 3     54    A       L
## 4     25    A       L
## 5     70    A       L
## 6     52    A       L
```

```
str(warpbreaks)
```

```
## 'data.frame':    54 obs. of  3 variables:
##  $ breaks : num  26 30 54 25 70 52 51 26 67 18 ...
##  $ wool   : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
##  $ tension: Factor w/ 3 levels "L","M","H": 1 1 1 1 1 1 1 1 1 2 ...
```

```
class(warpbreaks$tension)
```

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

```
typeof(warpbreaks$tension)
```

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

```
class(warpbreaks$wool)
```

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

```
typeof(warpbreaks$wool)
```

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

```
factor_bloody2<-factor(blood,levels=c("O","B","A","AB"))
factor_bloody2
```

```
## [1] A  B  A  AB O
## Levels: O B A AB
```

```
str(factor_bloody2)
```

```
##  Factor w/ 4 levels "O","B","A","AB": 3 2 3 4 1
```

```
levels(factor_bloody)
```

```
## [1] "A"  "AB" "B"  "O"
```

```
levels(factor_bloody)<-c("AB_GROUP","A_GROUP","B_GROUP","O_GROUP")
factor_bloody
```

```
## [1] AB_GROUP B_GROUP  AB_GROUP A_GROUP  O_GROUP
## Levels: AB_GROUP A_GROUP B_GROUP O_GROUP
```

```
size<-c("L","M","M","S","M","L")
size
```

```
## [1] "L" "M" "M" "S" "M" "L"
```

```
factor_size<-factor(size,ordered = TRUE,levels=c("S","M","L"))
factor_size
```

```
## [1] L M M S M L
## Levels: S < M < L
```

```
#ordered=TRUE led us to this
```

```
factor_size[1]
```

```
## [1] L
## Levels: S < M < L
```

```
factor_size[2]
```

```
## [1] M
## Levels: S < M < L
```

```
factor_size[1] > factor_size[2]
```

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

```
#L>M
```

```
x <- c(1,3,8,25,100);
x
```

```
## [1]   1   3   8  25 100
```

```
class(x)
```

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

```
typeof(x)
```

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

```
print("$$$$$$$$$$")
```

```
## [1] "$$$$$$$$$$"
```

```
seq(along = x)
```

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

```
seq
```

```
## function (...)
## UseMethod("seq")
## <bytecode: 0x000000001d2d39c0>
## <environment: namespace:base>
```

```
class(seq)
```

```
## [1] "function"
```

```
typeof(seq)
```

```
## [1] "closure"
```

```
class(seq(along = x))
```

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

```
typeof(seq(along = x))
```

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

```
class(c("A","B","C","D"))
```

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

```
typeof(c("A","B","C","D"))
```

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

```
print("$$$$$$$$$$$$$$$$$$$$$$")
```

```
## [1] "$$$$$$$$$$$$$$$$$$$$$$"
```

```
class(c(11.11,10.10))
```

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

```
typeof(c(11.11,10.10))
```

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

```
#typeof(c(A,B,C,D))--Error in typeof(c(A, B, C, D)) : object 'A' not found
#typeof(c(A,B,C,"D"))--Error in typeof(c(A, B, C, "D")) : object 'A' not found
print("$$$$$$$$$$$$$$$$$$$$$$")
```

```
## [1] "$$$$$$$$$$$$$$$$$$$$$$"
```

```
class(c("A","B",3,4))
```

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

```
typeof(c("A","B",3,4))
```

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

```
class(c(1,2,3,4))
```

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

```
typeof(c(1,2,3,4))
```

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

```
print("$$$$$$$$$$$$$$$$$$$$$$")
```

```
## [1] "$$$$$$$$$$$$$$$$$$$$$$"
```

```
class(c("A","B",3.5,4.5))
```

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

```
typeof(c("A","B",3.5,4.5))
```

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

```
print("$$$$$$$$$$$$$$$$$$$$$$")
```

```
## [1] "$$$$$$$$$$$$$$$$$$$$$$"
```

```
class(c("A","B",3.5,4.5))
```

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

```
class(c(3.5,4.5,"A","B"))
```

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

```
typeof(c(3.5,4.5,"A","B"))
```

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

```
#How to relevel factors
# Create a factor with the wrong order of levels
sizes <- factor(c("small", "large", "large", "small", "medium"))
sizes
```

```
## [1] small  large  large  small  medium
## Levels: large medium small
```

```
# Make medium first
sizes <- relevel(sizes, "medium")
sizes
```

```
## [1] small  large  large  small  medium
## Levels: medium large small
```

```
# Make small first
sizes <- relevel(sizes, "small")
sizes
```

```
## [1] small  large  large  small  medium
## Levels: small medium large
```

```
sizes
```

```
## [1] small  large  large  small  medium
## Levels: small medium large
```

```
#changes reflected if we are asssigning like this : sizes <- relevel(sizes, "small") and if we d
o only relevel(sizes, "small"), it wont reflect in factor sizes.
```

```
#You can also specify the proper order when the factor is created.

sizes1 <- factor(c("small", "large", "large", "small", "medium"),
                levels = c("small", "medium", "large"))
sizes1
```

```
## [1] small  large  large  small  medium
## Levels: small medium large
```

```
class(sizes1)
```

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

```
#sizes <- ordered(c("small", "large", "large", "small", "medium"))
sizes <- ordered(sizes, levels = c("small", "medium", "large"))
sizes
```

```
## [1] small  large  large  small  medium
## Levels: small < medium < large
```

```
class(sizes)
```

```
## [1] "ordered" "factor"
```

```
require(reshape2)
```

```
## Loading required package: reshape2
```

```
x = data.frame(subject = c("John", "Mary"),
               time = c(1,1),
               age = c(33,NA),
               weight = c(90, NA),
               height = c(2,2))
x
```

```
##   subject time age weight height
## 1    John    1  33     90      2
## 2    Mary    1  NA     NA      2
```

```
molten = melt(x, id = c("subject", "time"))
molten
```

```
##   subject time variable value
## 1    John    1      age    33
## 2    Mary    1      age    NA
## 3    John    1   weight    90
## 4    Mary    1   weight    NA
## 5    John    1   height     2
## 6    Mary    1   height     2
```

