# Chapter 4 Matrices, Data Frames and Lists

Math 3210

Dr. Zeng

# Outlines

- Matrices
- Arrays
- Data Frames
- Lists

# Data Structure in R

- Vectors
- Factors
- Matrices
- Arrays
- Data Frames
- Lists

# Constructing Matrix objects

A **matrix** is a rectangular way of storing data. You can simply view it as a way to store data. Matrices can be constructed using the functions **matrix()**.

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

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

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" cornere and running down the columns. If you need the matrix to be filled by rows, then add the **byrow=TRUE** argument.

We can then access elements using two indice. For example,

```
m[1,2]
```

```
[1] 3
```

```
m[1,2:3]
```

```
[1] 3 5
```

Whole rows or columns of matrices may be selected by leaving one index blank:

```
m[1,]  #display the first row of m
```

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

```
m[,1]  #display the first column of m
```

```
[1] 1 2
```

To show the dimension of the matrix, we can use **dim()**.

```
dim(m)
```

```
[1] 2 3
```

To assign column and row names, we can use

```
rownames(m)<-c("obs1","obs2")
colnames(m)<-LETTERS[1:3]
m
```

```
     A B C
obs1 1 3 5
obs2 2 4 6
```

Matrices can also be created by **cbind()**, or **rbind()**.

```r
x<-1:3
y<-10:12
z<--5:-3
cbind(x,y,z)
```

```
     x  y  z
[1,] 1 10 -5
[2,] 2 11 -4
[3,] 3 12 -3
```

```r
rbind(x,y,z)
```

```
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
z   -5   -4   -3
```

A more general way to store data is in an **array**. Arrays have multiple indices, and created using the **array()** function:

```
a<-array(1:24,c(3,4,2))
a
```

```
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

# Data Frames

- Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications.
- Most data sets are stored in R as data frames. They are like matrices, but with the columns having their own names.
- Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class.

**Example**:

The data frame *women* contains the average weights (in pounds) of American women aged 30 to 39 of particular heights (in inches):

```
women
```

```
   height weight
1      58    115
2      59    117
3      60    120
4      61    123
5      62    126
6      63    129
7      64    132
8      65    135
9      66    139
10     67    142
11     68    146
12     69    150
13     70    154
14     71    159
15     72    164
```

Sometimes, the data frames can be very large, it is not a good idea to display the entire data set. To obtain the basic information of the data set, we can use

```
nrow(women) #show the number of rows
```

```
[1] 15
```

```
ncol(women) #show the number of columns
```

```
[1] 2
```

```
dim(women)  #show both dimensions
```

```
[1] 15  2
```

```
rownames(women) #show the row names
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "
[15] "15"
```

```
colnames(women) #show the column names
```

We can use the **str()** function to get summary information for structure. It works with almost any R object and it is often a quick way to find out what you are working with.

```r
str(women)
```

```
'data.frame':   15 obs. of  2 variables:
 $ height: num  58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

To summarize the data numerically,

```r
summary(women)
```

```
     height         weight     
 Min.   :58.0   Min.   :115.0  
 1st Qu.:61.5   1st Qu.:124.5  
 Median :65.0   Median :135.0  
 Mean   :65.0   Mean   :136.7  
 3rd Qu.:68.5   3rd Qu.:148.0  
 Max.   :72.0   Max.   :164.0  
```

```r
colMeans(women) # compute the mean for each column
```

```
  height   weight 
 65.0000 136.7333 
```

```r
rowMeans(women) # compute the mean for each row
```

```
 [1]  86.5  88.0  90.0  92.0  94.0  96.0  98.0 100.0 102.5 104.5 107.0
[12] 109.5 112.0 115.0 118.0
```

# The apply() function

The apply( ) function allows you to apply a function to each row or to each column of data.

```
apply(X, MARGIN, FUN, ...) # applies a function to each row or column.

## X: a data frame or matrix
## MARGIN: 1 indicates rows, 2 indicates columns
## FUN: the function to be applied to each row or column
```

```
apply( X= trees, MARGIN= 2, FUN= mean) # same output as colMeans
```

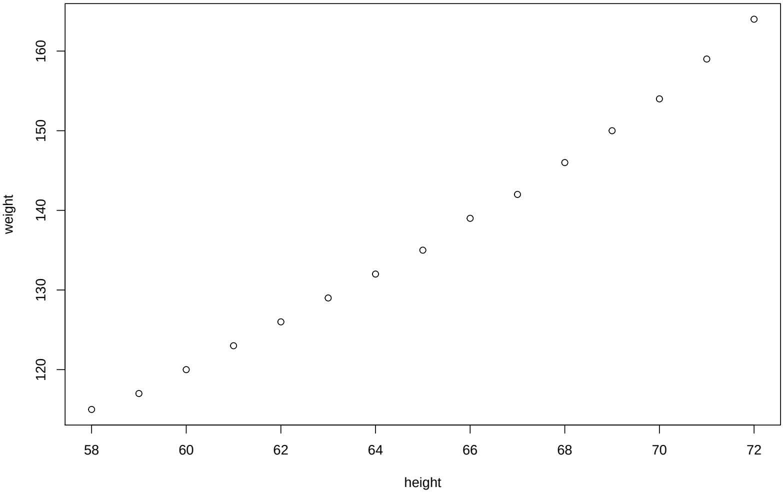
```
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

