```
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
```

```
#IMPPPPPPPPPPPPPPPPPPPPPPPP: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: 0x000000001a2349b0>
## <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  65.57421 Treatment
## 2         2  68.14241 Treatment
## 3         3  28.15180 Treatment
## 4         4  34.55713 Treatment
## 5         5  56.90333 Treatment
## 6         6  65.77779 Treatment
## 7         7  70.40495 Treatment
## 8         8  70.72516 Treatment
## 9         9  56.73492 Treatment
## 10       10  65.62677 Treatment
## 11       11  55.89783 Treatment
## 12       12 104.78913 Treatment
## 13       13  52.19317 Treatment
## 14       14  82.45254 Treatment
## 15       15  62.33181 Treatment
## 16       16  55.62278 Treatment
## 17       17  42.72955 Treatment
## 18       18  59.59021 Treatment
## 19       19  83.81727 Treatment
## 20       20  66.32876 Treatment
## 21       21  73.44580 Treatment
## 22       22  87.11353 Treatment
## 23       23  55.82506 Treatment
## 24       24  60.61183 Treatment
## 25       25  67.52752 Treatment
## 26       26  68.92382 Treatment
## 27       27  58.71549 Treatment
## 28       28  52.98686 Treatment
## 29       29  70.35653 Treatment
## 30       30  78.66962 Treatment
## 31       31  62.53961 Treatment
## 32       32  54.01405 Treatment
## 33       33  45.11498 Treatment
## 34       34  70.17720 Treatment
## 35       35  88.98306 Treatment
## 36       36  63.33212 Treatment
## 37       37  88.55476 Treatment
## 38       38  64.36704 Treatment
## 39       39  44.33235 Treatment
## 40       40  60.13657 Treatment
## 41       41  63.99803 Treatment
## 42       42  71.50681 Treatment
## 43       43  63.20994 Treatment
## 44       44  68.74515 Treatment
## 45       45  54.81908 Treatment
## 46       46  55.04715 Treatment
## 47       47  65.25031 Treatment
## 48       48  80.67150 Treatment
## 49       49  48.46874 Treatment
## 50       50  66.37070 Treatment
## 51       51  73.18687   Control
## 52       52  55.86824   Control
```

```
## 53       53  63.78987   Control
## 54       54  30.32992   Control
## 55       55  80.49176   Control
## 56       56  54.20113   Control
## 57       57  62.21905   Control
## 58       58  59.49137   Control
## 59       59  71.62069   Control
## 60       60  65.06372   Control
## 61       61  37.51113   Control
## 62       62  60.24355   Control
## 63       63  61.56535   Control
## 64       64  59.69202   Control
## 65       65  79.26416   Control
## 66       66  56.55007   Control
## 67       67  59.77465   Control
## 68       68  54.69386   Control
## 69       69  59.39943   Control
## 70       70  46.49641   Control
## 71       71  61.57087   Control
## 72       72  70.74519   Control
## 73       73  86.20733   Control
## 74       74  66.90181   Control
## 75       75  66.86793   Control
## 76       76  71.95865   Control
## 77       77  59.67378   Control
## 78       78  57.45588   Control
## 79       79  68.02627   Control
## 80       80  74.33814   Control
## 81       81  55.10618   Control
## 82       82  62.58994   Control
## 83       83  63.55607   Control
## 84       84  60.14728   Control
## 85       85  37.72443   Control
## 86       86  84.28544   Control
## 87       87  78.02801   Control
## 88       88  57.11394   Control
## 89       89  80.74165   Control
## 90       90  55.88742   Control
## 91       91  38.46410   Control
## 92       92  57.30065   Control
## 93       93  59.19879   Control
## 94       94  49.99471   Control
## 95       95  76.44989   Control
## 96       96  64.54443   Control
## 97       97  57.70651   Control
## 98       98  65.28082   Control
## 99       99  67.86424   Control
## 100     100  56.53617   Control
```

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

```
##     patient            age             treatment
##  Min.   :  1.00   Min.   : 28.15   Treatment:50
##  1st Qu.: 25.75   1st Qu.: 56.38   Control  :50
##  Median : 50.50   Median : 62.56
##  Mean   : 50.50   Mean   : 63.06
##  3rd Qu.: 75.25   3rd Qu.: 70.22
##  Max.   :100.00   Max.   :104.79
```

```
#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
##  64.04333  62.07440
```

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

```
##  [1] 3.626437 4.508192 3.862749 3.011216 3.945118 3.482121 1.395364
##  [8] 4.796532 3.897235 1.439331 4.655969 4.183715 4.856647 1.597610
## [15] 1.951729 3.447906 3.290197 4.946005 3.011069 1.646877
```

```
#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      x       0.2670936
## 2  Team A      f       0.3893077
## 3  Team A      n       0.2230643
## 4  Team A      z       0.3792712
## 5  Team A      y       0.3369177
## 6  Team B      v       0.3036548
## 7  Team B      p       0.3360653
## 8  Team B      u       0.2169533
## 9  Team B      m       0.2987423
## 10 Team B      j       0.3025397
## 11 Team C      q       0.2826812
## 12 Team C      o       0.2603211
## 13 Team C      g       0.2524500
## 14 Team C      h       0.3216906
## 15 Team C      e       0.3767608
## 16 Team D      i       0.2236263
## 17 Team D      b       0.2572783
## 18 Team D      r       0.3616477
## 19 Team D      k       0.3381213
## 20 Team D      d       0.3078460
## 21 Team E      a       0.2978404
## 22 Team E      t       0.3109533
## 23 Team E      c       0.2398892
## 24 Team E      s       0.3188059
## 25 Team E      l       0.2628020
```

