Data Manipulation

Arthur Allignol

arthur.allignol@uni-ulm.de

Table of Contents

Reading Data

scan()

scan() reads data into a vector or list from the console or from file

- scan is more appropriate when all the data to be read are of the same mode
- Arguments:
  - file: Name of a file. When "", reads from the console
  - what: Type of what gives the type of data to be read. what can also be a list
  - scan calls can be embedded in a call to matrix

```
matrix(scan(), ncol = 3, byrow = TRUE)
```

Data Frames

`read.table`

- The `read.table` function is used to read data into R in the form of a data frame, i.e., data with mixed modes
- `read.table` expects each field (variable) to be separated by separators (by default, spaces, tabs, newlines or carriage returns)
    - The `sep` argument can be used to specify an alternative separator
- R provides convenience functions for reading comma- and tab-separated data

`read.csv`	Separated by `,`
`read.csv2`	Separated by `;` decimal point `,`
`read.delim`	Separated by tabs
`read.delim2`	Separated by tabs, decimal point `,`

These functions are wrappers for `read.table` with the `sep` argument set appropriately

Data Frames

`read.table`: Useful options

`file`	File to be read or a *connection*
`sep`	e.g., `"\t"`, `","`
`dec`	Specify decimal point (default is `.`)
`header`	`TRUE` if the the first line are the column names (default to `TRUE` for `read.csv`...)
`col.names`	A vector of column names
`stringsAsFactors`	Logical. If `FALSE`, prevent the automatic conversion of character strings into factors
`na.strings`	By default, NA, NaN, `Inf` and `-Inf` are considered as missing values. Change this behaviour using `na.strings`
`skip` and `nrows`	Number of lines to skip and number of lines to read, respectively
`fill`	If `TRUE`, observations with fewer variables are filled with NAs or blanks
`colClasses`	Specify the modes of the columns to be read
`fileEncoding`	Encoding of the file. Useful for non ASCII characters from other platforms

Data Frames

Fixed Width Input Files

- Files without delimiters but for which each variable is stored in one column
- Can be read in R using the `read.fwf`
  - `file`: the file to be read
  - `widths`: vector containing the widths of the fields to be read

readLines

- `readLines` reads some or all text lines from a file or a connection
- Useful, for example, if only some lines of an enormous file need to be read
  - `con`: a connection or file name
  - n: The maximal number of lines to read. Negative values indicate that the whole file should be read

Connections

- Connections provide a flexible way to read data from a variety of sources
- Connections can be used as input for, e.g., `read.table` or `readLines`

Function	Data source
`file`	Local files
`url`	Remote read via http ot ftp
`gzfile`	Local gzipped file
`unz`	Local zip archive
`pipe`	Output from a command
...	

Connections

```
myCon <- url("http://taz.de/")
aa <- readLines(myCon, 3)
aa

## [1] "<!DOCTYPE html SYSTEM \"about:legacy-compat\">"
## [2] "<html xmlns=\"http://www.w3.org/1999/xhtml\" xmlns:my=\"mynames\"
## [3] "\t\tContent Management: openNewspaper www.opennewspaper.org based

close(myCon)
```

Other useful functions

Function	Format
`load`	Read R data format `.rda`, `.RData`
`read.dta`	Read data saved by Stata (**foreign** package)
`read.spss`	Read data from SPSS (**foreign**)
`read.ssd`, `read.xport`	Read SAS files

- Read a data from Excel
  - Export from Excel into a .csv file
  - On Windows, the **RODBC** package permits to access Excel files
  - **gdata** package — for all platforms. Requires some specific perl modules to be installed
  - **xlsx** package. Requires Java / installation tricky
  - **readxl** package. Not tested (released 15.4.2015)

The **readr** package

The **readr** package provides alternatives to the base `read.***` functions that are

- 10 times faster (according to `http://blog.rstudio.org/2015/04/09/readr-0-1-0/`)
- more consistent
- more flexible column specification
- ...

`read_csv`	Separated by `,`
`read_csv2`	Separated by `;` decimal point `,`
`read_tsv`	Separated by tabs
`read_delim`	Separated by arbitrary delimiter

The **readr** package

Three important arguments (similar for all functions)

`file` File name

`col_names` column name; equivalent to `header`.

- `TRUE` ⇔ `header = TRUE` in base R
- `FALSE`
- A character vector to use as column names

`col_types` Override the default column types

Default to `stringsAsFactors = FALSE`!