```
#IMPPPPPPPPPPP: All measured variables(age weight and height) must be of the same type, e.g., nu
meric, factor, date. This is required because molten data is stored in a R data frame, and the v
alue column can assume only one type.
#id variables: subject and time
#measured variable : age weight height
```

```
molten = melt(x, id = c("subject", "time"), na.rm = TRUE)
molten
```

```
##   subject time variable value
## 1    John    1      age    33
## 3    John    1   weight    90
## 5    John    1   height     2
## 6    Mary    1   height     2
```

```
#Reshaping your data

#Now that you have a molten data you can reshape it into a data frame using dcast function or in
to a vector/matrix/array using the acast function. The basic arguments of *cast is the molten da
ta and a formula of the form x1 + x2 ~ y1 + y2. The order of the variables matter, the first var
ies slowest, and the last fastest. There are a couple of special variables: "..." represents all
 other variables not used in the formula and "." represents no variable, so you can do formula =
 x1 ~ .

dcast(molten, formula = time + subject ~ variable)
```

```
##   time subject age weight height
## 1    1    John  33     90      2
## 2    1    Mary  NA     NA      2
```

```
dcast(molten, formula = subject + time  ~ variable)
```

```
##   subject time age weight height
## 1    John    1  33     90      2
## 2    Mary    1  NA     NA      2
```

```
dcast(molten, formula = subject  ~ variable)
```

```
##   subject age weight height
## 1    John  33     90      2
## 2    Mary  NA     NA      2
```

```
dcast(molten, formula = ...  ~ variable)
```

```
##   subject time age weight height
## 1    John    1  33     90      2
## 2    Mary    1  NA     NA      2
```

```
acast(molten, formula = subject  ~ time ~ variable)
```

```
## , , age
##
##       1
## John 33
## Mary NA
##
## , , weight
##
##       1
## John 90
## Mary NA
##
## , , height
##
##      1
## John 2
## Mary 2
```

```
#subject and time vs age
#subject and time vs height
#subject and time vs weight

class(acast(molten, formula = subject  ~ time ~ variable))#Array
```

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

```
typeof(acast(molten, formula = subject  ~ time ~ variable))#double
```

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

```
#acast does convert into array matrix vector---> mostly arrays
```

```
#gl() function to generate factors by specifying patterns in their levels
gl(3,2,labels = c("green","red","yellow"))
```

```
## [1] green  green  red    red    yellow yellow
## Levels: green red yellow
```

```
class(gl(3,2,labels = c("green","red","yellow")))
```

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

```
typeof(gl(3,2,labels = c("green","red","yellow")))
```

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

```
#gl(n, k, length = n*k, labels = 1:n, ordered = FALSE)

# n: number of levels
# k: number of replications
# length: length of the result
# labels: labels for the resulting factor levels
gl(4,2,labels = c("green","red","yellow","blue"))
```

```
## [1] green  green  red    red    yellow yellow blue   blue
## Levels: green red yellow blue
```

```
medical.example <-
    data.frame(patient = 1:100,
               age = rnorm(100, mean = 60, sd = 12),
               treatment = gl(2, 50,
                 labels = c("Treatment", "Control")))
medical.example
```

```
##    patient      age treatment
## 1        1 38.17060 Treatment
## 2        2 47.08745 Treatment
## 3        3 60.84205 Treatment
## 4        4 52.47247 Treatment
## 5        5 54.50716 Treatment
## 6        6 75.90560 Treatment
## 7        7 54.83071 Treatment
## 8        8 40.58231 Treatment
## 9        9 45.38774 Treatment
## 10      10 57.47152 Treatment
## 11      11 38.29298 Treatment
## 12      12 43.59190 Treatment
## 13      13 46.24405 Treatment
## 14      14 53.96694 Treatment
## 15      15 62.57091 Treatment
## 16      16 46.05824 Treatment
## 17      17 56.48826 Treatment
## 18      18 69.57072 Treatment
## 19      19 72.76236 Treatment
## 20      20 64.16870 Treatment
## 21      21 70.34503 Treatment
## 22      22 51.24919 Treatment
## 23      23 68.24804 Treatment
## 24      24 69.53587 Treatment
## 25      25 40.31416 Treatment
## 26      26 72.03438 Treatment
## 27      27 36.08880 Treatment
## 28      28 76.30155 Treatment
## 29      29 67.72121 Treatment
## 30      30 62.46761 Treatment
## 31      31 64.61954 Treatment
## 32      32 39.70348 Treatment
## 33      33 64.19246 Treatment
## 34      34 78.03270 Treatment
## 35      35 67.93100 Treatment
## 36      36 59.53587 Treatment
## 37      37 44.46924 Treatment
## 38      38 73.29812 Treatment
## 39      39 66.21558 Treatment
## 40      40 59.32157 Treatment
## 41      41 68.77497 Treatment
## 42      42 57.53266 Treatment
## 43      43 50.67252 Treatment
## 44      44 52.63397 Treatment
## 45      45 68.05077 Treatment
## 46      46 63.32079 Treatment
## 47      47 65.53524 Treatment
## 48      48 46.13509 Treatment
## 49      49 53.49929 Treatment
## 50      50 67.97070 Treatment
## 51      51 54.09810   Control
## 52      52 52.31448   Control
```

```
## 53        53 77.02692   Control
## 54        54 67.39554   Control
## 55        55 65.18127   Control
## 56        56 55.54913   Control
## 57        57 54.41399   Control
## 58        58 74.46173   Control
## 59        59 49.33095   Control
## 60        60 47.98664   Control
## 61        61 61.06392   Control
## 62        62 52.47439   Control
## 63        63 53.61553   Control
## 64        64 73.89189   Control
## 65        65 60.13499   Control
## 66        66 72.67313   Control
## 67        67 64.94175   Control
## 68        68 59.87007   Control
## 69        69 53.82436   Control
## 70        70 54.80887   Control
## 71        71 59.13669   Control
## 72        72 68.89924   Control
## 73        73 65.77614   Control
## 74        74 63.98182   Control
## 75        75 57.43357   Control
## 76        76 84.61740   Control
## 77        77 69.04503   Control
## 78        78 30.93931   Control
## 79        79 58.11266   Control
## 80        80 56.11014   Control
## 81        81 58.03228   Control
## 82        82 63.80165   Control
## 83        83 48.69633   Control
## 84        84 52.79639   Control
## 85        85 64.07860   Control
## 86        86 48.67977   Control
## 87        87 61.90011   Control
## 88        88 63.63306   Control
## 89        89 63.14846   Control
## 90        90 58.10997   Control
## 91        91 48.98152   Control
## 92        92 48.32564   Control
## 93        93 62.47733   Control
## 94        94 55.91915   Control
## 95        95 58.78441   Control
## 96        96 63.38931   Control
## 97        97 63.86050   Control
## 98        98 77.98950   Control
## 99        99 63.60090   Control
## 100     100 86.34583   Control
```

