



# Matrices, Lists, Dataframes, S3 and S4 Objects.

Elements of the R language

Marcin Kierczak

# Contents of the lecture

- variables and their types
- operators
- vectors
- numbers as vectors
- strings as vectors
- **matrices**
- **lists**
- **data frames**
- **objects**
- repeating actions: iteration and recursion
- decision taking: control structures
- functions in general
- variable scope
- core functions

# Matrices

A **matrix** is a 2-dimensional data structure, like vector, it consists of elements of the same type. A matrix has *rows* and *columns*.

Say, we want to construct this matrix in R:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
X <- matrix(1:9, # a sequence of numbers to fill in
            nrow=3, # three rows (alt. ncol=3)
            byrow=T) # populate matrix by row
```

X

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

# Matrices — indexing

Elements of a matrix are retrieved using the '[]' notation, like we have seen for vectors. Here, we have to specify 2 dimensions -- the row and the column:

```
X[1,2] # Retrieve element from the 1st row, 2nd column  
X[3,] # Retrieve the entire 3rd row  
X[,2] # Retrieve the 2nd column
```

```
## [1] 2  
## [1] 7 8 9  
## [1] 2 5 8
```

# Matrices — indexing cted.

```
X[c(1,3),] # Retrieve rows 1 and 3  
X[c(1,3),c(3,1)]
```

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

# Matrices — dimensions

To check the dimensions of a matrix, use `dim()`:

```
X  
dim(X) # 3 rows and 3 columns
```

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

Nobody knows why `dim()` does not work on vectors... use `length()` instead.

# Matrices — operations

Usually the functions that work for a vector also work for matrices. To order a matrix with respect to, say, 2nd column:

```
X <- matrix(sample(1:9,size = 9), nrow = 3)
ord <- order(X[,2])
X[ord,]
```

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

# Matrices — transposition

To **transpose** a matrix use `t()`:

```
X
t(X)
```

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

Nobody knows why `dim()` does not work on vectors... use `length()` instead.



## Matrices — operations 2

To get the diagonal, of the matrix:

```
X  
diag(X) # get values on the diagonal
```

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

# Matrices — operations, triangles

To get the upper or the lower triangle use **upper.tri()** and **lower.tri()** respectively:

```
X # print X
upper.tri(X) # which elements form the upper triangle
X[upper.tri(X)] <- 0 # set them to 0
X # print the new matrix
```

```
##      [,1] [,2] [,3]
## [1,]    5    4    2
## [2,]    7    8    9
## [3,]    1    6    3
##      [,1] [,2] [,3]
## [1,] FALSE TRUE  TRUE
## [2,] FALSE FALSE TRUE
## [3,] FALSE FALSE FALSE
##      [,1] [,2] [,3]
## [1,]    5    0    0
## [2,]    7    8    0
## [3,]    1    6    3
```

# Matrices — multiplication

Different types of matrix multiplication exist:

```
A <- matrix(1:4, nrow = 2, byrow=T)
B <- matrix(5:8, nrow = 2, byrow=T)
A * B # Hadamard product
A %*% B # Matrix multiplication
# A %x% B # Kronecker product
# A %o% B # Outer product (tensor product)
```

```
##      [,1] [,2]
## [1,]    5   12
## [2,]   21   32
##      [,1] [,2]
## [1,]   19   22
## [2,]   43   50
```

# Matrices — outer

Outer product can be useful for generating names

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

```
##      [,1] [,2] [,3] [,4]  
## [1,] "a-A" "a-B" "a-C" "a-D"  
## [2,] "b-A" "b-B" "b-C" "b-D"  
## [3,] "c-A" "c-B" "c-C" "c-D"  
## [4,] "d-A" "d-B" "d-C" "d-D"
```

# Expand grid

But `expand.grid()` is more convenient when you want, e.g. generate combinations of variable values:

```
expand.grid(height = seq(120, 121),  
            weight = c('1-50', '51+'),  
            sex = c("Male", "Female"))
```

```
##   height weight  sex  
## 1    120   1-50  Male  
## 2    121   1-50  Male  
## 3    120    51+  Male  
## 4    121    51+  Male  
## 5    120   1-50 Female  
## 6    121   1-50 Female  
## 7    120    51+ Female  
## 8    121    51+ Female
```

# Matrices — apply

Function **apply** is a very useful function that applies a given function to either each value of the matrix or in a column/row-wise manner. Say, we want to have mean of values by column:

```
X  
apply(X, MARGIN=2, mean) # MARGIN=1 would do it for rows
```

```
##      [,1] [,2] [,3]  
## [1,]    5    0    0  
## [2,]    7    8    0  
## [3,]    1    6    3  
## [1] 4.333333 4.666667 1.000000
```

# Matrices — apply cted.

And now we will use *apply()* to replace each element it a matrix with its deviation from the mean squared:

```
X
my.mean <- mean(X)
apply(X, MARGIN=c(1,2),
      function(x, my.mean) (x - my.mean)^2,
      my.mean)
```

```
##      [,1] [,2] [,3]
## [1,]    5    0    0
## [2,]    7    8    0
## [3,]    1    6    3
##      [,1]      [,2]      [,3]
## [1,]  2.777778 11.111111 11.111111
## [2,] 13.444444 21.777778 11.111111
## [3,]  5.444444  7.111111  0.111111
```

## Matrices — useful fns.

While *apply()* is handy, it is a bit slow and for the most common statistics, there are special functions col/row Sums/Means:

```
X  
colSums(X)
```

```
##      [,1] [,2] [,3]  
## [1,]    5    0    0  
## [2,]    7    8    0  
## [3,]    1    6    3  
## [1] 13 14  3
```

These functions are faster!



# Matrices — adding rows/columns

One may wish to add a row or a column to an already existing matrix or to make a matrix out of two or more vectors of equal length:

```
x <- c(1,1,1)
y <- c(2,2,2)
cbind(x,y)
rbind(x,y)
```

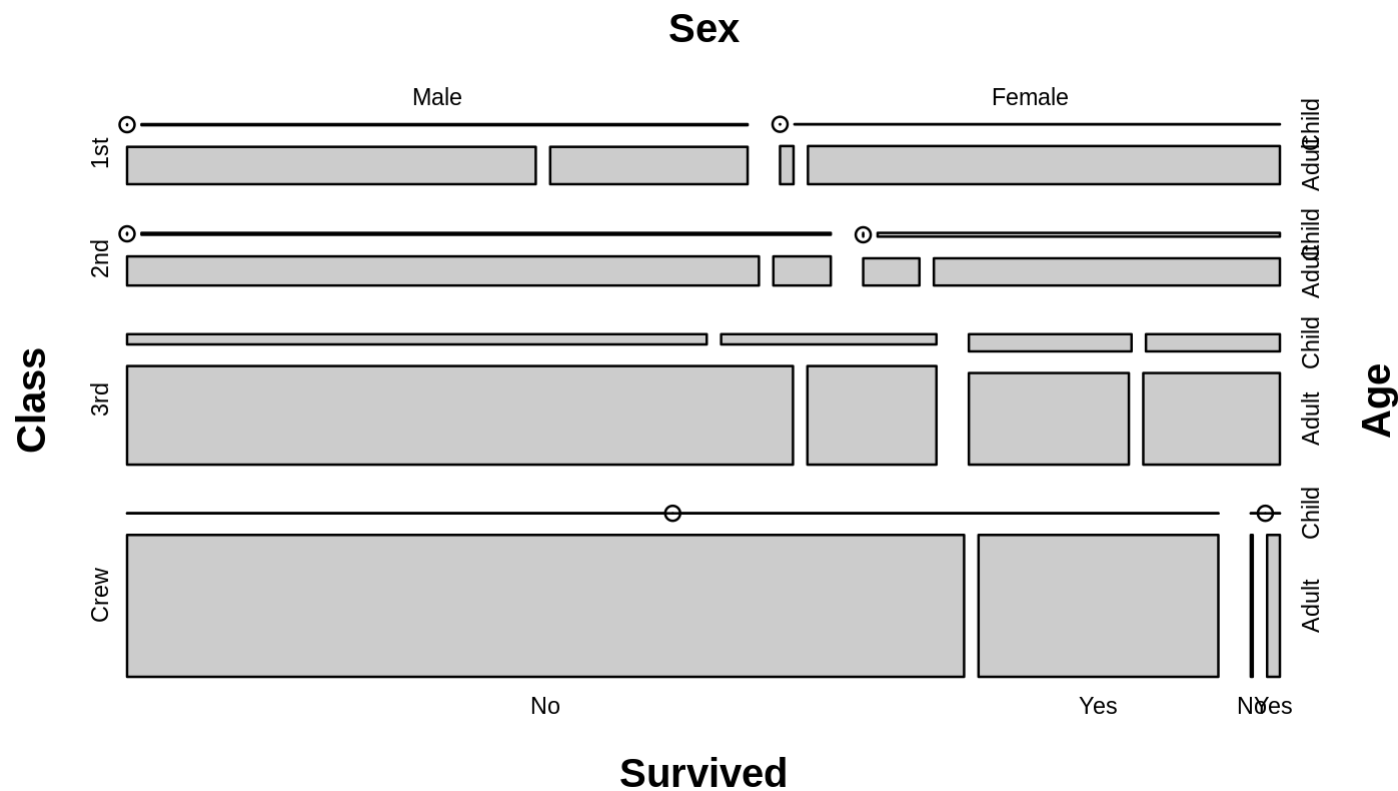
```
##      x y
## [1,] 1 2
## [2,] 1 2
## [3,] 1 2
##      [,1] [,2] [,3]
## x      1   1   1
## y      2   2   2
```

# Matrices — more dimensions