Writing Data

- The `save` function can be used to save R objects

```r
save(a_data_set, x, y, file = "my_data.rda")
```

- The `write` function
  - Takes an R object and the name of a file or a connection as arguments
  - Writes a ASCII representation of the object
  - The `ncolumns` argument specifies the number of values to write on each line

Writing Data

- The `write.table` function
  - Requires the name of a data set or matrix (if no file is specified, `write.table()` writes into the console)
  - The `file` argument specifies the destination. `file` can also be a connection
  - `row.names` and `col.names` specify whether to write the rows' and columns' names, respectively
  - `sep` specifies the separator. Default is blank
  - `write.csv` and `write.csv2` available for writing comma-separated files

Your Turn

- The zip file `data.zip` contains several versions of the same (fake) data set
  - `data1.rda` can be opened using `load()` and will serve as reference
  - `data2` to `data5` are ASCII files
  - `data6.xls` is an Excel file that is rather realistic
- Read all the data sets in R and check whether they compare to the original using, e.g.,

```
all(dd1 == dd2) # dd2 being another data set
```

Table of Contents

Factors

- Factor are variables in R that take on a limited number of different values
    - Categorical variables
    - Ordinal variables
- Factors are useful for statistical modelling as ordinal variables should be treated differently than continuous variables
- Factors are also useful for statistical report generation. Think SAS labels

Factors

- Factors are stored internally as numeric values
- A corresponding set of characters is used for displaying

```
aa <- factor(c("cats", "dogs", "apples"))
aa

## [1] cats   dogs   apples
## Levels: apples cats dogs

as.integer(aa)

## [1] 2 3 1
```

Factor Creation

- Factors are created using the `factor` function
- The `levels` argument permits to control the order
- The `labels` argument is used to change the levels' names
- `ordered = TRUE` creates an ordered factor (ordinal variable)

```r
set.seed(21324)
data <- sample(c(1, 2, 3), 10, TRUE)
f0 <- factor(data)
f1 <- factor(data, levels = c(2, 3, 1))
f2 <- factor(data, labels = c("I", "II", "III"))
f3 <- factor(data, levels = c(2, 3, 1),
             labels = c("II", "III", "I"))
```

```
table(f0)
```

```
## f0
## 1 2 3
## 4 3 3
```

```
table(f1)
```

```
## f1
## 2 3 1
## 3 3 4
```

```
table(f2)
```

```
## f2
##   I  II III
##   4   3   3
```

```
table(f3)
```

```
## f3
##  II III   I
##   3   3   4
```

Factors

- The `levels()` function can be used to change the labels once a factor has been created

```
levels(f0) <- c("I", "II", "III")
f0

##  [1] II  I   III III III I   I   II  I   II
## Levels: I II III
```

- The reference level of a factor can be changed using the `relevel` function

```
f0 <- relevel(f0, "II")
f0

##  [1] II  I   III III III I   I   II  I   II
## Levels: II I III
```

Ordered Factors

```
set.seed(433443534)
mon <- sample(month.name, 29, TRUE)
table(factor(mon))
```

```
##
##      April    August  December  February   January      July      June
##          3         2         5         2         3         3         2
##      March       May  November September
##          1         3         3         2
```

```
mon2 <- factor(mon, levels = month.name,
               ordered = TRUE)
table(mon2)
```

```
## mon2
##    January  February     March     April       May      June      July
##          3         2         1         3         3         2         3
##     August September   October  November  December
##          2         2         0         3         5
```

Order operator can be used with ordered factors

When Factors Are a PITA

```r
set.seed(423423)
ff <- factor(sample(1:4, 10, TRUE))
```

```r
mean(ff)

## Warning in mean.default(ff):  argument is not numeric or logical:
returning NA

## [1] NA

ff + 10

## Warning in Ops.factor(ff, 10):  ’+’ not meaningful for factors

##  [1] NA NA NA NA NA NA NA NA NA NA

c(ff, 10) # Not a factor anymore

##  [1]  1  2  1  1  2  1  3  1  2  3 10
```

When Factors Are a PITA

```
(a <- factor(sample(letters, 10, replace = TRUE)))

##  [1] a u o j i h d g n d
## Levels: a d g h i j n o u

(b <- factor(sample(letters, 10, replace = TRUE)))

##  [1] t y k g v k p b d d
## Levels: b d g k p t v y
```

```
c(a, b)

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

When Factors Are a PITA

```
(a <- factor(sample(letters, 10, replace = TRUE)))

##  [1] a u o j i h d g n d
## Levels: a d g h i j n o u

(b <- factor(sample(letters, 10, replace = TRUE)))

##  [1] t y k g v k p b d d
## Levels: b d g k p t v y
```

```
c(a, b)

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

```
factor(c(as.character(a), as.character(b)))

##  [1] a u o j i h d g n d t y k g v k p b d d
## Levels: a b d g h i j k n o p t u v y
```

Factors

- Pros
  - Needed for modelling categorical variable
  - Memory efficient, i.e., factors only need to store values as integer and the unique levels as character strings
  - Nice output

```
table(factor(c(1, 2, 3),
             labels = c("Healthy", "Diseased", "Dead")))

##
##  Healthy Diseased     Dead
##        1        1        1
```

- Cons
  - Require to be cautious for some data manipulation
- I'd recommend reading data using the option `stringsAsFactors=FALSE` and transform variables into factors as needed

Dates

R provides several options to deal with dates, which is a challenging problem, i.e., time zones, daylight savings, leap second, ...