```
apply( X= trees, MARGIN= 2, FUN= median)
```

```
 Girth Height Volume
  12.9   76.0   24.2
```

To summarize the data graphically,

```
plot(weight~height,data=women)
```



# Indexing of data frames

We can extract elements from data frames using similar syntax to what was used with matrices. For example,

```r
women[7,2]
```

```
[1] 132
```

```r
women[3,]
```

```
  height weight
3     60    120
```

```r
women[4:7,1]
```

```
[1] 61 62 63 64
```

# The $ operator

Data frame columns can also be addressed using their namems using the **$** operator. For example, the weight column can be extracted as follows:

```
women$weight
```

```
 [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
```

```
women$height
```

```
 [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
```

# Conditional Selection

Sometimes, it can be useful to extract all data for cases that satisfy some criterion. For example, we can extract all heights for which the weights exceed 140 using

```
women$height[women$weight>140]
```

```
[1] 67 68 69 70 71 72
```

```
women$weight>140
```

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

```
which(women$weight>140) #show the women whose weights exceed 140
```

```
[1] 10 11 12 13 14 15
```

# The **with()** function

The **with()** function allows us to access columns of a data frame directly without using the $. For example, we can divide the weights by the heights in the women data frame using

```
with(women,weight/height)  #use the with() function
```

```
 [1] 1.982759 1.983051 2.000000 2.016393 2.032258 2.047619 2.062500
 [8] 2.076923 2.106061 2.119403 2.147059 2.173913 2.200000 2.239437
[15] 2.277778
```

```
women$weight/women$height  #without the with() function
```

```
 [1] 1.982759 1.983051 2.000000 2.016393 2.032258 2.047619 2.062500
 [8] 2.076923 2.106061 2.119403 2.147059 2.173913 2.200000 2.239437
[15] 2.277778
```

# Constructing data frames

Use the **data.frame()** function to construct data frames from vectors that already exist in your workspace:

```
x<-1:5
y<-7:3
z<-11:15
xyzdata<-data.frame(x,y,z)
xyzdata
```

```
  x y  z
1 1 7 11
2 2 6 12
3 3 5 13
4 4 4 14
5 5 3 15
```

Alternatively, we can use

```
xyzdata<-data.frame(x=1:5,y=7:3,z=11:15)
```

# Non-numeric columns in data frames

Columns of data frames can be of different types. For example, the built in data frame 'chickwts' has a numeric column and a factor.

```
chickwts
```

To view the basic information about the data frame 'chickwts'

```
str(chickwts)
```

```
'data.frame':   71 obs. of  2 variables:
 $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
 $ feed  : Factor w/ 6 levels "casein","horsebean",..: 2 2 2 2 2 2 2 2
```

```
summary(chickwts)
```

```
     weight             feed   
 Min.   :108.0   casein   :12  
 1st Qu.:204.5   horsebean:10  
 Median :258.0   linseed  :12  
 Mean   :261.3   meatmeal :11  
 3rd Qu.:323.5   soybean  :14  
 Max.   :423.0   sunflower:12  
```

# Example:

Consider the following data that might be used as a baseline in an obesity study:

```
gender<-c("M","M","F","F","F")
weight<-c(73,68,52,69,64)
obesitystudy<-data.frame(gender,weight)
obesitystudy
```

```
  gender weight
1      M     73
2      M     68
3      F     52
4      F     69
5      F     64
```

# Practice: Changing variable names

```
obesitystudy$gender
```

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

Obviously, the character vector ‘gender’ is convereted to a factor in the data frame. Suppose we wish to globally change F to Female in the data frame.

```
as.integer(obesitystudy$gender)
```

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

```
levels(obesitystudy$gender)[1]<-"Female" #F is the 1st level
obesitystudy$gender  #Check the new data frame
```

```
[1] M      M      Female Female Female
Levels: Female M
```

# Grouped data and data frames

- The natural way of storing grouped data in a data frame is to have the data themselves in one vector and parallel to that have a factor telling which data are from which group.
- However, sometimes it is desirable to have data in a separate vector for each group.
- For example, suppose we want to extract the weights of chickens in the soybean group.

```
soybeangrp<-chickwts$weight[chickwts$feed=="soybean"]
soybeangrp
```

```
 [1] 243 230 248 327 329 250 193 271 316 267 199 171 158 248
```

# The **split()** function

Alternatively, you can use the **split()** function, which generates a list of vectors according to a grouping.

