Getting to use data in **R**

Alexandre Courtiol & Colin Vullioud

Leibniz Institute of Zoo and Wildlife Research

June 2018

Getting started with **R**

Handling data in **R**

There are many types of objects designed to store data in **R**.

We will focus on:

- vectors
- matrices (and arrays)
- data frames (and tibbles)
- lists

Note: if you master those, we are pretty much all set because most other objects derive from those!

Handling data in **R**

- vectors
  - a single row of data
  - all elements have the same type (e.g. `logical`, `integer`, `double`, `character`...)
- matrices (and arrays)
  - all rows & columns have same length
  - all rows & columns have the same type
- data frames (and tibbles)
  - all rows & columns have same length
  - each column can have its own type
- lists
  - each element can have its own length
  - each element can have its own type

Getting started with **R**

Vector

The vector is the simplest way to store data in **R**; it is a sequence of data elements of the same kind.

Example of a vector:

```
height_girls <- c(178, 175, 159, 164, 183, 192)
height_girls
```

```
## [1] 178 175 159 164 183 192
```

Getting started with R

Vector: general properties

They can be combined:

```
height_boys <- c(181, 189, 174, 177)
height <- c(height_boys, height_girls)
height
```

```
##  [1] 181 189 174 177 178 175 159 164 183 192
```

Vector: general properties

Subsets can be made (with indexes, booleans or names):

```
height_girls[2]  ## returns element 2
## [1] 175
height_girls[-3] ## remove element 3
## [1] 178 175 164 183 192
```

Vector: general properties

Subsets can be made (with indexes, booleans or names):

```
height_girls[2]  ## returns element 2
## [1] 175
height_girls[-3] ## remove element 3
## [1] 178 175 164 183 192
```

```
height_girls[c(1, 1, 2, 2, 2)]  ## open room for bootstraps and more
## [1] 178 178 175 175 175
```

Vector: general properties

Subsets can be made (with indexes, booleans or names):

```
height_girls[2]  ## returns element 2
## [1] 175
height_girls[-3] ## remove element 3
## [1] 178 175 164 183 192
```

```
height_girls[c(1, 1, 2, 2, 2)]  ## open room for bootstraps and more
## [1] 178 178 175 175 175
```

```
height_girls[height_girls > 168]
## [1] 178 175 183 192
height_girls[!(height_girls == min(height_girls))]
## [1] 178 175 164 183 192
height_girls[height_girls != min(height_girls)]
## [1] 178 175 164 183 192
```

Vector: general properties

The elements of a vector can be named and those names can be used for subsetting:

```
foo <- c(alex = 1, colin = 2)
foo

##  alex colin
##     1     2

foo["colin"]

## colin
##     2
```

Vector: general properties

The elements of a vector can be named and those names can be used for subsetting:

```
foo <- c(alex = 1, colin = 2)
foo
## alex colin
##    1     2
foo["colin"]
## colin
##     2
```

But names tend to be dropped in sometimes unexpected ways:

```
foo[1] + foo[2]
## alex
##    3
```

Vector: general properties

Vectors can sometimes have metadata attached to them:

```
foo <- c(1, 2, 3)
attr(foo, "whatever") <- "Learning to count"
attr(foo, "something else?") <- "nope"
```

```
foo
```

```
## [1] 1 2 3
## attr(,"whatever")
## [1] "Learning to count"
## attr(,"something else?")
## [1] "nope"
```

Vector: general properties

Vectors can sometimes have metadata attached to them:

```
foo <- c(1, 2, 3)
attr(foo, "whatever") <- "Learning to count"
attr(foo, "something else?") <- "nope"
```

```
foo
```

```
## [1] 1 2 3
## attr(,"whatever")
## [1] "Learning to count"
## attr(,"something else?")
## [1] "nope"
```

```
attr(foo, "whatever")
```

```
## [1] "Learning to count"
```

Vector: general properties

Vectors can sometimes have metadata attached to them:

```
foo <- c(1, 2, 3)
attr(foo, "whatever") <- "Learning to count"
attr(foo, "something else?") <- "nope"
```

```
foo
```

```
## [1] 1 2 3
## attr(,"whatever")
## [1] "Learning to count"
## attr(,"something else?")
## [1] "nope"
```

```
attr(foo, "whatever")
```

```
## [1] "Learning to count"
```

```
attributes(foo) ## this gives a list, see later!
```

```
## $whatever
## [1] "Learning to count"
##
## $`something else?`
## [1] "nope"
```

Note: this is useful to know for handling outputs in certain packages (e.g. `spaMM`).