```
summary(medical.example)
```

```
##     patient             age           treatment
##  Min.   :  1.00   Min.   :30.94   Treatment:50
##  1st Qu.: 25.75   1st Qu.:52.59   Control  :50
##  Median : 50.50   Median :59.70
##  Mean   : 50.50   Mean   :59.38
##  3rd Qu.: 75.25   3rd Qu.:66.51
##  Max.   :100.00   Max.   :86.35
```

```
#The tapply function is useful when we need to break up a vector into groups(age into treatment
 and control groups) defined by some classifying factor, compute a function on the subsets(mea
n), and return the results in a convenient form(table returned).
## Medical Example
tapply(medical.example$age, medical.example$treatment, mean)
```

```
## Treatment   Control
##  58.13448  60.63321
```

```
#runif() function :
#runif(n=no of observations,min=abc,max=cvb)
u <- runif(20,min = 1,max=5)
u
```

```
##  [1] 2.885596 2.301303 3.327228 3.029521 4.757065 3.140628 3.266154
##  [8] 1.712399 1.206636 1.386725 1.606180 3.970129 1.557350 4.951380
## [15] 4.432663 4.215958 2.067910 4.486171 1.217654 2.317339
```

```
#in gl() function :
#n= no of levels =5
#k= no of replications=5
#labels=c("","","","","")--> total 5

baseball.example <-
    data.frame(team = gl(5, 5,
                 labels = paste("Team", LETTERS[1:5])),
               player = sample(letters, 25),
               batting.average = runif(25, .200, .400))
baseball.example
```

```
##      team player batting.average
## 1  Team A      c       0.2119914
## 2  Team A      e       0.3947608
## 3  Team A      g       0.3131166
## 4  Team A      v       0.3191434
## 5  Team A      n       0.2844533
## 6  Team B      s       0.3351842
## 7  Team B      o       0.3415224
## 8  Team B      q       0.3538669
## 9  Team B      y       0.3448908
## 10 Team B      b       0.2173657
## 11 Team C      u       0.2202075
## 12 Team C      l       0.3402717
## 13 Team C      x       0.3221189
## 14 Team C      t       0.3810328
## 15 Team C      i       0.3249527
## 16 Team D      z       0.3945531
## 17 Team D      w       0.3188343
## 18 Team D      p       0.3830486
## 19 Team D      d       0.3125409
## 20 Team D      m       0.2816642
## 21 Team E      f       0.2330552
## 22 Team E      r       0.2459221
## 23 Team E      a       0.3967481
## 24 Team E      k       0.3076886
## 25 Team E      j       0.3062302
```

```
summary(baseball.example)
```

```
##      team       player   batting.average
##  Team A:5   a      : 1   Min.   :0.2120
##  Team B:5   b      : 1   1st Qu.:0.2845
##  Team C:5   c      : 1   Median :0.3191
##  Team D:5   d      : 1   Mean   :0.3154
##  Team E:5   e      : 1   3rd Qu.:0.3449
##             f      : 1   Max.   :0.3967
##             (Other):19
```

```
baseball.example$team
```

```
##  [1] Team A Team A Team A Team A Team A Team B Team B Team B Team B Team B
## [11] Team C Team C Team C Team C Team C Team D Team D Team D Team D Team D
## [21] Team E Team E Team E Team E Team E
## Levels: Team A Team B Team C Team D Team E
```

```
#to find minimum from a column
min(baseball.example$batting.average)
```

```
## [1] 0.2119914
```

```
max(baseball.example$batting.average)
```

```
## [1] 0.3967481
```

```
## Baseball Example
#tapply(the variable you want to summarize on which you would apply function,the grouping variab
le which would be in column , the function to be applied)
tapply(baseball.example$batting.average, baseball.example$team,
         max)
```

```
##    Team A    Team B    Team C    Team D    Team E
## 0.3947608 0.3538669 0.3810328 0.3945531 0.3967481
```

```
baseball.example[1:5,]
```

```
##     team player batting.average
## 1 Team A      c       0.2119914
## 2 Team A      e       0.3947608
## 3 Team A      g       0.3131166
## 4 Team A      v       0.3191434
## 5 Team A      n       0.2844533
```

```
#you can convert tapply function result into a data frame

class(tapply(baseball.example$batting.average, baseball.example$team,
         max))
```

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

```
typeof(tapply(baseball.example$batting.average, baseball.example$team,
         max))
```

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

```
dftapply<-data.frame(tapply(baseball.example$batting.average, baseball.example$team,
         max))
```

