# Basic R - List, matrix and array

Dr. Nguyen Quang Huy

Aug 30, 2020

# Creating a list

- Lists are the R objects which contain elements of different types, like numbers, strings, vectors, matrix, function and even some lists inside it.
- We can create a list using *list()* function

```
x<-1
y<-c("A","B")
z<-matrix(5,2,2)
f<-function(x){f<-x^2}
t<-list(x,y,z)
mylist<-list(x,y,z,f,t)
```

# Creating a list

```
str(mylist)
```

```
## List of 5
##  $ : num 1
##  $ : chr [1:2] "A" "B"
##  $ : num [1:2, 1:2] 5 5 5 5
##  $ :function (x)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 4 4 4 22 4 22 4
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile
##  $ :List of 3
##   ..$ : num 1
##   ..$ : chr [1:2] "A" "B"
##   ..$ : num [1:2, 1:2] 5 5 5 5
```

# Subseting a list

- **"[.]"** extracts a sub-list, the result will be a list.

```
str(mylist[3])
```

```
## List of 1
##  $ : num [1:2, 1:2] 5 5 5 5
```

```
str(mylist[5])
```

```
## List of 1
##  $ :List of 3
##   ..$ : num 1
##   ..$ : chr [1:2] "A" "B"
##   ..$ : num [1:2, 1:2] 5 5 5 5
```

# Subseting a list

- **"[[.]]"** extracts a single component from a list; it removes a level of hierarchy from the list.

```
str(mylist[[3]])
```

```
##  num [1:2, 1:2] 5 5 5 5
```

```
str(mylist[[5]])
```

```
## List of 3
##  $ : num 1
##  $ : chr [1:2] "A" "B"
##  $ : num [1:2, 1:2] 5 5 5 5
```

# Naming elements of a list

```r
names(mylist)<-c("x","y","z","f","t")
mylist$f # using "$" accessor ~ [[.]]
```

```
## function(x){f<-x^2}
```

```r
print(mylist$f(mylist$z))
```

```
##      [,1] [,2]
## [1,]   25   25
## [2,]   25   25
```

# Manipulating List Elements

- Adding an element

```
mylist[6]<-list(c(1,2,3))
mylist<-c(mylist,list(c(4,5,6)))
```

- Remove an element of a list

```
mylist[6]<-NULL
str(mylist)
```

```
## List of 6
##  $ x: num 1
##  $ y: chr [1:2] "A" "B"
##  $ z: num [1:2, 1:2] 5 5 5 5
##  $ f:function (x)
##   ..- attr(*, "srcref")= 'srcref' int [1:8] 4 4 4 22 4 22 4
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile
##  $ t:List of 3
```

# Practice

Write a function of a numeric vector **x**, named *mysummary*, with output is a list:

- The first element, named "mean", is the average value of **x**
- The second element, named "sd", is the standard deviation of **x**
- The third element, named "quantile", is a numeric vector of 5 elements: $min(x)$, $1^{st}$ quartile, the median, $3^{rd}$ quartile and $max(x)$
- The fourth element, names "plot", is the density plot of **x**

# Practice

```
mysummary<-function(x){
  m<-mean(x,na.rm=TRUE)
  sd<-sd(x,na.rm = TRUE)
  quan<-c(min(x,na.rm=TRUE),
          quantile(x,0.25,na.rm=TRUE),
          quantile(x,0.5,na.rm=TRUE),
          quantile(x,0.75,na.rm=TRUE),
          max(x))
  pl<-plot(density(x))
  mylist<-list(m,sd,quan,pl)
  names(mylist)<-c("mean","sd","quantile","plot")
  mysummary<-mylist
}
```

# Matrix

- Matrix is the R objects in which the elements are arranged in a two-dimensional.
- Matrices contains elements of the same type i.e. we can only create a matrix containing only numbers or only logical values.
- Matrices containing numeric elements are very useful in data analytics.

```
M<-matrix(1:6,2,3) # 2*3 matrix
M
```

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

# Accessing elements of a matrix

```
# Access the element at 2rd column and 1st row.
print(M[1,2])
```

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

```
# Access the element at 2nd column and 3th row.
print(M[2,3])
```

```
## [1] 6
```

```
# Access only the  2nd row.
print(M[2,])
```

```
## [1] 2 4 6
```

```
# Access only the 3rd column.
print(M[,3])
```

```
## [1] 5 6
```

# Matrix computations

```
print(M*2)
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    4    8   12
```

```
M1<-matrix(rep(c(1,2),3),2,3)
M*M1
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    4    8   12
```

```
M+M1
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    4    6    8
```