```
split(chickwts$weight,chickwts$feed)
```

```
$casein
 [1] 368 390 379 260 404 318 352 359 216 222 283 332

$horsebean
 [1] 179 160 136 227 217 168 108 124 143 140

$linseed
 [1] 309 229 181 141 260 203 148 169 213 257 244 271

$meatmeal
 [1] 325 257 303 315 380 153 263 242 206 344 258

$soybean
 [1] 243 230 248 327 329 250 193 271 316 267 199 171 158 248

$sunflower
 [1] 423 340 392 339 341 226 320 295 334 322 297 318
```

# Lists

Data frames are actually a special kind of list. List in R can contain any other objects. The **list()** function is one way of organizing multiple pieces of output from functions.

# Example 1:

```
a <- matrix( c( 1, 2, 3, 4 ), nrow= 2, ncol= 2 )
b <- c(1, 3, 7)
c <- "hello!"
d <- c(TRUE, FALSE, FALSE, TRUE)
z <- list( "FirstItem"=a, "SecItem"=b, "ThirdItem"=c, "FourthItem"=d)
z
```

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

$SecItem
[1] 1 3 7

$ThirdItem
[1] "hello!"

$FourthItem
[1]  TRUE FALSE FALSE  TRUE
```

# Working with Lists

You can see the names of the objects in a list using the **names()** function, and extract parts of it:

```
names(z) #Print the names of the objects in z
```

```
[1] "FirstItem"  "SecItem"    "ThirdItem"  "FourthItem"
```

```
z$SecItem # Print the FirstItem of z
```

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

```
z[[2]] #You can use this when the elements in the list don't have names
```

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

# Example 2:

```
x <- c(2, 7, 8, 6)
y <- c(1, 3, 8, 9)
z <- c(4, 2, 7, 3)
G <- list( x, y, z )
G # elements don't have names! Try names(G)
```

```
[[1]]
[1] 2 7 8 6

[[2]]
[1] 1 3 8 9

[[3]]
[1] 4 2 7 3
```

```
G[[3]]
```

```
[1] 4 2 7 3
```

# The **lapply()** function

There are several functions which make working with lists easy. Two of them are **lapply()** and **vapply()**. The **lapply()** function "applies" another functin to every element of a list and returns the results in a new list. The general form is:

```
lapply( X, FUN )
## X: a list
## FUN: the function to be applied to each element of X
```

```
lapply(G,mean)
```

```
[[1]]
[1] 5.75

[[2]]
[1] 5.25

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

```
lapply(G,fivenum)  #returns the five numbers for each list element
```

# The **vapply()** function

In the previous example, it might be more convenient to have the results in a vector. The **vapply()** does the same thing as **lapply( )** but return the result of the function in vector form. The general form is:

```
vapply( X, FUN, FUN.VALUE ) #applies a function to each list element.
## X: a list
## FUN: the function to be applied to each element of X
## FUN.VALUE: An example of the output
```

```
vapply(X= G, FUN= mean, FUN.VALUE= 1 ) # the '1' serves as an example o
```

```
[1] 5.75 5.25 4.00
```

If we expect more than a single value, the results will be organized into a matrix, e.g.

```
vapply(X= G, FUN= summary, numeric(6) )
```

```
        [,1] [,2] [,3]
Min.    2.00 1.00 2.00
1st Qu. 5.00 2.50 2.75
Median  6.50 5.50 3.50
Mean    5.75 5.25 4.00
3rd Qu. 7.25 8.25 4.75
Max.    8.00 9.00 7.00
```

```
vapply(X= G, FUN= summary, FUN.VALUE= c( 1,1,1,1,1,1 ) )
```

```
        [,1] [,2] [,3]
Min.    2.00 1.00 2.00
1st Qu. 5.00 2.50 2.75
Median  6.50 5.50 3.50
Mean    5.75 5.25 4.00
3rd Qu. 7.25 8.25 4.75
Max.    8.00 9.00 7.00
```

# More examples:

```
xvec <- c( 2, 4, 8)
yvec <- c(5, 4, 10, 44)
range(xvec) # no element names
```

```
[1] 2 8
```

```
vecsList <- list( xvec, yvec)
vapply(X= vecsList, FUN= range, FUN.VALUE= c( 1, 1 ) )
```

```
     [,1] [,2]
[1,]    2    4
[2,]    8   44
```

```
vapply(X= vecsList, FUN= range, numeric(2) )
```

```
     [,1] [,2]
[1,]    2    4
[2,]    8   44
```

Sometimes, we need to add names to the outputs.

```r
vapply( vecsList , range, c( Min. = 1, Max. = 0 ) )
```

```
     [,1] [,2]
Min.    2    4
Max.    8   44
```