```
dftapply
```

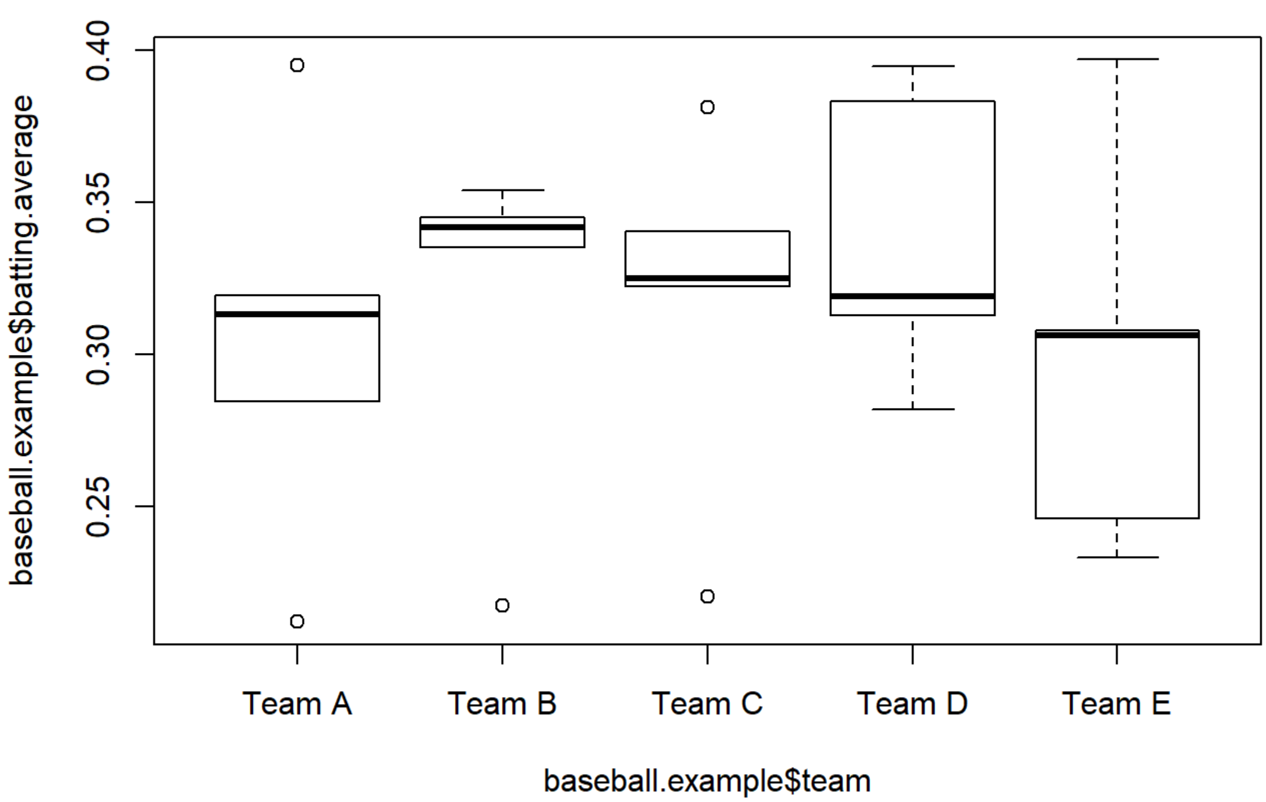
```
##        tapply.baseball.example.batting.average..baseball.example.team..
## Team A                                                        0.3947608
## Team B                                                        0.3538669
## Team C                                                        0.3810328
## Team D                                                        0.3945531
## Team E                                                        0.3967481
```

```
#TEAM A TEAM B and all are index and the 1st column is at the right end
```

```
#you need to give column name to that dataframe to build logically and also to view dataframe fu
lly
colnames(dftapply)<-"maximum value"
dftapply
```

```
##        maximum value
## Team A     0.3947608
## Team B     0.3538669
## Team C     0.3810328
## Team D     0.3945531
## Team E     0.3967481
```

```
plot(baseball.example$batting.average~baseball.example$team)
```



```
paste("A", 1, "%")       #A bunch of individual character strings.
```

```
## [1] "A 1 %"
```

```
paste(1:4, letters[1:4]) #2 or more strings pasted element for element.
```

```
## [1] "1 a" "2 b" "3 c" "4 d"
```

```
paste(1:10)
```

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

```
class(paste(1:10))
```

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

```
typeof(paste(1:10))
```

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

```
class(paste(1:4, letters[1:4]))
```

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

```
typeof(paste(1:4, letters[1:4]))
```

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

```
class(paste("A", 1, "%") )
```

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

```
typeof(paste("A", 1, "%") )
```

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

```
letters[1:4]
```

```
## [1] "a" "b" "c" "d"
```

```
paste("TEAM",LETTERS[1:4])
```

```
## [1] "TEAM A" "TEAM B" "TEAM C" "TEAM D"
```

```
paste("TEAM",letters[1:4])
```

```
## [1] "TEAM a" "TEAM b" "TEAM c" "TEAM d"
```

```
paste("TEAM",letters[1:4],sep="-")
```

```
## [1] "TEAM-a" "TEAM-b" "TEAM-c" "TEAM-d"
```

```
paste0("TEAM",LETTERS[1:4])
```

```
## [1] "TEAMA" "TEAMB" "TEAMC" "TEAMD"
```

```
paste("TEAM",LETTERS[1:4],sep="")
```

```
## [1] "TEAMA" "TEAMB" "TEAMC" "TEAMD"
```

```
paste("TEAM",LETTERS[1:4])
```

```
## [1] "TEAM A" "TEAM B" "TEAM C" "TEAM D"
```

```
person <-"Grover"
action <-"flying"
message(paste0("On ", Sys.Date(), " I realized ", person, " was...\n", action, " by the street"
))
```

```
## On 2018-04-23 I realized Grover was...
## flying by the street
```

```
message(paste("On ", Sys.Date(), " I realized ", person, " was...\n", action, " by the street"))
```

```
## On  2018-04-23  I realized  Grover  was...
##  flying  by the street
```

```
#But we can use sprintf to make one string (less commas + less quotations marks = less errors) a
nd feed the elements that may differ from user to user or time to time.
person <-"Grover"
action <-"flying"
message(sprintf("On %s I realized %s was...\n%s by the street", Sys.Date(), person, action))
```

```
## On 2018-04-23 I realized Grover was...
## flying by the street
```

```
#Working with List in R
# create three different classes of objects
vec <- 1:4
df <- data.frame(y = c(1:3), x = c("m", "m", "f"))
char <- "Hello!"
# add all three objects to one list using list() function
list1 <- list(vec, df, char)
```

```
list1
```

```
## [[1]]
## [1] 1 2 3 4
##
## [[2]]
##   y x
## 1 1 m
## 2 2 m
## 3 3 f
##
## [[3]]
## [1] "Hello!"
```