- `as.Date` handles dates without time
- The **chron** package handles dates and times, but without support for time zones
- The `POSIXct` and `POSIXlt` allow for dates and times with control for time zones
- The **lubridate** packages is supposed to facilitate the use of dates and times in R

Rule of thumb: Use the simplest technique possible. If you only have dates, use `as.Date`

Dates

as.Date

- as.Date accepts a variety of input style through the format argument
- Default is yyyy-mm-dd

```
as.Date("2014-06-12")

## [1] "2014-06-12"
```

```
as.Date("12.6.2014", format = "%d.%m.%Y")

## [1] "2014-06-12"
```

```
as.Date("12 June 14", format = "%d %B %y")

## [1] "2014-06-12"
```

See ?strptime for a complete list of format symbols

Dates

as.Date

- Internally, dates are stored as the number of days since January 1, 1970
- `as.numeric` can be used to convert a date to its numeric form

```
as.integer(as.Date("2014-06-12"))

## [1] 16233
```

- The `weekdays` and `months` functions can be used to extract the dates' components
- Calculation on dates: See `?Ops.Date`. Addition, subtraction, logical operations (==, <, . . .) are available

Dates

POSIXct and POSIXlt

- POSIXct is represented as seconds since January 1, 1970 GMT
- POSIXlt is represented as a list

→ Use POSIXct for calculation and POSIXlt for extracting date components

```
(t1 <- as.POSIXct("2014-06-12 10:15:00"))

## [1] "2014-06-12 10:15:00 CEST"

(t2 <- as.POSIXlt("2014-06-12 10:15:00"))

## [1] "2014-06-12 10:15:00 CEST"
```

Dates

POSIXct and POSIXlt

```
## Internal representation of t2
str(unclass(t2))

## List of 11
##  $ sec   : num 0
##  $ min   : int 15
##  $ hour  : int 10
##  $ mday  : int 12
##  $ mon   : int 5
##  $ year  : int 114
##  $ wday  : int 4
##  $ yday  : int 162
##  $ isdst : int 1
##  $ zone  : chr "CEST"
##  $ gmtoff: int NA
```

Your Turn

Consider the data set `data7.csv`

1. Read the data
2. Compute the age of the patients
3. `CO_DIABETES` equals 1 if patients have diabetes, otherwise 0
   - Create a factor `Diabetes` with levels `Yes` and No with reference value No
4. `CO_LIVER` equals 1 for mild liver disease, 2 for severe liver disease and 0 for no liver disease
   - Create a factor `Liver_Disease` with levels `Mild`, `Severe`, No with reference value No
5. Create a factor `Gender` with levels `Female` (`sex == 0`) and `Male` (`sex == 1`)

Table of Contents

1. Reading and Writing Data
2. Factors and Dates
3. Data Manipulation
4. Data Aggregation

Subscripting

- Logical subscripts

```
nums <- c(12, 9, 8, 14, 7, 16, 3, 2, 9)
nums > 10

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

```
nums[nums > 10]

## [1] 12 14 16

which(nums > 10)

## [1] 1 4 6

nums[which(nums > 10)]

## [1] 12 14 16
```

```
nums[nums > 10] <- 0
nums

## [1] 0 9 8 0 7 0 3 2 9
```

Subscripting Matrices and Arrays

```
(mat <- matrix(1:12, 4, 3))
```

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

```
mat[, c(3, 1)]
```

```
##      [,1] [,2]
## [1,]    9    1
## [2,]   10    2
## [3,]   11    3
## [4,]   12    4
```

Subscripting Matrices and Arrays

```
mat[, 1]
```

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

```
mat[, 1, drop = FALSE]
```

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

```
mat[mat > 4]
```

```
## [1]  5  6  7  8  9 10 11 12
```

Lists

Lists are the most general R object.

```
(ll <- list(a = 1:3, b = month.name[1:5], c = c(TRUE, FALSE),
            d = data.frame(y = rnorm(5), x = rbinom(5, 1, .5))))

## $a
## [1] 1 2 3
##
## $b
## [1] "January"  "February" "March"    "April"    "May"
##
## $c
## [1]  TRUE FALSE
##
## $d
##            y x
## 1 -1.1065764 0
## 2  1.6957258 0
## 3 -1.0641906 1
## 4 -0.0415854 1
## 5  0.8534742 0
```

Lists

```
class(ll[[4]]); class(ll[["d"]]); class(ll$d)

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

class(ll[4])

## [1] "list"
```

```
ll[c(1, 3)]

## $a
## [1] 1 2 3
##
## $c
## [1]  TRUE FALSE
```

Subscripting Data Frames

```
set.seed(4234234)
(df <- data.frame(x = c(rnorm(3), NA, 3),
                  y = c(NA, rexp(2, 0.01), NA, 3)))
```

```
##            x         y
## 1  1.7547348        NA
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
## 4         NA        NA
## 5  3.0000000   3.00000
```