Getting started with R

Vector: types

Types refer to the internal representation of the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
typeof(foo)
## [1] "logical"
```

Vector: types

Types refer to the internal representation of the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
typeof(foo)
## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))
## [1] 1 5 7 0
typeof(foo)
## [1] "integer"
```

Vector: types

Types refer to the internal representation of the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
typeof(foo)
## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))
## [1] 1 5 7 0
typeof(foo)
## [1] "integer"
```

- doubles

```
(foo <- c(1, 1.2, pi))
## [1] 1.000000 1.200000 3.141593
typeof(foo)
## [1] "double"
```

Vector: types

Types refer to the internal representation of the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
typeof(foo)
## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))
## [1] 1 5 7 0
typeof(foo)
## [1] "integer"
```

- doubles

```
(foo <- c(1, 1.2, pi))
## [1] 1.000000 1.200000 3.141593
typeof(foo)
## [1] "double"
```

- characters

```
(foo <- c("bla", "bli", "blo"))
## [1] "bla" "bli" "blo"
typeof(foo)
## [1] "character"
```

Vector: types

Types refer to the internal representation of the objects:

- logicals

```r
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
typeof(foo)
## [1] "logical"
```

- integers

```r
(foo <- c(1L, 5L, 7L, 0L))
## [1] 1 5 7 0
typeof(foo)
## [1] "integer"
```

- doubles

```r
(foo <- c(1, 1.2, pi))
## [1] 1.000000 1.200000 3.141593
typeof(foo)
## [1] "double"
```

- characters

```r
(foo <- c("bla", "bli", "blo"))
## [1] "bla" "bli" "blo"
typeof(foo)
## [1] "character"
```

Note: **R** detects automatically the type of input and creates the right type of vector for you!

Vector: classes

Classes refer to the how functions interact with the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))
## [1]  TRUE FALSE FALSE  TRUE
class(foo)
## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))
## [1] 1 5 7 0
class(foo)
## [1] "integer"
```

- numerics (from the type doubles)

```
(foo <- c(1, 1.2, pi))
## [1] 1.000000 1.200000 3.141593
class(foo)
## [1] "numeric"
```

- characters

```
(foo <- c("bla", "bli", "blo"))
## [1] "bla" "bli" "blo"
class(foo)
## [1] "character"
```

Note: many don't make the distinction between types and classes explicit but it helps to understand some weird behaviours of **R**.

Vector: classes

There are more classes than types:

- factors

```
(foo <- factor(c("bla", "bli", "blo")))

## [1] bla bli blo
## Levels: bla bli blo

class(foo)

## [1] "factor"

typeof(foo)

## [1] "integer"

levels(foo)

## [1] "bla" "bli" "blo"

levels(foo) <- c(levels(foo), "blu")  ## set extra level
table(foo)

## foo
## bla bli blo blu
##   1   1   1   0
```

Vector: classes

There are more classes than types:

- factors

```r
(foo <- factor(c("bla", "bli", "blo")))
```

```
## [1] bla bli blo
## Levels: bla bli blo
```

```r
class(foo)
```

```
## [1] "factor"
```

```r
typeof(foo)
```

```
## [1] "integer"
```

```r
levels(foo)
```

```
## [1] "bla" "bli" "blo"
```

```r
levels(foo) <- c(levels(foo), "blu")  ## set extra level
table(foo)
```

```
## foo
## bla bli blo blu
##   1   1   1   0
```

- dates

```r
(foo <- c(as.Date("2018/06/18"),
          as.Date("19-06-18", format = "%d-%m-%y")))
```

```
## [1] "2018-06-18" "2018-06-19"
```

```r
class(foo)
```

```
## [1] "Date"
```

```r
typeof(foo)
```

```
## [1] "double"
```

```r
foo + 50  ## you can do simple maths on dates!
```

```
## [1] "2018-08-07" "2018-08-08"
```

Vector: classes

There are more classes than types:

- factors

```
(foo <- factor(c("bla", "bli", "blo")))
```

```
## [1] bla bli blo
## Levels: bla bli blo
```

```
class(foo)
```

```
## [1] "factor"
```

```
typeof(foo)
```

```
## [1] "integer"
```

```
levels(foo)
```

```
## [1] "bla" "bli" "blo"
```

```
levels(foo) <- c(levels(foo), "blu")  ## set extra level
table(foo)
```

```
## foo
## bla bli blo blu
##   1   1   1   0
```

- dates

```
(foo <- c(as.Date("2018/06/18"),
          as.Date("19-06-18", format = "%d-%m-%y")))
```

```
## [1] "2018-06-18" "2018-06-19"
```

```
class(foo)
```

```
## [1] "Date"
```

```
typeof(foo)
```

```
## [1] "double"
```

```
foo + 50  ## you can do simple maths on dates!
```

```
## [1] "2018-08-07" "2018-08-08"
```

Note: factors are heavily used in the context of linear models!

Vector: classes

Vectors must contain elements of the same class (otherwise errors or automatic coercion may occur):

```
foo <- 1

bar <- "A"

foo_bar <- c(foo, bar)
foo_bar

## [1] "1" "A"
```

Vector: classes

Vectors must contain elements of the same class (otherwise errors or automatic coercion may occur):

```
foo <- 1

bar <- "A"

foo_bar <- c(foo, bar)
foo_bar
## [1] "1" "A"
```

```
foo + 1
## [1] 2
foo_bar[1] + 1
## Error in foo_bar[1] + 1:  non-numeric argument to binary operator
```

Vector: classes

Vectors must contain elements of the same class (otherwise errors or automatic coercion may occur):

```
foo <- 1

bar <- "A"

foo_bar <- c(foo, bar)
foo_bar
## [1] "1" "A"
```

```
foo + 1
## [1] 2
foo_bar[1] + 1
## Error in foo_bar[1] + 1:  non-numeric argument to binary operator
```

Challenges:

- find out why the previous call produces an error.
- try to check how the automatic coercion occurs by mixing different classes in different ways (logical, integers, numeric, characters, factors).
- find out which date is internally stored as 0?

Getting started with **R**

Factors

You can create them after in two steps:

```
sex <- c("girl","girl","girl","girl","girl", "girl",
"boy","boy","boy","boy")
class(sex)
```

```
## [1] "character"
```

```
sex <- factor(sex)
sex
```

```
##  [1] girl girl girl girl girl girl boy  boy  boy  boy
## Levels: boy girl
```

Factors

You can create them after in two steps:

```
sex <- c("girl","girl","girl","girl","girl", "girl",
"boy","boy","boy","boy")
class(sex)
```

```
## [1] "character"
```

```
sex <- factor(sex)
sex
```

```
##  [1] girl girl girl girl girl girl boy  boy  boy  boy
## Levels: boy girl
```

Better code:

```
sex <- factor(c(rep("girl", times = 6),
                rep("boy", times = 4)))
```

Even better code:

```
sex <- factor(c(rep("girl", times = length(height_girls)),
                rep("boy", times = length(height_boys))))
```

Note: more on programming style later!

Changing the order of levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A B B C
## Levels: C B A
```

Changing the order of levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A B B C
## Levels: C B A
```

You do:

```
my_factor2 <- factor(my_factor1, levels(my_factor1)[c(3, 2, 1)])
my_factor2
## [1] A A B B C
## Levels: C B A
```

Changing the order of levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A B B C
## Levels: C B A
```

You do:

```
my_factor2 <- factor(my_factor1, levels(my_factor1)[c(3, 2, 1)])
my_factor2
## [1] A A B B C
## Levels: C B A
```

Or if you only care of the first level:

```
my_factor3 <- relevel(my_factor1, ref = "C")
my_factor3
## [1] A A B B C
## Levels: C A B
```

Changing the order of levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A B B C
## Levels: C B A
```

You do:

```
my_factor2 <- factor(my_factor1, levels(my_factor1)[c(3, 2, 1)])
my_factor2
## [1] A A B B C
## Levels: C B A
```

Or if you only care of the first level:

```
my_factor3 <- relevel(my_factor1, ref = "C")
my_factor3
## [1] A A B B C
## Levels: C A B
```

Note: the order of levels influences the meaning of parameter estimates in linear models and some plotting functions (e.g. order in the legend of a ggplot) . . .

Changing the levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A A A D
## Levels: A D
```

Changing the levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A A A D
## Levels: A D
```

You do:

```
## Using base:
levels(my_factor1)
## [1] "A" "B" "C"
my_factor2 <- my_factor1
levels(my_factor2) <- c("A", "A", "D") ## in same order!
my_factor2
## [1] A A A A D
## Levels: A D
```

Changing the levels of a factor

You have:

```
my_factor1
## [1] A A B B C
## Levels: A B C
```

You want:

```
my_factor2
## [1] A A A A D
## Levels: A D
```

You do:

```
## Using base:
levels(my_factor1)
## [1] "A" "B" "C"
my_factor2 <- my_factor1
levels(my_factor2) <- c("A", "A", "D") ## in same order!
my_factor2
## [1] A A A A D
## Levels: A D
```

```
## Using dplyr:
library(dplyr)
my_factor2 <- recode(my_factor1, A = "A", B = "A", C = "D")
my_factor2
## [1] A A A A D
## Levels: A D
```

Some words about `dplyr` & co.

`dplyr` is part of the growing `tidyverse` world (`https://www.tidyverse.org/`) developped by RStudio:

Some words about `dplyr` & co.

`dplyr` is part of the growing `tidyverse` world (`https://www.tidyverse.org/`) developped by RStudio:

R core team

- build the core of **R**
- backward compatibility is the priority
- limited man power (20 selected volunteers)
- not commercial (but Microsoft may creep in?)

RStudio

- build RStudio & the `tidyverse` packages
- different philosophy: 1 function = 1 behaviour
- backward compatibility is not the priority
- 1 leader (Hadley Wickham) + ∼ 70 full time employees + tons of volunteers
- free + commercial

Some words about `dplyr` & co.

`dplyr` is part of the growing `tidyverse` world (`https://www.tidyverse.org/`) developped by RStudio:

R core team

- build the core of **R**
- backward compatibility is the priority
- limited man power (20 selected volunteers)
- not commercial (but Microsoft may creep in?)

RStudio

- build RStudio & the `tidyverse` packages
- different philosophy: 1 function = 1 behaviour
- backward compatibility is not the priority
- 1 leader (Hadley Wickham) + ∼ 70 full time employees + tons of volunteers
- free + commercial

Note 1: that has led to two quite distinct **R** dialects
Note 2: more and more users rely on `tidyverse`...
Note 3: we will see a bit of both dialects

Getting started with **R**

Some simple functions for vectors

```r
foo <- c("bla", "bla", "bli")
bar <- c(1, 1.2, pi, NA)
```

```r
any(is.na(foo))
## [1] FALSE
unique(foo)
## [1] "bla" "bli"
length(foo)
## [1] 3
str(foo)
##  chr [1:3] "bla" "bla" "bli"
summary(foo)
##    Length     Class      Mode
##         3 character character
```

Some simple functions for vectors