```
class(vec)
```

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

```
class(char)
```

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

```
class(list1)
```

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

```
typeof(list)
```

```
## [1] "builtin"
```

```
names(list1)
```

```
## NULL
```

```
# coerce vector into a list
as.list(vec)
```

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

```
# name the components of the list
names(list1) <- c("Numbers", "Some.data", "Letters")
list1
```

```
## $Numbers
## [1] 1 2 3 4
##
## $Some.data
##   y x
## 1 1 m
## 2 2 m
## 3 3 f
##
## $Letters
## [1] "Hello!"
```

```
# could have named them when we created list
another.list <- list(Numbers = vec, Letters = char)
another.list
```

```
## $Numbers
## [1] 1 2 3 4
##
## $Letters
## [1] "Hello!"
```

```
names(another.list)
```

```
## [1] "Numbers" "Letters"
```

```
#Extract components from a list: the first is using the [[ ]] operator . Note that we can use the single [ ] operator on a list, but it will return a list rather than the data structure that is the component of the list, which is normally not what we would want to do.
list1[[3]]
```

```
## [1] "Hello!"
```

```
class(list1[[3]])
```

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

```
typeof(list1[[3]])
```

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

```
list1[3]
```

```
## $Letters
## [1] "Hello!"
```

```
class(list1[3])
```

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

```
typeof(list1[3])
```

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

```
# extract 3rd component using $
list1$Letters
```

```
## [1] "Hello!"
```

```
# extract 3rd component using [[ ]] and the name of the component
list1[["Letters"]]
```

```
## [1] "Hello!"
```

```
# subset the first and third components
list1[c(1, 3)]
```

```
## $Numbers
## [1] 1 2 3 4
##
## $Letters
## [1] "Hello!"
```

```
#We can also add a new component to the list or replace a component using the $ or [[ ]] operato
rs. This time I'll add a linear model to the list (remember we can put anything into a list).
# add new component to existing list using $
list1$newthing <- lm(y ~ x, data = df)

# add a new component to existing list using [[ ]]
list1[[5]] <- "new component"
```

```
#Finally, we can delete a component of a list by setting it equal to NULL.
list1$Letters <- NULL
list1
```

```
## $Numbers
## [1] 1 2 3 4
##
## $Some.data
##   y x
## 1 1 m
## 2 2 m
## 3 3 f
##
## $newthing
##
## Call:
## lm(formula = y ~ x, data = df)
##
## Coefficients:
## (Intercept)           xm
##         3.0         -1.5
##
##
## [[4]]
## [1] "new component"
```

```
names(list1)
```

```
## [1] "Numbers"   "Some.data" "newthing"  ""
```

```
#Letters is deleted
```

```
# extract first row of dataframe that is in a list
list1[[2]][1, ]
```

```
##   y x
## 1 1 m
```

```
# describe class of the whole list
class(list1)
```

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

```
# describe the class of the first component of the list
class(list1[[1]])
```

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

```
# take out the model from list and then show summary of what's in the list
list1$newthing <- NULL
str(list1)
```

```
## List of 3
##  $ Numbers  : int [1:4] 1 2 3 4
##  $ Some.data:'data.frame':   3 obs. of  2 variables:
##   ..$ y: int [1:3] 1 2 3
##   ..$ x: Factor w/ 2 levels "f","m": 2 2 1
##  $          : chr "new component"
```

```
names(list1)
```

```
## [1] "Numbers"   "Some.data" ""
```

```
list1
```

```
## $Numbers
## [1] 1 2 3 4
##
## $Some.data
##   y x
## 1 1 m
## 2 2 m
## 3 3 f
##
## [[3]]
## [1] "new component"
```

```
# construct new list of two components
new.list <- list(vec, char)

# notice that it has two components
length(new.list)
```

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

```
# append a component to the end and print
new.list[[length(new.list) + 1]] <- "Appended"

new.list
```

```
## [[1]]
## [1] 1 2 3 4
##
## [[2]]
## [1] "Hello!"
##
## [[3]]
## [1] "Appended"
```

```
names(new.list)
```

```
## NULL
```

```
# initialize list to have 3 null components and print
list2 <- vector("list", 3)
list2
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

```
class(list2)
```

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

```
typeof(list2)
```

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

```
# convert to one long string - use unlist
unlist(list1)
```

```
##        Numbers1        Numbers2        Numbers3        Numbers4
##             "1"             "2"             "3"             "4"
##    Some.data.y1    Some.data.y2    Some.data.y3    Some.data.x1
##             "1"             "2"             "3"             "2"
##    Some.data.x2    Some.data.x3
##             "2"             "1" "new component"
```

```
class(unlist(list1))
```

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

```
#create list of matrices and print
mat.list <- list(mat1=matrix(c(1,2,3,4), nrow=2), mat2=matrix(c(5,6,7,8), nrow=2))
mat.list
```

```
## $mat1
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $mat2
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

```
#convert list to data frame
#1. use ldply
require(plyr)
```

```
## Loading required package: plyr
```

```
ldply(mat.list, data.frame)
```

```
##    .id X1 X2
## 1 mat1  1  3
## 2 mat1  2  4
## 3 mat2  5  7
## 4 mat2  6  8
```

```
#ldply() for list to dataframe using plyr
```

```
#2. use rbind
do.call(rbind.data.frame, mat.list)
```

```
##        V1 V2
## mat1.1  1  3
## mat1.2  2  4
## mat2.1  5  7
## mat2.2  6  8
```

```
rbind.data.frame(mat.list)
```

```
##   mat1 mat2
## 1    1    5
## 2    2    6
## 3    3    7
## 4    4    8
```

```
#apply(x,index,function)
mat<-matrix(1:9,3,3)
mat
```

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

```
#Applying a function to the rows (index=1) or columns (index=2) of a matrix.
apply(mat,1,sum)#1rowsum,2rowsum,3rowsum
```

```
## [1] 12 15 18
```

```
apply(mat,2,sum)#1colsum,2colsum,3colsum
```

```
## [1]  6 15 24
```

```
#lapply(x,function)
#apply a function to each element of the list x
x<-list(1:10)
x
```

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

```
lapply(x,sqrt)
```

```
## [[1]]
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
#lapply always returns a list
```