```
df$x
```

```
## [1]  1.7547348 -0.3676785 -1.5529115         NA  3.0000000
```

```
df[, "x", drop = FALSE]
```

```
##            x
## 1  1.7547348
## 2 -0.3676785
## 3 -1.5529115
```

Subscripting Data Frames

```
df[df$y > 10, ]
```

```
##               x         y
## NA           NA        NA
## 2    -0.3676785 108.34508
## 3    -1.5529115  85.43826
## NA.1         NA        NA
```

```
df[!is.na(df$y) & df$y > 10, ]
```

```
##            x         y
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
```

```
subset(df, y > 10)
```

```
##            x         y
## 2 -0.3676785 108.34508
## 3 -1.5529115  85.43826
```

Subscripting Data Frames

```
subset(df, y > 10, select = x)
```

```
##             x
## 2 -0.3676785
## 3 -1.5529115
```

```
subset(df, y > 10, select = 1)
```

```
##             x
## 2 -0.3676785
## 3 -1.5529115
```

Order a data frame

```
df[order(df$x), ]
```

```
##             x         y
## 3 -1.5529115  85.43826
## 2 -0.3676785 108.34508
## 1  1.7547348        NA
## 5  3.0000000   3.00000
## 4         NA        NA
```

Your Turn

1. Compute a variable `hypertension` that equals 1 if `DIAS` > 120 **and** `SYS` > 80
2. Compute a variable `Hypotension` that equals 1 if `DIAS` < 100 **and** `SYS` < 65
3. Create a variable that equals 1 if the patient's `CITY` is in New York state (NY)
4. Create a data set that contains the patients that have hypotension or hypertension don't live in New York state

Table of Contents

Data Aggregation

- For simple tabulation and cross-tabulation, the `table`, `ftable` and `xtabs` functions are available
- For more complex tasks, the available functions can be classified into two groups
  - Functions that operate on arrays and/or lists (e.g., `*apply`, `sweep`)
  - Functions oriented towards data frames (e.g., `aggregate`, `by`)

The `table` function

```
data(iris)

head(iris)

##   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
```

The table function

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
table(iris$Species, iris$Petal.Length > 6)
```

```
##
##               FALSE TRUE
##   setosa         50    0
##   versicolor     50    0
##   virginica      41    9
```

```
as.data.frame(table(iris$Species, iris$Petal.Length > 6))
```

```
##         Var1  Var2 Freq
## 1     setosa FALSE   50
## 2 versicolor FALSE   50
## 3  virginica FALSE   41
## 4     setosa  TRUE    0
## 5 versicolor  TRUE    0
```

The `table` function

```
table(iris$Species, iris$Petal.Length > 6, iris$Sepal.Width > 3.5)

## , ,  = FALSE
##
##
##               FALSE TRUE
##   setosa        34    0
##   versicolor    50    0
##   virginica     41    6
##
## , ,  = TRUE
##
##
##               FALSE TRUE
##   setosa        16    0
##   versicolor     0    0
##   virginica      0    3
```

The table function

```
as.data.frame(table(iris$Species, iris$Petal.Length > 6, iris$Sepal.Width

##          Var1  Var2  Var3 Freq
## 1      setosa FALSE FALSE   34
## 2  versicolor FALSE FALSE   50
## 3   virginica FALSE FALSE   41
## 4      setosa  TRUE FALSE    0
## 5  versicolor  TRUE FALSE    0
## 6   virginica  TRUE FALSE    6
## 7      setosa FALSE  TRUE   16
## 8  versicolor FALSE  TRUE    0
## 9   virginica FALSE  TRUE    0
## 10     setosa  TRUE  TRUE    0
## 11 versicolor  TRUE  TRUE    0
## 12  virginica  TRUE  TRUE    3
```

The table function

addmargins

```
tt <- table(iris$Species, iris$Petal.Length > 6)

addmargins(tt, margin = 1)

##
##              FALSE TRUE
##   setosa        50    0
##   versicolor    50    0
##   virginica     41    9
##   Sum          141    9

addmargins(tt, margin = c(1, 2))

##
##              FALSE TRUE Sum
##   setosa        50    0  50
##   versicolor    50    0  50
##   virginica     41    9  50
##   Sum          141    9 150
```

The table function

prop.table

```
prop.table(tt, margin = 1)

##
##                FALSE TRUE
##   setosa        1.00 0.00
##   versicolor    1.00 0.00
##   virginica     0.82 0.18

prop.table(tt, margin = 2)

##
##                    FALSE      TRUE
##   setosa     0.3546099 0.0000000
##   versicolor 0.3546099 0.0000000
##   virginica  0.2907801 1.0000000
```

The ftable function

The ftable function creates **flat** tables

```
ftable(iris$Species, iris$Petal.Length > 6, iris$Sepal.Width > 3.5)

##                      FALSE TRUE
##
##  setosa     FALSE       34   16
##             TRUE         0    0
##  versicolor FALSE       50    0
##             TRUE         0    0
##  virginica  FALSE       41    0
##             TRUE         6    3
```