# Matrix computations

| Operation/formula | Note |
|---|---|
| $M_1 \ \% * \% \ M_2$ | Matrix multiplication, where $M_1$ is a $n * m$ and $M_2$ is a $m * k$ matrix |
| $t(M)$ | The transpose matrix of matrix $M$ |
| $det(M)$ | The determination of squared matrix $M$ |
| $solve(M)$ | The inversed matrix of squared matrix $M$ |
| $chol(M)$ | The Cholesky decomposition |
| $eigen(M)\$values$ | The eigenvalues of the matrix $M$ |
| $eigen(M)\$vectors$ | The eigenvectors of the matrix $M$ |

# Matrix application

Matrix representation of an image

```r
dat <- read_mnist() # size 60000 * 784

# matrix 28*28 of an image
MM<-matrix(mnist$train$images[8,],28,28)
MM

image(1:28,1:28,MM,
      col = gray.colors(10,start=0,end=1)) #black-white

MM<-ifelse(MM>0,255,0)
image(1:28,1:28,MM,
      col = gray.colors(10,start=0,end=1))
```

# Matrix application

Solve the system equation

$$\begin{aligned} 9x + 8y + 9z + 2t &= 42 \\ 5x + 2y + 7z + 3t &= 45 \\ 6x + 4y + 3z + 6t &= 53 \\ 8x + 2y + 5z + 6t &= 63 \end{aligned}$$

# Matrix application

```r
M<-matrix(c(c(9,5,6,8),
            c(8,2,4,2),
            c(9,7,3,5),
            c(2,3,6,6)),4,4)
v<-matrix(c(42,45,53,63),4,1)

# SOLUTION (x,y,z,t)
solve(M)%*%v
```

```
##      [,1]
## [1,]    1
## [2,]   -1
## [3,]    3
## [4,]    7
```

# Matrix application

Calculation of least squares estimates by matrix manipulation: You are fitting a multiple linear regression model

$$Y_i = \beta_0 + \beta_1 \; X_1 + \cdots + \beta_p \; X_p$$

**a.** You are using Boston data set in package MASS, the dependent variable (Y) is *medv* and there are 3 independent variables: *lstat*, *crim*, *rm*.

**b.** Set up the vector $Y$ and matrix $\mathbf{X}$

**c.** Using the following fomula to calculate the least squares estimate: $\hat{\beta} = (X^{'}X)^{-1}X^{'}Y$

**d.** Compare result in (c.) to the result from *lm* function

# Matrix application

```
library(MASS)
n<-nrow(Boston)
X<-matrix(c(rep(1,n),Boston$lstat,Boston$crim,Boston$rm),n,4)
Y<-matrix(Boston$medv,n,1)
#c. b0,b1,b2,b3
t(solve(t(X)%*%X) %*% t(X) %*% Y)
```

```
##           [,1]       [,2]       [,3]     [,4]
## [1,] -2.562251 -0.5784858 -0.1029409 5.216955
```

```
#d. check with lm
lm(medv~lstat+crim+rm,data=Boston)
```

```
##
## Call:
## lm(formula = medv ~ lstat + crim + rm, data = Boston)
##
## Coefficients:
```

# Array

- Arrays are objects which can store data in more than 2 dimensions.
- If we create an array of dimension $(n, m, k)$, it creates $k$ rectangular matrices each with $n$ rows and $m$ columns.
- Arrays can store only data type.

```
# 2 matrix 3*4
M<-array(1:24,dim=c(3,4,2))
M
```

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

# Accessing array elements

```
M[,,1] # 3*4 matrix
```

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

```
M[,4,1] # vector length = 3
```

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

```
M[1,2,2] # a number
```

```
## [1] 16
```

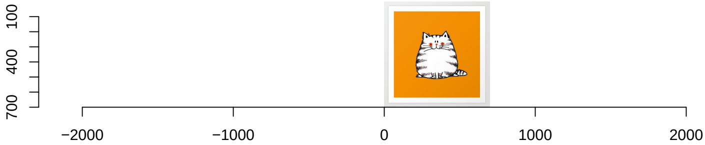
# Application of array

Images and videos are 4-dimension array

```
library(imager)
setwd("C:/Users/AD/Desktop/Tex file/Thu latex/Introduction to
img<-load.image("cat.jpg")
dim(img) # width, height, depth, spectrum
```