```
lapply(mat,sqrt)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
##
## [[7]]
## [1] 2.645751
##
## [[8]]
## [1] 2.828427
##
## [[9]]
## [1] 3
```

```
#if we give matrix to lapply, it will convert it into list and return
```

```
class(lapply(x,sqrt))
```

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

```
class(lapply(mat,sqrt))
```

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

```
#sapply(x,function)
#apply a function to each element of the list x with simplification of result
x<-list(1:10)
sapply(x,sqrt)
```

```
##           [,1]
##  [1,] 1.000000
##  [2,] 1.414214
##  [3,] 1.732051
##  [4,] 2.000000
##  [5,] 2.236068
##  [6,] 2.449490
##  [7,] 2.645751
##  [8,] 2.828427
##  [9,] 3.000000
## [10,] 3.162278
```

```
#does not change x here
```

```
x
```

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

```
#sapply returns matrix if list is provided
class(sapply(x,sqrt))
```

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

```
sapply(mat,sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
## [9] 3.000000
```

```
class(sapply(mat,sqrt))
```

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

```
typeof(sapply(mat,sqrt))
```

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

```
#tapply(x,y,function)
#Apply a function to subsets of a vector X and defined the subset by vector Y.
x<-1:10
y <-rep(c(T,F),5)
x
```

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

```
y
```

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

```
tapply(x, y, sum)
```

```
## FALSE  TRUE
##    30    25
```

```
# x   y
# 1   t
# 2   f
# 3   t
# 4   f
# 5   t
# 6   f
# 7   t
# 8   f
# 9   t
# 10  f
#treat x and y as columns of a dataframe and like this, tapply(x,y,sum) x is value on which func
tion is to be applied
#And y is the name of column or the variable that will be tables column name ie TRUE FALSE here
```

```
tapply(x, y, list)
```

```
## $`FALSE`
## [1]  2  4  6  8 10
##
## $`TRUE`
## [1] 1 3 5 7 9
```

```
#convert to list by function = list... previous function = sum
```

```
tapply(x, y, max)
```

```
## FALSE  TRUE
##    10     9
```

```
class(tapply(x, y, list))
```

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

```
class(tapply(x, y, max))
```

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

```
#Intro to apply-based functions for lists
#FUNCTION   INPUT             OUTPUT
#apply    matrix            vector or matrix
#sapply   vector or list    vector or matrix
#lapply   vector or list        list

#lapply always returns list
#
```

```
N <- 10
x1 <- rnorm(N)
x2 <- rnorm(N) + x1 + 1
male <- rbinom(N,1,0.48)
y <- 1 + x1 + x2 + male + rnorm(N)
df <- data.frame(y,x1,x2,male)  # data frame
df
```

```
##             y         x1         x2 male
## 1  -0.6614962 -0.2089954 -1.0249918    0
## 2  -2.2914838 -1.0775456 -1.4183792    1
## 3   1.8662480 -0.2909422  0.6574261    0
## 4   0.9816915 -2.0814960 -0.2800338    1
## 5   2.5843673 -0.1748042 -0.1387656    1
## 6   2.4716188 -1.3884311  0.3052528    1
## 7  -0.2683437 -1.2894795 -0.2504778    1
## 8   1.0231063 -0.7286876  0.2230576    1
## 9  -0.3330070 -2.5145281 -0.6465092    0
## 10  2.3482915  1.0567219  1.3457185    0
```

```
apply(df,2,mean)#across column
```

```
##          y         x1         x2       male
##  0.7720993 -0.8698188 -0.1227702  0.6000000
```

```
apply(df,1,mean)#across each row
```

```
##  [1] -0.47387087 -0.94685215  0.55818299 -0.09495958  0.81769939
##  [6]  0.59711012 -0.20207526  0.37936907 -0.87351109  1.18768298
```

```
class(apply(df,1,mean))#vector or matrix as output...here vector
```

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

```
mylist<-list(x=c(1,5,7), y=c(4,2,6), z=c(0,3,4))
mylist
```

```
## $x
## [1] 1 5 7
##
## $y
## [1] 4 2 6
##
## $z
## [1] 0 3 4
```

```
lapply(mylist, function(x) mean(x))
```

```
## $x
## [1] 4.333333
##
## $y
## [1] 4
##
## $z
## [1] 2.333333
```

```
#lapply(list) -->list
```

```
#But let's say we wanted the result in a vector, not in a list
#sapply() can take in a list as the input, and it will return a vector (or matrix). Let's try i
t:

sapply(mylist, function(x) mean(x))
```

```
##        x        y        z
## 4.333333 4.000000 2.333333
```

```
class(sapply(mylist, function(x) mean(x)))
```

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

```
typeof(sapply(mylist, function(x) mean(x)))
```

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

```
#USe of apply family:  This reduces the need to run a loop, which can take a lot longer.
span.fun<-function(x) {(max(x)-min(x))>=5}
```

```
#apply that function to the list
sapply(mylist, span.fun)
```

```
##     x     y     z
##  TRUE FALSE FALSE
```

```
class(sapply(mylist, span.fun))
```

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

```
#sapply(list)--> gave vector(vector or matrix)
```

```
#Creating a list using lapply()
#initialize list to 2 empty matrices of 2 by 3
list2<-lapply(1:2, function(x) matrix(NA, nrow=2, ncol=3))
list2
```

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

```
#rnorm(n,mean=,std=)
#initialize list to 2 matrices with random numbers from normal distribution
list2<-lapply(1:2, function(x) matrix(rnorm(6, 10, 1), nrow=2, ncol=3))
list2
```

```
## [[1]]
##          [,1]     [,2]      [,3]
## [1,] 9.068449 10.10527  8.309451
## [2,] 9.514666 10.07722 11.024555
##
## [[2]]
##          [,1]     [,2]     [,3]
## [1,] 8.988827 9.165729 10.39344
## [2,] 8.654382 9.265435 10.46211
```

```
#input list, output column sums of each matrix into a new list
lapply(list2, colSums)
```

```
## [[1]]
## [1] 18.58311 20.18249 19.33401
##
## [[2]]
## [1] 17.64321 18.43116 20.85555
```