```
dim(Titanic)
```

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

```
library(vcd)
mosaic(Titanic, gp_labels=gpar(fontsize=7))
```



# Lists — collections of various data types

A list is a collection of elements that can be of various data types:

```
name <- c('R2D2', 'C3PO', 'BB8')
weight <- c(21, 54, 17)
data <- list(name=name, weight)
data
data$name
data[[1]]
```

```
## $name
## [1] "R2D2" "C3PO" "BB8"
##
## [[2]]
## [1] 21 54 17
##
## [1] "R2D2" "C3PO" "BB8"
## [1] "R2D2" "C3PO" "BB8"
```

# Lists — collections of various data types

Elements of a list can also be different data structures:

```
weight <- matrix(sample(1:9, size = 9), nrow=3)
data <- list(name, weight)
data
data[[2]][3]
```

```
## [[1]]
## [1] "R2D2" "C3PO" "BB8"
##
## [[2]]
##      [,1] [,2] [,3]
## [1,]    1    4    5
## [2,]    6    2    3
## [3,]    7    9    8
##
## [1] 7
```

# Data frames

A **data frame** or a **data table** is a data structure very handy to use. In this structure elements of every column have the same type, but different columns can have different types. Technically, a data frame is a list of vectors...

```
df <- data.frame(c(1:5),  
                LETTERS[1:5],  
                sample(c(TRUE, FALSE), size = 5,  
                      replace=T))
```

df

```
##      c.1.5. LETTERS.1.5. sample.c.TRUE..FALSE...size...5..replace...T.  
## 1      1      A                                TRUE  
## 2      2      B                                TRUE  
## 3      3      C                                TRUE  
## 4      4      D                                FALSE  
## 5      5      E                                TRUE
```

## Data frames — cted.

As you have seen, columns of a data frame are named after the call that created them. Not always the best option...

```
df <- data.frame(no=c(1:5),  
                 letter=c('a','b','c','d','e'),  
                 isBrown=sample(c(TRUE, FALSE),  
                                size = 5,  
                                replace=T))
```

df

```
##   no letter isBrown  
## 1  1      a    TRUE  
## 2  2      b    TRUE  
## 3  3      c   FALSE  
## 4  4      d    TRUE  
## 5  5      e    TRUE
```

# Data frames — accessing values

As you have seen, columns of a data frame are named after the call that created them. Not always the best option...

```
df[1,] # get the first row
df[,2] # the first column
df[2:3, 'isBrown'] # get rows 2-3 from the isBrown column
df$letter[1:2] # get the first 2 letters
```

```
## no letter isBrown
## 1 1 a TRUE
## [1] "a" "b" "c" "d" "e"
## [1] TRUE FALSE
## [1] "a" "b"
```

# Data frames — factors

An interesting observation:

```
df$letter  
df$letter <- as.character(df$letter)  
df$letter
```

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



## Data frames — factors cted.

To treat characters as characters at data frame creation time, one can use the **stringsAsFactors** option set to TRUE:

```
df <- data.frame(no=c(1:5),  
                 letter=c("a","b","c","d","e"),  
                 isBrown=sample(c(TRUE, FALSE),  
                                size = 5,  
                                replace=T),  
                 stringsAsFactors = TRUE)  
df$letter
```

```
## [1] a b c d e  
## Levels: a b c d e
```

Well, as you see, it did not work as expected...

# Data frames — names

To get or change row/column names:

```
colnames(df) # get column names
rownames(df) # get row names
rownames(df) <- letters[1:5]
rownames(df)
df['b', ]
```

```
## [1] "no"      "letter"  "isBrown"
## [1] "1" "2" "3" "4" "5"
## [1] "a" "b" "c" "d" "e"
##   no letter isBrown
## b  2      b    TRUE
```

# Data frames — merging

A very useful feature of R is merging two data frames on certain key using **merge**:

```
df1 <- data.frame(no=c(1:5),  
                  letter=c("a","b","c","d","e"))  
df2 <- data.frame(no=c(1:5),  
                  letter=c("A","B","C","D","E"))  
merge(df1, df2, by='no')
```

```
##   no letter.x letter.y  
## 1  1      a      A  
## 2  2      b      B  
## 3  3      c      C  
## 4  4      d      D  
## 5  5      e      E
```

# Objects — type vs. class

An object of class **factor** is internally represented by numbers:

```
size <- factor('small')  
class(size) # Class 'factor'  
mode(size) # Is represented by 'numeric'  
typeof(size) # Of integer type
```

```
## [1] "factor"  
## [1] "numeric"  
## [1] "integer"
```

# Objects — structure

Many functions return **objects**. We can easily examine their **structure**:

```
his <- hist(1:5, plot=F)
str(his)
object.size(hist) # How much memory the object consumes
```

```
## List of 6
## $ breaks   : int [1:5] 1 2 3 4 5
## $ counts   : int [1:4] 2 1 1 1
## $ density  : num [1:4] 0.4 0.2 0.2 0.2
## $ mids     : num [1:4] 1.5 2.5 3.5 4.5
## $ xname    : chr "1:5"
## $ equidist: logi TRUE
## - attr(*, "class")= chr "histogram"
## 1240 bytes
```

# Objects — fix

We can easily modify values of object's **attributes**:

```
attributes(his)
attr(his, "names")
#fix(his) # Opens an object editor
```

```
## $names
## [1] "breaks"    "counts"    "density"   "mids"      "xname"     "equidist"
##
## $class
## [1] "histogram"
##
## [1] "breaks"    "counts"    "density"   "mids"      "xname"     "equidist"
```

# Lists as S3 classes

A list that has been named, becomes an S3 class:

```
my.list <- list(numbers = c(1:5),  
               letters = letters[1:5])  
class(my.list)  
class(my.list) <- 'my.list.class'  
class(my.list) # Now the list is of S3 class
```

```
## [1] "list"  
## [1] "my.list.class"
```

However, that was it. We cannot enforce that *numbers* will contain numeric values and that *letters* will contain only characters. S3 is a very primitive class.

# S3 classes

For an S3 class we can define a *generic function* applicable to all objects of this class.

```
print.my.list.class <- function(x) {  
  cat('Numbers:', x$numbers, '\n')  
  cat('Letters:', x$letters)  
}  
print(my.list)
```

```
## Numbers: 1 2 3 4 5  
## Letters: a b c d e
```

But here, we have no error-proofing. If the object will lack *numbers*, the function will still be called:

```
class(his) <- 'my.list.class' # alter class  
print(his) # Gibberish but no error...
```

```
## Numbers:  
## Letters:
```



## S3 classes — still useful?

Well, S3 class mechanism is still in use, esp. when writing **generic** functions, most common examples being *print* and *plot*. For example, if you plot an object of a `Manhattan.plot` class, you write *plot(gwas.result)* but the true call is: *plot.manhattan(gwas.result)*. This makes life easier as it requires less writing, but it is up to the function developers to make sure everything works!

# S4 class mechanism

S4 classes are more advanced as you actually define the structure of the data within the object of your particular class:

```
setClass('gene',  
        representation(name='character',  
                        coords='numeric')  
        )  
my.gene <- new('gene', name='ANK3',  
              coords=c(1.4e6, 1.412e6))
```

## S4 class — slots

The variables within an S4 class are stored in the so-called **slots**. In the above example, we have 2 such slots: *name* and *coords*. Here is how to access them:

```
my.gene@name # access using @ operator  
my.gene@coords[2] # access the 2nd element in slot coords
```

```
## [1] "ANK3"  
## [1] 1412000
```

See you at the next lecture!

Graphics from  freepik.com

Created: 16-Sep-2021 • Roy Francis • SciLifeLab • NBIS