```r
foo <- c("bla", "bla", "bli")
bar <- c(1, 1.2, pi, NA)
```

```r
any(is.na(foo))
## [1] FALSE
unique(foo)
## [1] "bla" "bli"
length(foo)
## [1] 3
str(foo)
##  chr [1:3] "bla" "bla" "bli"
summary(foo)
##    Length     Class      Mode 
##         3 character character
```

```r
any(is.na(bar))
## [1] TRUE
unique(bar)
## [1] 1.000000 1.200000 3.141593       NA
length(bar)
## [1] 4
str(bar)
##  num [1:4] 1 1.2 3.14 NA
summary(bar)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   1.000   1.100   1.200   1.781   2.171   3.142       1
```

A more complex function: sapply()

sapply() is a function to apply a function on each element of a vector:

```
triple <- function(x) c(x, x, x)  ## let us create a sily function
triple("a")
```

```
## [1] "a" "a" "a"
```

```
sapply(X = bar, FUN = triple) ## here returns a matrix (automatic choice)
```

```
##      [,1] [,2]     [,3] [,4]
## [1,]    1  1.2 3.141593   NA
## [2,]    1  1.2 3.141593   NA
## [3,]    1  1.2 3.141593   NA
```

A more complex function: sapply()

`sapply()` is a function to apply a function on each element of a vector:

```
triple <- function(x) c(x, x, x)  ## let us create a sily function
triple("a")
```

```
## [1] "a" "a" "a"
```

```
sapply(X = bar, FUN = triple) ## here returns a matrix (automatic choice)
```

```
##      [,1] [,2]     [,3] [,4]
## [1,]    1  1.2 3.141593   NA
## [2,]    1  1.2 3.141593   NA
## [3,]    1  1.2 3.141593   NA
```

```
sapply(X = bar, FUN = triple, simplify = FALSE) ## same but always returns a list
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 1.2 1.2 1.2
##
## [[3]]
## [1] 3.141593 3.141593 3.141593
##
## [[4]]
## [1] NA NA NA
```

A more complex function: `sapply()`

`sapply()` is a function to apply a function on each element of a vector:

```
triple <- function(x) c(x, x, x)  ## let us create a sily function
triple("a")
## [1] "a" "a" "a"
```

```
sapply(X = bar, FUN = triple) ## here returns a matrix (automatic choice)
##      [,1] [,2]     [,3] [,4]
## [1,]    1  1.2 3.141593   NA
## [2,]    1  1.2 3.141593   NA
## [3,]    1  1.2 3.141593   NA
```

```
sapply(X = bar, FUN = triple, simplify = FALSE) ## same but always returns a list
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 1.2 1.2 1.2
##
## [[3]]
## [1] 3.141593 3.141593 3.141593
##
## [[4]]
## [1] NA NA NA
```

Note: this is useful when the function cannot work on vector and when the return is more than one element. For example, the input could be a vector of file names and the output one dataset per file!
Challenge: can you think of an alternative to do that without using `sapply()`?

The purrr alternative to sapply(): map()

```r
library(purrr)
map(.x = bar, .f = triple) ## always returns a list
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 1.2 1.2 1.2
##
## [[3]]
## [1] 3.141593 3.141593 3.141593
##
## [[4]]
## [1] NA NA NA
```

The purrr alternative to sapply(): map()

```
library(purrr)
map(.x = bar, .f = triple) ## always returns a list
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 1.2 1.2 1.2
##
## [[3]]
## [1] 3.141593 3.141593 3.141593
##
## [[4]]
## [1] NA NA NA
```

```
map_dfc(.x = bar, .f = triple) ## always returns a tibble binding columns
```

```
## # A tibble: 3 x 4
##      V1    V2    V3    V4
##   <dbl> <dbl> <dbl> <dbl>
## 1     1   1.2  3.14    NA
## 2     1   1.2  3.14    NA
## 3     1   1.2  3.14    NA
```

Getting started with **R**

Matrices & arrays

The matrices and arrays are direct extentions of vectors when there is more than one dimention (1 or 2 dimensions for matrices, any for arrays).

Example of a matrix:

```
my_matrix <- matrix(1:12, ncol = 4, nrow = 3)
my_matrix
```

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

```
class(my_matrix)
```

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

```
typeof(my_matrix) ## behind the curtain, matrices are stored as vectors!
```

```
## [1] "integer"
```

Matrices & arrays

The matrices and arrays are direct extentions of vectors when there is more than one dimention (1 or 2 dimensions for matrices, any for arrays).

Example of a matrix:

```
my_matrix <- matrix(1:12, ncol = 4, nrow = 3)
my_matrix
```

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

```
class(my_matrix)
```

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

```
typeof(my_matrix) ## behind the curtain, matrices are stored as vectors!
```

```
## [1] "integer"
```

Note 1: since there are kind of vectors, same restriction: all elements must have the same class!
Note 2: useful for bulding the input of some statistical tests (e.g. chi-square), for linear algebra (e.g. computation behind linear models), for handling GIS information & for understanding data frames.

Getting started with R

Matrices: general properties

They can be combined:

```
(my_2nd_matrix <- matrix(13:18, ncol = 2, nrow = 3))
```

```
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
```

```
(my_3rd_matrix <- matrix(1:4, nrow = 1))
```

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

```
cbind(my_matrix, my_2nd_matrix)  ## bind columns
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    4    7   10   13   16
## [2,]    2    5    8   11   14   17
## [3,]    3    6    9   12   15   18
```