```
##input list, output column sums into a **vector** (which binds them into a matrix)
sapply(list2, colSums)
```

```
##          [,1]     [,2]
## [1,] 18.58311 17.64321
## [2,] 20.18249 18.43116
## [3,] 19.33401 20.85555
```

```
class(sapply(list2, colSums))
```

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

```
#sapply(list,fun)--> matrix output(vector or matrix)
```

```
#instead of binding, we can stack these column sums by using tranpose function t():
t(sapply(list2, colSums))
```

```
##          [,1]     [,2]     [,3]
## [1,] 18.58311 20.18249 19.33401
## [2,] 17.64321 18.43116 20.85555
```

```
#IMPPPPPPPP: An object in R is a vector if its mode is logical, numeric, complex, character
```

```
is.vector(sapply(mylist, function(x) mean(x)))
```

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

```
#Part 1 (Data Frame)
##################################
N <- 10
x1 <- rnorm(N)
x2 <- rnorm(N) + x1 + 1
male <- rbinom(N,1,0.48)
y <- 1 + x1 + x2 + male + rnorm(N)


df <- data.frame(y,x1,x2,male)  # data frame df
df
```

```
##             y         x1         x2 male
## 1  -0.4597563 -0.9274483 -0.7107421    1
## 2   8.8269147  2.2028191  4.0003031    1
## 3  -0.1798840  0.3150100 -1.1091017    0
## 4   5.1321259  2.2274408  2.7905935    0
## 5   6.2896757  0.8905415  3.7262129    0
## 6   1.8936419 -0.2732935  0.3535457    0
## 7  -4.7183140 -2.1739489 -3.2451168    0
## 8   0.5178717 -1.4930348  0.4301682    1
## 9  -2.2182817 -2.7064166 -1.0172972    0
## 10 -0.3002988 -0.1021768 -0.2513999    0
```

```
apply(df,1,mean)                # applies function to each row
```

```
##  [1] -0.2744867  4.0075092 -0.2434939  2.5375400  2.7266075  0.4934735
##  [7] -2.5343449  0.1137513 -1.4854989 -0.1634689
```

```
apply(df,2,mean)                # applies function to each column
```

```
##          y         x1         x2       male
##  1.4783695 -0.2040507  0.4967166  0.3000000
```

```
class(apply(df,2,mean))
```

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

```
lapply(df, mean)                # returns a list
```

```
## $y
## [1] 1.47837
##
## $x1
## [1] -0.2040507
##
## $x2
## [1] 0.4967166
##
## $male
## [1] 0.3
```

```
class(lapply(df, mean))
```

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

```
sapply(df, mean)                # returns a vector
```

```
##          y         x1         x2       male
##  1.4783695 -0.2040507  0.4967166  0.3000000
```

```
class(sapply(df, mean) )
```

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

```
tapply(df$x1,df$x2,mean)        # applies function to each level of the factor
```

```
##  -3.24511676407491  -1.10910172848783   -1.0172972021211
##         -2.1739489          0.3150100         -2.7064166
## -0.710742054848627 -0.251399864496028  0.353545700923988
##         -0.9274483         -0.1021768         -0.2732935
##  0.430168242059516   2.79059347814482   3.72621289276039
##         -1.4930348          2.2274408          0.8905415
##    4.0003031464175
##          2.2028191
```

```
tapply(df$y,df$male,sum)       # applies function to each level of the factor
```

```
##        0        1
## 5.898665 8.885030
```

```
class(tapply(df$x1,df$x2,mean))
```

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

```
#Part 2 (Matrix)
# apply
# create a matrix of 10 rows x 2 columns
m <- matrix(c(1:10, 11:20), nrow = 10, ncol = 2)

m
```

```
##       [,1] [,2]
##  [1,]    1   11
##  [2,]    2   12
##  [3,]    3   13
##  [4,]    4   14
##  [5,]    5   15
##  [6,]    6   16
##  [7,]    7   17
##  [8,]    8   18
##  [9,]    9   19
## [10,]   10   20
```

```
# mean of the rows
apply(m, 1, mean)
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

```
# mean of the columns
apply(m, 2, mean)
```

```
## [1]  5.5 15.5
```

```
# divide all values by 2
apply(m, 1:2, function(x) x/2)
```

```
##       [,1] [,2]
##  [1,]  0.5  5.5
##  [2,]  1.0  6.0
##  [3,]  1.5  6.5
##  [4,]  2.0  7.0
##  [5,]  2.5  7.5
##  [6,]  3.0  8.0
##  [7,]  3.5  8.5
##  [8,]  4.0  9.0
##  [9,]  4.5  9.5
## [10,]  5.0 10.0
```

```
class(apply(m, 2, mean))#list
```

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

```
# quantiles
m <- matrix(rnorm(200), 20, 10)#20 cross 10 matrix
dim(m)
```

```
## [1] 20 10
```

```
apply(m, 1, quantile, probs = c(0.25, 0.75))#m is matrix , 1 is along row so 20 rows --> 20 colu
mns with 25 and 75 percentiles
```

```
##           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
## 25% -0.7358792 -0.4874064 -0.8383405 -0.3222198 -0.4971215 -0.2853360
## 75%  0.5406132  0.2485929  0.6836833  0.3264025  0.3571169  0.6587249
##           [,7]       [,8]       [,9]      [,10]      [,11]      [,12]
## 25% -1.5012348 -0.2197933 0.06725554 0.08314797 -0.6144078 -0.6429861
## 75%  0.6082904  0.4645109 0.97658080 0.96400666  0.6703069  0.4156332
##          [,13]       [,14]      [,15]      [,16]      [,17]      [,18]
## 25% -0.7925599 -1.43676578 0.09321268 -0.5582328 -0.6511432 -0.8651181
## 75%  0.6689333  0.09032333 0.99991366  0.8230488  0.2599016  0.1249940
##          [,19]     [,20]
## 25% -0.4011156 -1.426726
## 75%  0.4737169  0.481886
```

```
# eapply
e <- new.env()
# two environment variables, a and b
e$a <- 1:10
e$b <- 11:20
class(e)
```

```
## [1] "environment"
```

```
# mean of the variables
eapply(e, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 15.5
```

```
class(eapply(e, mean))
```