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

```
##      team       player   batting.average
##  Team A:5   a      : 1   Min.   :0.2170
##  Team B:5   b      : 1   1st Qu.:0.2603
##  Team C:5   c      : 1   Median :0.3025
##  Team D:5   d      : 1   Mean   :0.2987
##  Team E:5   e      : 1   3rd Qu.:0.3361
##             f      : 1   Max.   :0.3893
##             (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] 0.2169533
```

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

```
## [1] 0.3893077
```

```
## 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
## 0.3893077 0.3360653 0.3767608 0.3616477 0.3188059
```

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

```
##     team player batting.average
## 1 Team A      x       0.2670936
## 2 Team A      f       0.3893077
## 3 Team A      n       0.2230643
## 4 Team A      z       0.3792712
## 5 Team A      y       0.3369177
```

```
#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
```

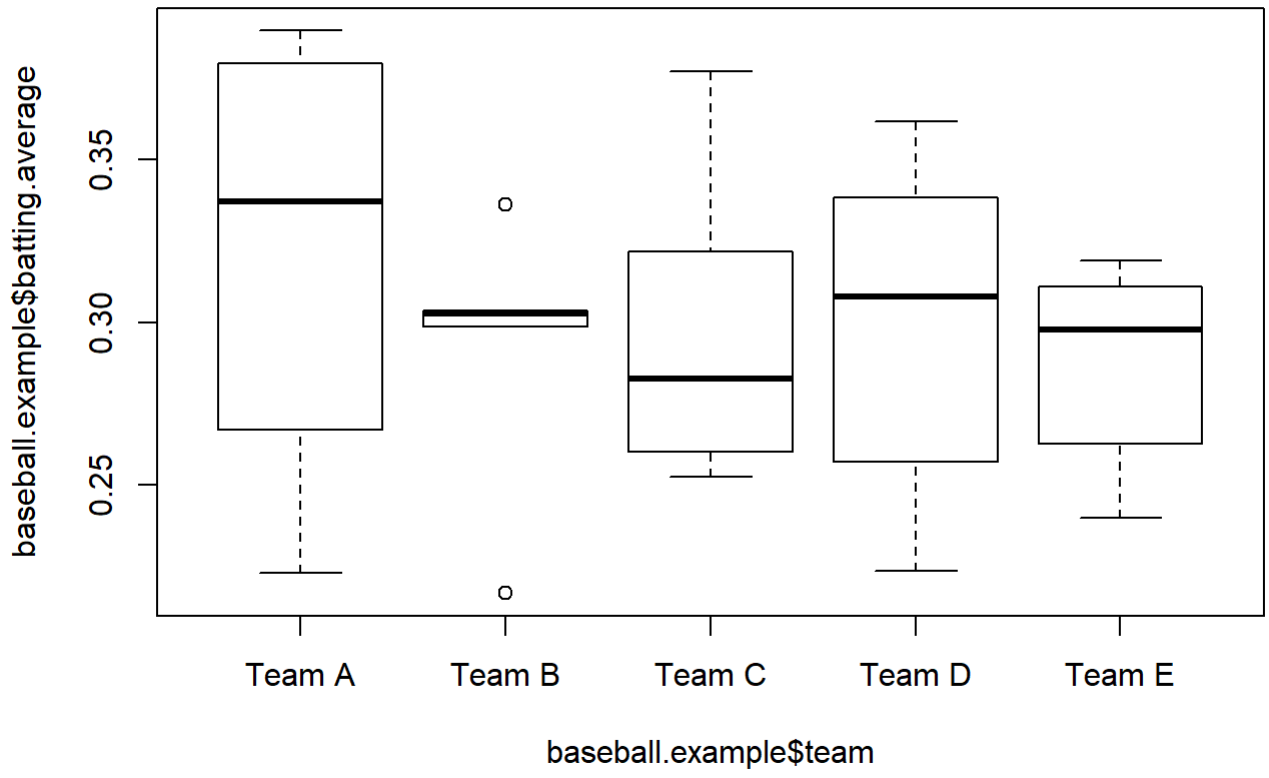
```
##        tapply.baseball.example.batting.average..baseball.example.team..
## Team A                                                        0.3893077
## Team B                                                        0.3360653
## Team C                                                        0.3767608
## Team D                                                        0.3616477
## Team E                                                        0.3188059
```

```
#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     0.3893077
## Team B     0.3360653
## Team C     0.3767608
## Team D     0.3616477
## Team E     0.3188059
```

```
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"
```

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

```
## On 2018-04-21 I realized Grover was...
## flying by the street
```

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

```
## On  2018-04-21  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 2018-04-21 I realized Grover was...
## flying by the street
```

```
#Working with List in R
```