The xtabs function

The `xtabs` function produces similar results as the `table` function but uses the `formula` interface

```
iris$sepal_width <- factor(as.integer(iris$Sepal.Width > 3.5),
                           levels = c(0, 1),
                           labels = c("<= 3.5", "> 3.5"))

with(iris, table(Species, sepal_width))

##              sepal_width
## Species        <= 3.5 > 3.5
##   setosa           34    16
##   versicolor       50     0
##   virginica        47     3

xtabs(~ Species + sepal_width, iris)

##              sepal_width
## Species        <= 3.5 > 3.5
##   setosa           34    16
##   versicolor       50     0
##   virginica        47     3
```

Missing values with table

The useNA and exclude Arguments

```
iris$sepal_widthNA <- iris$sepal_width
iris$sepal_widthNA[seq(1, 150, 25)] <- NA

with(iris, table(Species, sepal_widthNA))


##              sepal_widthNA
## Species        <= 3.5 > 3.5
##   setosa           32    16
##   versicolor       48     0
##   virginica        45     3

with(iris, table(Species, sepal_widthNA, useNA = "ifany"))


##              sepal_widthNA
## Species        <= 3.5 > 3.5 <NA>
##   setosa           32    16    2
##   versicolor       48     0    2
##   virginica        45     3    2

## If there is NAs, they will be included
```

Missing values with `table`

The useNA and exclude Arguments

```
with(iris, table(Species, sepal_widthNA, useNA = "always"))

##             sepal_widthNA
## Species       <= 3.5 > 3.5 <NA>
##   setosa          32    16    2
##   versicolor      48     0    2
##   virginica       45     3    2
##   <NA>             0     0    0

with(iris, table(Species, sepal_width, useNA = "always"))

##             sepal_width
## Species       <= 3.5 > 3.5 <NA>
##   setosa          34    16    0
##   versicolor      50     0    0
##   virginica       47     3    0
##   <NA>             0     0    0

## An NA column is always included, even if there is no missing values
```

The useNA argument is specific to `table`

Missing values with table

The useNA and exclude Arguments

The exclude argument

- By default, exclude = c(NA, NaN)
- E.g., exclude = NULL to include missing values

```
with(iris, table(Species, sepal_widthNA, exclude = NULL))

##             sepal_widthNA
## Species       <= 3.5 > 3.5 <NA>
##   setosa          32    16    2
##   versicolor      48     0    2
##   virginica       45     3    2
##   <NA>             0     0    0

## equivalent to useNA = "always"
```

Missing values with `table`

The `useNA` and `exclude` Arguments

```
with(iris, table(Species, sepal_widthNA, exclude = "setosa"))

##             sepal_widthNA
## Species      <= 3.5 > 3.5
##   versicolor     48     0
##   virginica      45     3

with(iris, table(Species, sepal_widthNA, exclude = "setosa", useNA = "ifar

##             sepal_widthNA
## Species      <= 3.5 > 3.5 <NA>
##   versicolor     48     0    2
##   virginica      45     3    2
```

- `exclude = NULL` is equivalent to `useNA="always"`
- `exclude = "somethingElse"` only exclude level `"somethingElse"` from the factor

Road Map for Aggregation

Three things to consider

1. How are the groups that divide the data defined?
2. What is the nature of the data to be operated on?
3. What is the desired end result

Groups Defined as Lists Elements

`sapply` or `lapply` are the appropriate functions

- `lapply` always returns a list
- `sapply` tries to “simplify” the output

Groups Defined as Lists Elements

sapply or lapply are the appropriate functions

- lapply always returns a list
- sapply tries to “simplify” the output

```
myList <- list()
for (i in 1:4) {
    myList[[i]] <- rnorm(n = 3 * i)
}
myList
```

```
## [[1]]
## [1]  0.3429579 -0.3193258  0.7808710
##
## [[2]]
## [1]  1.16866312  0.01419804  0.45813283 -0.43180622  0.34224696 -1.30745260
##
## [[3]]
## [1]  1.4005004 -1.7575754 -0.2415508  1.0928182 -1.1926425  1.8645074
## [7] -0.3128976  0.8755070 -1.9690762
##
## [[4]]
##  [1]  1.1415283 -1.1269112 -2.2914106  0.9855559 -1.4959317  2.2773178
##  [7]  0.5160894 -0.5481570 -0.6098171 -1.4302086  1.2207910 -1.7708338
```

Groups Defined as Lists Elements

Both for `lapply` and `sapply`, the first argument is a list, the second argument is a function
Third, fourth, . . . arguments are further arguments for the function that is applied

```
lapply(myList, length)

## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 12
```

```
sapply(myList, length)

## [1]  3  6  9 12
```