```
rbind(my_matrix, my_3rd_matrix)  ## bind rows
```

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

Vector: general properties

Subsets can be made (with indexes, booleans or names):

```
my_matrix[2, ]
## [1]  2  5  8 11
my_matrix[, 1]
## [1] 1 2 3
my_matrix[3, , drop = FALSE]  ## to keep a matrix
##      [,1] [,2] [,3] [,4]
## [1,]    3    6    9   12
my_matrix[2, 1]
## [1] 2
my_matrix[c(1:2), c(1:2)]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
```

Vector: general properties

Subsets can be made (with indexes, booleans or names):

```
my_matrix[2, ]
## [1]  2  5  8 11
my_matrix[, 1]
## [1] 1 2 3
my_matrix[3, , drop = FALSE]  ## to keep a matrix
##      [,1] [,2] [,3] [,4]
## [1,]    3    6    9   12
my_matrix[2, 1]
## [1] 2
my_matrix[c(1:2), c(1:2)]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
```

```
colnames(my_matrix) <- c("A", "B", "C", "D")
rownames(my_matrix) <- c("a", "b", "c")
my_matrix
##   A B C  D
## a 1 4 7 10
## b 2 5 8 11
## c 3 6 9 12
my_matrix["b", ]
##  A  B  C  D
##  2  5  8 11
```

Getting started with **R**

Some simple functions for matrices

Dimensions:

```
dim(my_matrix)
## [1] 3 4
ncol(my_matrix)
## [1] 4
nrow(my_matrix)
## [1] 3
length(my_matrix)
## [1] 12
```

Names:

```
colnames(my_matrix)
## [1] "A" "B" "C" "D"
rownames(my_matrix)
## [1] "a" "b" "c"
```

Linear algebra:

```
t(my_matrix)  ## transpose
##    a  b  c
## A  1  2  3
## B  4  5  6
## C  7  8  9
## D 10 11 12
my_matrix %*% c(1:4) ## matrix multiplication
##   [,1]
## a   70
## b   80
## c   90
diag(my_matrix) ## extract diagonal
## [1] 1 5 9
```

A more complex function: apply()

apply() is a function to apply a function on each row or column of a matrix:

```
apply(X = my_matrix, MARGIN = 1, FUN = mean)  ## row means
```

```
##   a   b   c
## 5.5 6.5 7.5
```

```
apply(X = my_matrix, MARGIN = 2, FUN = sd)  ## column SDs
```

```
## A B C D
## 1 1 1 1
```

Note: `tidyverse` alternatives require to turn the matrix into a data frame, so we keep this for later.

Arrays?

Arrays are very similar to matrices but allow for more dimensions:

```
foo <- array(1:8, dim = c(2, 2, 2))
foo
```

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

```
foo[1, 2, 2]
```

```
## [1] 7
```

```
apply(X = foo, MARGIN = 3, FUN = sum)
```

```
## [1] 10 26
```

Note: only useful in some very specific situations.

Getting started with **R**

Getting started with **R**

Data frames

Data frames allow the organisation of entities as a matrix-like structure whose columns have the same length:

```
dataframe.ht <- data.frame(Height = height, Sex = sex)
dataframe.ht
```

```
##    Height  Sex
## 1     181 girl
## 2     189 girl
## 3     174 girl
## 4     177 girl
## 5     178 girl
## 6     175 girl
## 7     159  boy
## 8     164  boy
## 9     183  boy
## 10    192  boy
```

Data frames

It is good practice to always check their structure:

```
str(dataframe.ht)

## 'data.frame': 10 obs. of  2 variables:
##  $ Height: num  181 189 174 177 178 175 159 164 183 192
##  $ Sex   : Factor w/ 2 levels "boy","girl": 2 2 2 2 2 2 1 1 1 1
```

Data frames

You access the columns by means of the extractor $

```
height
##  [1] 181 189 174 177 178 175 159 164 183 192
rm(list = c("height", "sex")) # removing original vectors
height
## Error in eval(expr, envir, enclos):  object 'height' not found
dataframe.ht$Height #Or: with(data = dataframe.ht, Height)
##  [1] 181 189 174 177 178 175 159 164 183 192
```

⇒ What is the average height?

Data frames

Some functions can take a data frame as an input:

```
summary(dataframe.ht)
```

```
##      Height         Sex   
##  Min.   :159.0   boy :4  
##  1st Qu.:174.2   girl:6  
##  Median :177.5           
##  Mean   :177.2           
##  3rd Qu.:182.5           
##  Max.   :192.0           
```

Note: this will be the case of a lot of functions performing statistical tests!

Data frames

How to compute the average height per sex?

- simple

```r
mean(dataframe.ht$Height[dataframe.ht$Sex == "boy"])
```

```
## [1] 174.5
```

- more elegant

```r
tapply(X = dataframe.ht$Height, INDEX = dataframe.ht$Sex,
       FUN = mean)
```

```
##   boy  girl
## 174.5 179.0
```

```r
# Or: with(data = dataframe.ht, tapply(X = Height,
#   INDEX = Sex, FUN = mean))
```

- even more elegant but dangerous

```r
library(dplyr)
dataframe.ht %>% group_by(Sex) %>% summarize(mean = mean(Height))  ## be aware of the rounding
```