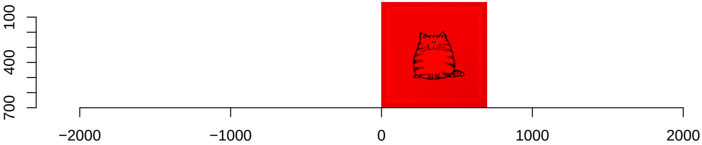
```
## [1] 700 700   1   3
```

```
plot(img) # plot a 4-dimension array
```



# Application of array

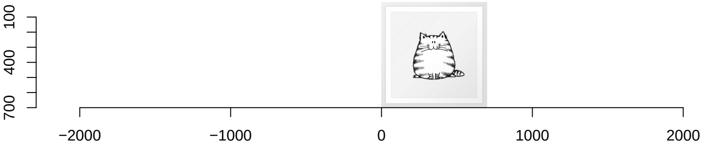
```
img1<-img
img1[,,,2:3]<-0 # red only
plot(img1) # red only
```




```
img1<-img
img1[,,,1:2]<-0 # blue only
plot(img1) # blue only
```

# Application of array

```
arr1<-img[,,1,1] # 2-dimension
img1<-array(arr1,dim=c(700,700,1,1))
plot(as.cimg(img1))
```




```
# add random (noise) into
arr2<-arr1+array(runif(700*700,0,2),dim=c(700,700))
img2<-array(arr2,dim=c(700,700,1,1)) # re dim
plot(as.cimg(img2))
```

# *apply()* function

X

apply(X ,2, sum)

Dimension — 2

1

| | | | | | |
|---|---|---|---|---|---|
| -1.7189391 | -1.0863995 | 1.0996117 | -0.55559727 | -0.1792310 | -0.8088577 |
| -2.2542126 | -1.3201873 | -2.0533779 | 1.29055209 | 0.3264156 | 0.5412132 |
| 1.9874737 | 0.6265486 | -0.3684977 | 1.40028967 | -0.7574303 | -2.3241569 |
| 0.2140376 | 0.8850445 | 1.4782993 | -1.28177703 | -0.5015628 | 1.1537703 |
| 0.9637687 | 1.3191502 | 0.8000988 | 0.09345943 | 1.4535431 | 1.0935720 |

Result

sum

| | | | | | |
|---|---|---|---|---|---|
| -0.8078717 | 0.4241565 | **0.9561342** | 0.9469269 | 0.3417346 | -0.3444591 |

# *apply()* function

- This function allow crossing the data in a number of ways and avoid use of loop constructs.
- *apply()* function can manipulate data from matrices, arrays, lists and dataframe.
- *apply()* function form the basis of more complex combinations and helps to perform operations with very few lines of code including: *lapply()* , *sapply()*, *vapply()*, *mapply()*, *rapply()*, and *tapply()* functions.

# *apply()* function

Using apply() function to normalize (to [0,1]) variables in Boston data set.

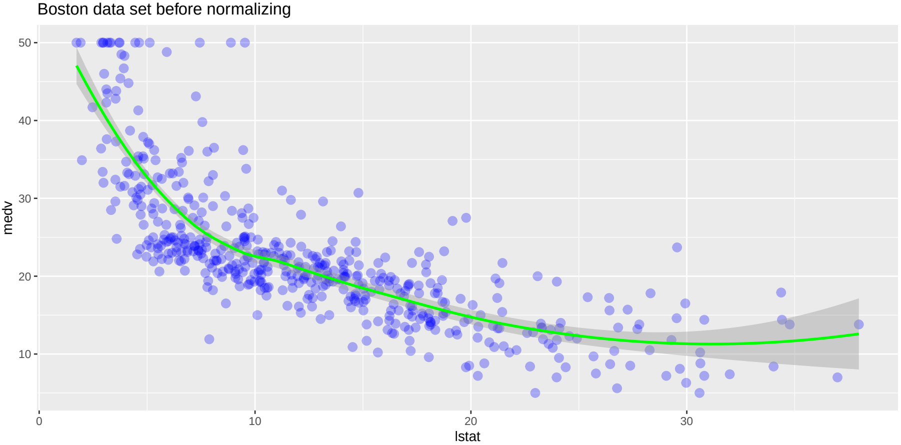
```
normalize<-function(x){x<-(x-min(x))/(max(x)-min(x))}

# normalize all variables (columns) of Boston
dat<-apply(Boston,2,normalize)
class(dat)
```

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

```
dat<-as.data.frame(dat,colnames=names(Boston))
```

# *apply()* function

Using apply() function to normalize (to [0,1]) variables in Boston data set.



# *apply()* function

Using apply() function to normalize (to [0,1]) variables in Boston data set.