Groups Defined as Lists Elements

```
myList[[2]][c(3, 5)] <- NA
sapply(myList, mean)

## [1]  0.26816768          NA -0.02671217 -0.26099896
```

```
sapply(myList, mean, na.rm = TRUE)

## [1]  0.26816768 -0.13909942 -0.02671217 -0.26099896
```

```
sapply(myList, quantile, probs = c(0.25, 0.75), na.rm = TRUE)

##          [,1]       [,2]      [,3]      [,4]
## 25% 0.0118160 -0.6507178 -1.192642 -1.446639
## 75% 0.5619144  0.3028143  1.092818  1.024549
```

Groups Defined as Lists Elements

```
## A user defined function
lapply(myList, function(x) {
    data.frame(
        Mean = mean(x, na.rm = TRUE),
        SD = sd(x, na.rm = TRUE),
        Min = min(x, na.rm = TRUE),
        Max = max(x, na.rm = TRUE))
})
```

```
## [[1]]
##        Mean        SD        Min      Max
## 1 0.2681677 0.5538984 -0.3193258 0.780871
##
## [[2]]
##         Mean       SD       Min      Max
## 1 -0.1390994 1.030286 -1.307453 1.168663
##
## [[3]]
##          Mean       SD       Min      Max
## 1 -0.02671217 1.411432 -1.969076 1.864507
##
## [[4]]
##        Mean       SD       Min      Max
## 1 -0.260999 1.446369 -2.291411 2.277318
```

Groups Defined as Lists Elements

```
## A user defined function
sapply(myList, function(x) {
    data.frame(
        Mean = mean(x, na.rm = TRUE),
        SD = sd(x, na.rm = TRUE),
        Min = min(x, na.rm = TRUE),
        Max = max(x, na.rm = TRUE))
})

##      [,1]       [,2]       [,3]        [,4]
## Mean 0.2681677  -0.1390994 -0.02671217 -0.260999
## SD   0.5538984  1.030286   1.411432    1.446369
## Min  -0.3193258 -1.307453  -1.969076   -2.291411
## Max  0.780871   1.168663   1.864507    2.277318
```

Groups Defined as Lists Elements

```
mySummary <- function(x, na.rm = FALSE) {
    data.frame(
        Mean = mean(x, na.rm = na.rm),
        SD = sd(x, na.rm = na.rm),
        Min = min(x, na.rm = na.rm),
        Max = max(x, na.rm = na.rm))
}

sapply(myList, mySummary, na.rm = TRUE)

##      [,1]       [,2]       [,3]        [,4]
## Mean 0.2681677  -0.1390994 -0.02671217 -0.260999
## SD   0.5538984  1.030286   1.411432    1.446369
## Min  -0.3193258 -1.307453  -1.969076   -2.291411
## Max  0.780871   1.168663   1.864507    2.277318
```

Groups Defined as Lists Elements

`sapply` or `lapply` can be used as alternative to loops. This way you don’t have to take care too much of the form of the output

```r
## check type 1 error of the t-test
check_level <- function(i, n = 100) {
    a <- rnorm(n)
    b <- rnorm(n)
    tt <- t.test(a, b)
    tt$p.value < 0.05
}
```

```r
nsimul <- 1000
res <- sapply(1:nsimul, check_level, n = 100)
sum(res) / nsimul

## [1] 0.047
```

Groups Defined as Lists Elements

For this kind of simple repetitive tasks, the `replicate` function can also be used

```r
res2 <- replicate(nsimul, t.test(rnorm(10), rnorm(10))$p.value < 0.05)
sum(res2) / nsimul

## [1] 0.044
```

- First argument is the number of replication
- Second argument is an *expression*, i.e., a piece of R language and not a function

Groups Defined by Rows or Columns of a Matrix/Array

In this case, the `apply` function is the logical choice.
The `apply` function requires three arguments

- the array/matrix on which to operate
- An index telling `apply` which dimension to operate on (1 on rows; 2 on columns, `c(1, 2)` on both
- The function to use
- Optionally further arguments to be used by the function that we want to apply

```
apply(iris[, 1:4], 2, mean)

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Groups Defined by Rows or Columns of a Matrix/Array

```
apply(iris[, 1:4], 2, mySummary)

## $Sepal.Length
##        Mean        SD Min Max
## 1 5.843333 0.8280661 4.3 7.9
##
## $Sepal.Width
##        Mean        SD Min Max
## 1 3.057333 0.4358663   2 4.4
##
## $Petal.Length
##    Mean       SD Min Max
## 1 3.758 1.765298   1 6.9
##
## $Petal.Width
##        Mean        SD Min Max
## 1 1.199333 0.7622377 0.1 2.5
```

Groups Defined by Rows or Columns of a Matrix/Array

rowSums, colSums, rowMeans, colMeans

These are specialised functions that are potentially way faster than apply (which is a general function)

```
colMeans(iris[, 1:4], na.rm = TRUE)

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

```
colSums(iris[, 1:4] > 2, na.rm = TRUE)

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##          150          149          100           23
```

Groups Based on One or More Grouping Variables

A very common operation
A lot of choice in base R + a couple of additional packages that facilitates these operations

- `aggregate`
- `tapply`, `by`
- *split-apply-combine* strategy
  - `split`, `lapply`, `do.call`
  - **plyr**, **dplyr** package
  - ...