```
## # A tibble: 2 x 2
##   Sex    mean
##   <fct> <dbl>
## 1 boy    174.
## 2 girl   179
```

Data frames

They can also be indexed:

```
dataframe.ht[1, ]
```

```
##   Height  Sex
## 1    181 girl
```

```
dataframe.ht[, 1]  # Or: dataframe.ht[, "Sex"]
```

```
##  [1] 181 189 174 177 178 175 159 164 183 192
```

Data frames

They can be edited:

```
dataframe.ht[1, 1]

## [1] 181

dataframe.ht[1, 1] <- 171.3
dataframe.ht[1, 1]

## [1] 171.3

dataframe.ht$linenumber <- 1:nrow(dataframe.ht)  # add column
ncol(dataframe.ht) # try dim()

## [1] 3

dataframe.ht$linenumber <- NULL  # remove column
ncol(dataframe.ht)

## [1] 2
```

Getting started with R

dplyr

- `dplyr` is a useful package for data manipulation

dplyr

- `dplyr` is a useful package for data manipulation

- `dplyr` is a grammar of data manipulation: one verb = one operation

dplyr

- `dplyr` is a useful package for data manipulation
- `dplyr` is a grammar of data manipulation: one verb = one operation
- operations can be chained with the pipe operator %>%

dplyr

- `dplyr` is a useful package for data manipulation
- `dplyr` is a grammar of data manipulation: one verb = one operation
- operations can be chained with the pipe operator %>%
- the pipe operator %>% takes the output from one function as input of another function.

Data frames

Useful `dplyr` verbs

add column with `mutate()`

```
dataframe.ht <- dataframe.ht %>% mutate(ID = 1:nrow(dataframe.ht))
head(dataframe.ht, n= 3)
```

```
##   Height  Sex ID
## 1  171.3 girl  1
## 2  189.0 girl  2
## 3  174.0 girl  3
```

Data frames

Useful `dplyr` verbs

add column with `mutate()`

```
dataframe.ht <- dataframe.ht %>% mutate(ID = 1:nrow(dataframe.ht))
head(dataframe.ht, n= 3)
```

```
##   Height  Sex ID
## 1  171.3 girl  1
## 2  189.0 girl  2
## 3  174.0 girl  3
```

select columns with `select()`

```
dataframe.ht.sex <- dataframe.ht %>% select(Sex)
head(dataframe.ht.sex, n= 3)
```

```
##    Sex
## 1 girl
## 2 girl
## 3 girl
```

Data frames

Useful `dplyr` verbs

add column with `mutate()`

```
dataframe.ht <- dataframe.ht %>% mutate(ID = 1:nrow(dataframe.ht))
head(dataframe.ht, n= 3)
```

```
##   Height  Sex ID
## 1  171.3 girl  1
## 2  189.0 girl  2
## 3  174.0 girl  3
```

select columns with `select()`

```
dataframe.ht.sex <- dataframe.ht %>% select(Sex)
head(dataframe.ht.sex, n= 3)
```

```
##    Sex
## 1 girl
## 2 girl
## 3 girl
```

select rows with `filter()`

```
dataframe.ht.female <- dataframe.ht %>% filter(Sex == "girl")
head(dataframe.ht.female, n= 3)
```

```
##   Height  Sex ID
## 1  171.3 girl  1
## 2  189.0 girl  2
## 3  174.0 girl  3
```

mutate_if()

you want to change all numeric variables into character variables

you have:

```
## 'data.frame': 10 obs. of  6 variables:
##  $ Height  : num  171 189 174 177 178 ...
##  $ Sex     : chr  "girl" "girl" "girl" "girl" ...
##  $ ID      : int  1 2 3 4 5 6 7 8 9 10
##  $ mean_H  : num  177 177 177 177 177 ...
##  $ median_H: num  176 176 176 176 176 ...
##  $ n       : int  6 6 6 6 6 6 4 4 4 4
```

you want

```
## 'data.frame': 10 obs. of  6 variables:
##  $ Height  : chr  "171.3" "189" "174" "177" ...
##  $ Sex     : chr  "girl" "girl" "girl" "girl" ...
##  $ ID      : chr  "1" "2" "3" "4" ...
##  $ mean_H  : chr  "177.38" "177.38" "177.38" "177.38" ...
##  $ median_H: chr  "176" "176" "176" "176" ...
##  $ n       : chr  "6" "6" "6" "6" ...
```

you do:

```
x_numeric <- x %>% mutate_if(is.numeric, ~ as.character(.))
```

group_by()

- `group_by()` allow you to perform operation on gouped data.

group_by()

- `group_by()` allow you to perform operation on gouped data.

- it is mostly used with `summarize()` `->` one value per group

group_by()

- `group_by()` allow you to perform operation on gouped data.
- it is mostly used with `summarize()` -> one value per group
- or with `mutate()` -> one value per observation

group_by() with summarize()

You want the mean height of males and females, the median height and the number in each group:

you do:

```
x <- dataframe.ht %>%
  group_by(Sex) %>%
  summarize(mean_H = mean(Height, na.rm = T),
            median_H = median(Height, na.rm = T),
            n = n())
```

you get:

```
as.data.frame(x)