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

```
# lapply
# create a list with 2 elements
l <- list(a = 1:10, b = 11:20)
# the mean of the values in each element
lapply(l, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 15.5
```

```
# the sum of the values in each element
lapply(l, sum)
```

```
## $a
## [1] 55
##
## $b
## [1] 155
```

```
# sapply
# create a list with 2 elements
l <- list(a = 1:10, b = 11:20)
# mean of values using sapply
sapply(l, mean)
```

```
##    a    b
##  5.5 15.5
```

```
class(sapply(l, mean))
```

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

```
# tapply
# use a vector
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3,10)
tapply(x, f, mean)
```

```
##          1          2          3
## -0.4340333  0.4002820  1.0613394
```

```
class(tapply(x, f, mean))
```

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

```
###IMPPPPPPPPPPPPPPPPPPPPPPPPPPP:
#The R language has five basic data structures. In order from simplest to complex: vectors, list
s, matrices, arrays, data frames.
#A vector is what is called an array in all other programming languages except R - a collection
 of cells with a fixed size where all cells hold the same type (integers or characters or reals
 or whatever).

#A list can hold items of different types and the list size can be increased on the fly. List co
ntents can be accessed either by index (like mylist[[1]]) or by name (like mylist$age).

#A matrix is a two-dimensional vector (fixed size, all cell types the same).

#An array is a vector with one or more dimensions. So, an array with one dimension is (almost) t
he same as a vector. An array with two dimensions is (almost) the same as a matrix. An array wit
h three or more dimensions is an n-dimensional array.

#A data frame is called a table in most languages. Each column holds the same type, and the colu
mns can have header names.


#vector: fixed size, fixed datatype
#matrix: 2D Vector, fixed size, fixed datatype
#Array: >2D Vector, 1DArray-Vector,2Darray-Matrix,nDarray-Array
#list->can be increased, can have different data types
#dataframe-->table with columns (across column same data typr)
```

```
######VECTOR IMPPP
v = c(1:3)  # a vector with [1.0 2.0 3.0]
cat(v, "\n\n")
```

```
## 1 2 3
```

```
class(v)
```

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

```
typeof(v)
```

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

```
v = vector(mode="integer", 4)  # [0 0 0 0]
cat(v, "\n\n")
```

```
## 0 0 0 0
```

```
class(v)
```

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

```
typeof(v)
```

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

```
v = c("a", "b", "x")
cat(v, "\n\n")
```

```
## a b x
```

```
class(v)##class is one of atomic datatypes ie. numeric logical character and all
```

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

```
typeof(v)
```

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

```
ls = list("a", 2.2)
ls[3] = as.integer(3)
print(ls)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 2.2
##
## [[3]]
## [1] 3
```

```
cat(ls[[2]], "\n\n")
```

```
## 2.2
```

```
ls = list(name="Smith", age=22)
cat(ls$name, ":", ls$age)
```

```
## Smith : 22
```

```
class(ls)##class is list
```

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

```
m = matrix(0.0, nrow=2, ncol=3) # 2x3
print(m)
```

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

```
class(m)####matrix is class
```

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

```
arr = array(0.0, 3)  # [0.0 0.0 0.0]
print(arr)
```

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

```
class(array)#1DArray is of class array
```

```
## [1] "function"
```

```
arr = array(0.0, c(2,3))  # 2x3 matrix
print(arr)
```

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

```
class(arr)#2D array is of class matrix
```

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

```
arr = array(0.0, c(2,5,4)) # 2x5x4 n-array
print(arr)  # 40 values displayed
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
##
## , , 3
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
##
## , , 4
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
```

```
class(arr)#3D array is of class array
```

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

```
people = c("Alex", "Barb", "Carl") # col 1
ages = c(19, 29, 39)  # col 2
df = data.frame(people, ages)  # create
names(df) = c("NAME", "AGE")  # headers
print(df)
```

```
##   NAME AGE
## 1 Alex  19
## 2 Barb  29
## 3 Carl  39
```

```
class(df)#data.frame
```

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

```
#DIFFERENCE BETWEEN cat and R
u="utsav"
cat("utsav is :",u)
```

```
## utsav is : utsav
```

```
#print("utsav is : ",u)
```

```
#An essential difference between cat and print is the class of the object they return.
#print returns a character vector:
print(paste("a", 100* 1:3))
```

```
## [1] "a 100" "a 200" "a 300"
```

```
class(print(paste("a", 100* 1:3)))
```

```
## [1] "a 100" "a 200" "a 300"
```

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

```
#Thus there is [1] on left side of printed character values
```

```
#cat returns an object of class NULL.
cat(paste("a", 100* 1:3))
```

```
## a 100 a 200 a 300
```

```
class(cat(paste("a", 100* 1:3)))
```

```
## a 100 a 200 a 300
```

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

```
#Print string and variable contents on the same line in R
print(paste0("Current working dir: ", u))
```

```
## [1] "Current working dir: utsav"
```

```
cat("Current working dir: ", u)
```

```
## Current working dir:  utsav
```

```
message("Current working dir: ", u)
```

```
## Current working dir: utsav
```

```
class(message("Current working dir: ", u))
```

```
## Current working dir: utsav
```

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

```
sprintf("Current working dir: %s", u)
```

```
## [1] "Current working dir: utsav"
```

```
class(sprintf("Current working dir: %s", u))
```

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

```
#same as print-->returns character vector
```

```
cat(sprintf("Current working dir: %s\n", u))
```

```
## Current working dir: utsav
```

```
print(paste0("Current working dir: " ,u," and yes utsav is : ",u))
```

```
## [1] "Current working dir: utsav and yes utsav is : utsav"
```

```
#IDEALLY you should use print and inside paste0 or paste()
paste("Today is", date())
```

```
## [1] "Today is Mon Apr 23 03:27:28 2018"
```

```
paste0("Today is", date())
```

```
## [1] "Today isMon Apr 23 03:27:28 2018"
```

```
print(paste("This is", date()))
```

```
## [1] "This is Mon Apr 23 03:27:28 2018"
```

```
print(paste0("This is ", date()))
```

```
## [1] "This is Mon Apr 23 03:27:28 2018"
```