Groups Based on One or More Grouping Variables

aggregate

A natural choice for data summaries of several variables

- First argument: A formula
    - LHS: Variables to “summarise”
    - RHS: Grouping variables
- Second argument: A data frame
- Third argument: Function to apply
- ...; Further arguments for FUN

```
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species, iris, mean)

##      Species Sepal.Length Sepal.Width
## 1     setosa        5.006       3.428
## 2 versicolor        5.936       2.770
## 3  virginica        6.588       2.974
```

Groups Based on One or More Grouping Variables

aggregate

```
iris$Petal.Length.f <- factor(iris$Petal.Length > 4.8,
                              levels = c(FALSE, TRUE),
                              labels = c("Small petals", "Big petals"))
aggregate(cbind(Sepal.Length, Sepal.Width) ~ Species + Petal.Length.f,
          data = iris, FUN = mean)

##      Species Petal.Length.f Sepal.Length Sepal.Width
## 1     setosa   Small petals     5.006000    3.428000
## 2 versicolor   Small petals     5.889130    2.765217
## 3  virginica   Small petals     5.700000    2.766667
## 4 versicolor     Big petals     6.475000    2.825000
## 5  virginica     Big petals     6.644681    2.987234
```

Groups Based on One or More Grouping Variables

tapply

Returns an array with as many dimensions as there were vectors that defined the groups, but can only process a single vector

```
with(iris, tapply(X = Sepal.Length,
                  INDEX = list(Species, Petal.Length.f),
                  FUN = mean))

##              Small petals Big petals
## setosa            5.00600         NA
## versicolor        5.88913   6.475000
## virginica         5.70000   6.644681
```

Groups Based on One or More Grouping Variables

tapply

Also works if FUN does not return a scalar

```
with(iris, tapply(X = Sepal.Length,
                  INDEX = Species,
                  FUN = range))
```

```
## $setosa
## [1] 4.3 5.8
##
## $versicolor
## [1] 4.9 7.0
##
## $virginica
## [1] 4.9 7.9
```

Groups Based on One or More Grouping Variables

tapply

```
(tt <- with(iris, tapply(X = Sepal.Length,
                         INDEX = list(Species, Petal.Length.f),
                         FUN = range)))

##            Small petals Big petals
## setosa     Numeric,2    NULL
## versicolor Numeric,2    Numeric,2
## virginica  Numeric,2    Numeric,2
```

In this case, a matrix of lists is returned...

Groups Based on One or More Grouping Variables

tapply

But individual elements can still be accessed

```
tt[["setosa", "Small petals"]]

## [1] 4.3 5.8
```

```
str(tt)

## List of 6
##  $ : num [1:2] 4.3 5.8
##  $ : num [1:2] 4.9 7
##  $ : num [1:2] 4.9 6.2
##  $ : NULL
##  $ : num [1:2] 6 6.9
##  $ : num [1:2] 5.6 7.9
##  - attr(*, "dim")= int [1:2] 3 2
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:3] "setosa" "versicolor" "virginica"
##   ..$ : chr [1:2] "Small petals" "Big petals"
```

Groups Based on One or More Grouping Variables

by

- by is a version of `tapply` oriented towards data frames
- First argument is a data frame, others are as in `tapply`
- by returns a list

```
## Don't work, because iris is a data frame
by(iris[, 1:4], iris$Species, mean)

## Warning in mean.default(data[x, , drop = FALSE], ...):  argument is not numeric or logical:
returning NA
## Warning in mean.default(data[x, , drop = FALSE], ...):  argument is not numeric or logical:
returning NA
## Warning in mean.default(data[x, , drop = FALSE], ...):  argument is not numeric or logical:
returning NA

## iris$Species: setosa
## [1] NA
## --------------------------------------------------------
## iris$Species: versicolor
## [1] NA
## --------------------------------------------------------
## iris$Species: virginica
## [1] NA
```

Groups Based on One or More Grouping Variables

by

```
(ex_by <- by(iris[, 1:4], iris[, "Species"], colMeans))

## iris[, "Species"]: setosa
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
## ---------------------------------------------------------
## iris[, "Species"]: versicolor
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
## ---------------------------------------------------------
## iris[, "Species"]: virginica
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

Groups Based on One or More Grouping Variables

by

```
do.call(rbind, ex_by)

##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

- `do.call` takes a *list* of arguments (second argument)
- and prepares a call to a function (first argument), using the list elements as if they had been passed to the function as individual arguments

Groups Based on One or More Grouping Variables

Split-Apply-Combine

Term coined by Hadley Wickham (author of the **ggplot2**, **plyr**, **reshape**, **dplyr**, . . . , packages)

Split	Divide the problem into smaller pieces
Apply	Work on each pieces independently
Combine	Recombine the pieces