##    Sex   mean_H median_H n
## 1  boy 174.5000    173.5 4
## 2 girl 177.3833    176.0 6
```

group_by() with mutate()

You want the mean height of males and females, the median height and the number in each group but get the value for each individual

you do:

```
x <- dataframe.ht %>%
  group_by(Sex) %>%
  mutate(mean_H = mean(Height, na.rm = T),
            median_H = median(Height, na.rm = T),
            n = n())
```

you get:

```
as.data.frame(x)
```

```
##    Height  Sex ID   mean_H median_H n
## 1   171.3 girl  1 177.3833    176.0 6
## 2   189.0 girl  2 177.3833    176.0 6
## 3   174.0 girl  3 177.3833    176.0 6
## 4   177.0 girl  4 177.3833    176.0 6
## 5   178.0 girl  5 177.3833    176.0 6
## 6   175.0 girl  6 177.3833    176.0 6
## 7   159.0  boy  7 174.5000    173.5 4
## 8   164.0  boy  8 174.5000    173.5 4
## 9   183.0  boy  9 174.5000    173.5 4
## 10  192.0  boy 10 174.5000    173.5 4
```

joining data frame

you have df1:

```
my_df1
```

```
##        ID      age
## 1    ID-1 12.49418
## 2    ID-2 15.73457
## 3    ID-3 11.65749
## 4    ID-4 21.38112
## 5    ID-5 16.31803
## 6    ID-6 11.71813
## 11 ID-11 21.04712
```

you have df2:

```
my_df2
```

```
##       ID school grade origin
## 1   ID-4 Youhou 76.42 French
## 2   ID-5 bababa 71.88  Swiss
## 3   ID-1 genius 78.38 French
## 4  ID-12 Youhou 75.64 German
## 5   ID-7 bababa 61.49 German
## 6   ID-3 genius 20.21 French
## 7   ID-8 Youhou 72.40 German
## 8   ID-6 bababa 58.88 German
## 9   ID-2 genius 56.88  Swiss
## 10 ID-10 Youhou 30.58 French
```

You want to merge the two data frames

joining data frame with base R

You can use `merge()`

```
my_df3 <- merge(my_df1, my_df2)
```

```
my_df3
```

```
##      ID      age school grade origin
## 1 ID-1 12.49418 genius 78.38 French
## 2 ID-2 15.73457 genius 56.88  Swiss
## 3 ID-3 11.65749 genius 20.21 French
## 4 ID-4 21.38112 Youhou 76.42 French
## 5 ID-5 16.31803 bababa 71.88  Swiss
## 6 ID-6 11.71813 bababa 58.88 German
```

joining data frame with dpylr join()

or use `inner_join()`

```
library(dplyr)
my_df3 <- inner_join(my_df1, my_df2)
        ## Joining, by = "ID"
```

```
  my_df3
##      ID      age school grade origin
## 1 ID-1 12.49418 genius 78.38 French
## 2 ID-2 15.73457 genius 56.88  Swiss
## 3 ID-3 11.65749 genius 20.21 French
## 4 ID-4 21.38112 Youhou 76.42 French
## 5 ID-5 16.31803 bababa 71.88  Swiss
## 6 ID-6 11.71813 bababa 58.88 German
```

joining data frame with left_join()

`left_join()` or `right_join()` keep all the rows of the data frame on the left (or right)

adds NA when no data are present

```r
library(dplyr)
my_df3 <- left_join(my_df1, my_df2)
         ## Joining, by = "ID"
```

```r
my_df3
```

```
##       ID      age school grade origin
## 1  ID-1 12.49418 genius 78.38 French
## 2  ID-2 15.73457 genius 56.88  Swiss
## 3  ID-3 11.65749 genius 20.21 French
## 4  ID-4 21.38112 Youhou 76.42 French
## 5  ID-5 16.31803 bababa 71.88  Swiss
## 6  ID-6 11.71813 bababa 58.88 German
## 7 ID-11 21.04712   <NA>    NA   <NA>
```

joining data frame with full_join()

`full_join()` keep all the rows of the two data frame

adds NA when no data are present

```
library(dplyr)
my_df3 <- full_join(my_df1, my_df2)
## Joining, by = "ID"
```

```
my_df3
##        ID      age school grade origin
## 1    ID-1 12.49418 genius 78.38 French
## 2    ID-2 15.73457 genius 56.88  Swiss
## 3    ID-3 11.65749 genius 20.21 French
## 4    ID-4 21.38112 Youhou 76.42 French
## 5    ID-5 16.31803 bababa 71.88  Swiss
## 6    ID-6 11.71813 bababa 58.88 German
## 7   ID-11 21.04712   <NA>    NA   <NA>
## 8   ID-12       NA Youhou 75.64 German
## 9    ID-7       NA bababa 61.49 German
## 10   ID-8       NA Youhou 72.40 German
## 11  ID-10       NA Youhou 30.58 French
```

Getting started with **R**

reshaping data frame

- one row = one observation, one column = one variable

reshaping data frame

- one row = one observation, one column = one variable

- `gather()` turns wide data into long

reshaping data frame

- one row = one observation, one column = one variable
- `gather()` turns wide data into long
- `spread()` turns long data into wide

reshaping data frame

you have wide data:

```
head(my_df1)
```

```
##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3
```

```
dim(my_df1)
```

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

you want long data:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3
```

```
dim(my_df2)
```

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

reshaping data frame

you have wide data:

```
head(my_df1)
```

```
##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3
```

```
dim(my_df1)
```

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

you want long data:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3
```

```
dim(my_df2)
```

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

you do:

```
my_df2 <- my_df1 %>% gather("Age", "Height", -Sex, -ID) %>% arrange(ID, Age)
```

reshaping data frame

you have wide data:

```
head(my_df1)
```

```
##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3
```

```
dim(my_df1)
```

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

you want long data:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3
```

```
dim(my_df2)
```

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

you do:

```
my_df2 <- my_df1 %>% gather("Age", "Height", -Sex, -ID) %>% arrange(ID, Age)
```

or:

```
my_df2 <- my_df1 %>% gather("Age", "Height", 3:ncol(my_df1)) %>% arrange(ID, Age)
```

reshaping data frame

The reverse is done with `spread()`

you have wide data:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3
```

```
dim(my_df2)
```

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

you want long data:

```
head(my_df1)
```

```
##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3
```

```
dim(my_df1)
```

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

reshaping data frame

The reverse is done with `spread()`

you have wide data:

```
head(my_df2)

##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3

dim(my_df2)

## [1] 4 4
```

you want long data:

```
head(my_df1)

##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3

dim(my_df1)

## [1] 1 6
```

you do:

```
my_df2 %>% spread(-Sex, -ID)

##   ID  Sex age1  age2  age3  age4
## 1  1 girl 71.3 146.3 161.3 171.3
```

some other useful functions

`unite()` merges 2 columns of a data frame

```
my_df3 <- my_df2 %>% unite(New_col, ID, Sex)
head(my_df3)
```

```
##   New_col  Age Height
## 1  1_girl age1   71.3
## 2  1_girl age2  146.3
## 3  1_girl age3  161.3
## 4  1_girl age4  171.3
```

some other useful functions

unite() merges 2 columns of a data frame

```
my_df3 <- my_df2 %>% unite(New_col, ID, Sex)
head(my_df3)
```

```
##   New_col  Age Height
## 1  1_girl age1   71.3
## 2  1_girl age2  146.3
## 3  1_girl age3  161.3
## 4  1_girl age4  171.3
```

separate() separate 2 columns of a data frame

```
my_df3 %>% separate(New_col, c("ID", "Sex"))
```

```
##   ID  Sex  Age Height
## 1  1 girl age1   71.3
## 2  1 girl age2  146.3
## 3  1 girl age3  161.3
## 4  1 girl age4  171.3
```

cheating data frame

plenty of informative cheatsheets on: `https://www.rstudio.com/resources/cheatsheets/`

Getting started with **R**

Lists

Lists allow the organisation of any set of entities into a single R object:

```
list.ht <- list(girls = height_girls, boys = height_boys)
list.ht
```

```
## $girls
## [1] 178 175 159 164 183 192
##
## $boys
## [1] 181 189 174 177
```

Lists

Lists can also be indexed and their elements extracted:

```
list.ht$girls
## [1] 178 175 159 164 183 192
list.ht["boys"]  # still a list
## $boys
## [1] 181 189 174 177
list.ht[["boys"]]  # vector
## [1] 181 189 174 177
list.ht[[2]][3]
## [1] 174
```

Lists

Some functions can take a list as an input:

```
lapply(list.ht, FUN = mean)

## $girls
## [1] 175.1667
##
## $boys
## [1] 180.25
```

Summary

```
dataframe.ht
```

```
##    Height  Sex  ID
## 1   171.3 girl   1
## 2   189.0 girl   2
## 3   174.0 girl   3
## 4   177.0 girl   4
## 5   178.0 girl   5
## 6   175.0 girl   6
## 7   159.0  boy   7
## 8   164.0  boy   8
## 9   183.0  boy   9
## 10  192.0  boy  10
```

```
list.ht
```

```
## $girls
## [1] 178 175 159 164 183 192
##
## $boys
## [1] 181 189 174 177
```

Summary

- `data.frame`
  - All columns have same length
  - Each column can have its own class (e.g. `numeric`, `factor`, `character`)
- `list`
  - Each element can have its own length
  - Each element can have its own class (e.g. `numeric`, `factor`, `character`)

Getting started with R

Working directory

```
getwd()  # to change, use setwd()
## [1] "/Users/alex/Dropbox/Boulot/Mes_projets_de_recherche/R_packages/BeginR_project/BeginR/sources_vignettes/usingdata"
dir() # listing all files in the working directory
## [1] "usingdata.nav"      "usingdata.pdf"
## [3] "usingdata.pdf.asis" "usingdata.Rnw"
## [5] "usingdata.snm"      "usingdata.tex"
## [7] "usingdata.toc"      "usingdata.vrb"
dir(pattern = "*.csv")
## character(0)
```

Exporting and importing data in R

```
write.csv(dataframe.ht,
    file = "my.first.R.dataframe.csv", row.names = FALSE)

rm(list = ls())  # deleting everything in R

dataframe.ht <- read.csv("my.first.R.dataframe.csv")
```

R cannot read/write .xls files out of the box
Packages can do that but it is safer to use .csv files
Excel can read and write .csv files!

Challenge #2

Create a dataframe using your favorite spreadsheet software
and import it in R!