A common problem for both programming and data analysis; many implementations

- In base R: `split()`, `*apply()`, `do.call()`
- R-packages: **plyr**, **doBy**, **dplyr**, **data.table** (to some extent)

Split-Apply-Combine

Base R

- Split by species

```
s_iris <- split(iris, iris$Species)

## s_iris is a list with number of items
## equal to the number of levels of iris$Species
length(s_iris) == length(levels(iris$Species))

## [1] TRUE
```

- Apply a function to each item of the list

```
s_means <- lapply(s_iris, function(x) colMeans(x[1:4]))
s_means[[1]]

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
```

- Combine

```
(res <- do.call(rbind, s_means))

##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

Split-Apply-Combine

Base R

```
myLM <- function(x) {
    temp <- lm(Sepal.Length ~ Petal.Length, x)
    summary(temp)$coefficients
}

(res <- lapply(split(iris, iris$Species), myLM))
```

```
## $setosa
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)  4.2131682  0.4155888 10.137830 1.614927e-13
## Petal.Length 0.5422926  0.2823153  1.920876 6.069778e-02
##
## $versicolor
##              Estimate Std. Error  t value     Pr(>|t|)
## (Intercept)  2.407523  0.4462583 5.394909 2.075294e-06
## Petal.Length 0.828281  0.1041364 7.953806 2.586190e-10
##
## $virginica
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)  1.0596591 0.46676645  2.270213 2.772289e-02
## Petal.Length 0.9957386 0.08366764 11.901120 6.297786e-16
```

Split-Apply-Combine

The **plyr** Package

The *apply functions in base R implement the split-apply-combine strategy, but are inconsistent

- `apply()` input `arrays`; split by row and/or columns; output `array`
- `lapply()` input `list` or `vector`; output `list`
- `sapply()` input `list` or `vector`; simplify to vector
- `tapply` input `data.frame`; output depends
- `rapply()`, `vapply()`, `mapply()`

Split-Apply-Combine

The **plyr** Package

The *apply functions in base R implement the split-apply-combine strategy, but are inconsistent

- `apply()` input `arrays`; split by row and/or columns; output `array`
- `lapply()` input `list` or `vector`; output `list`
- `sapply()` input `list` or `vector`; simplify to vector
- `tapply` input `data.frame`; output depends
- `rapply()`, `vapply()`, `mapply()`

plyr brings some consistency: `**ply()`

first * Input type (a array, d data frame, `l` list)

second * Output type (a array, d data frame, `l` list, _ discard)

plyr

a*ply()

```
y <- a*ply(.data, .margins., .fun, ...)
```

.data	An array
.margins	Subscripts which the function gets applied over
.fun	Function to apply to each piece

Returns an array (*=a), a data.frame (*=d), a list (*=l)

plyr

l*ply()

```
y <- l*ply(.data, .fun, ...)
```

.data An list

.fun Function to apply to each item of the list

Returns an array (*=a), a data.frame (*=d), a list (*=l)

plyr

d*ply()

```
y <- d*ply(.data, .variables, .fun, ...)
```

.data	A data frame
.variables	Variables defining the groups
.fun	Function to apply to each group

Returns an array (*=a), a data.frame (*=d), a list (*=l)

plyr

d*ply()

```
(res <- ddply(iris, "Species", myLM))

##      Species  Estimate Std. Error   t value     Pr(>|t|)
## 1     setosa 4.2131682 0.41558877 10.137830 1.614927e-13
## 2     setosa 0.5422926 0.28231526  1.920876 6.069778e-02
## 3 versicolor 2.4075231 0.44625834  5.394909 2.075294e-06
## 4 versicolor 0.8282810 0.10413643  7.953806 2.586190e-10
## 5  virginica 1.0596591 0.46676645  2.270213 2.772289e-02
## 6  virginica 0.9957386 0.08366764 11.901120 6.297786e-16
```

plyr

d*ply()

```
(res <- ddply(iris, "Species", myLM))

##      Species  Estimate Std. Error   t value     Pr(>|t|)
## 1     setosa 4.2131682 0.41558877 10.137830 1.614927e-13
## 2     setosa 0.5422926 0.28231526  1.920876 6.069778e-02
## 3 versicolor 2.4075231 0.44625834  5.394909 2.075294e-06
## 4 versicolor 0.8282810 0.10413643  7.953806 2.586190e-10
## 5  virginica 1.0596591 0.46676645  2.270213 2.772289e-02
## 6  virginica 0.9957386 0.08366764 11.901120 6.297786e-16
```

The only problem with **plyr** is that it is sometimes slow

The **dplyr** package

Your Turn

Other example (see dplyr examples)?
Consider `WM_teams_2014.csv` data set that contains information on each player of the World Cup 2014, e.g., age, club, country, caps (number of plays for the national team)

1. Find the three oldest and youngest players for each country
2. Create a data set with the mean (with 95% confidence interval), median, 25% and 75% percentile of the players' age stratified on country and position
3. Create a data set containing the clubs, number of players in each club that participate in the world cup. The data set should be ordered from highest to lowest