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 behave sometimes somewhat unexpectedly:

```
foo[1] + foo[2]

## alex
##    3
```

Vector: general properties

Vectors (as any other object) can have metadata called ‘attributes’ attached to them:

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

```
foo
```

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

Vector: general properties

Vectors (as any other object) can have metadata called ‘attributes’ attached to them:

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

```
foo

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

```
attr(x = foo, which = "whatever")

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

Vector: general properties

Vectors (as any other object) can have metadata called 'attributes' attached to them:

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

```r
foo
```

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

```r
attr(x = foo, which = "whatever")
```

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

```r
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(x = 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(x = foo)

## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))

## [1] 1 5 7 0

typeof(x = 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(x = foo)

## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))

## [1] 1 5 7 0

typeof(x = foo)

## [1] "integer"
```

- doubles

```
(foo <- c(1, 1.2, pi))

## [1] 1.000000 1.200000 3.141593

typeof(x = 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(x = foo)

## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))

## [1] 1 5 7 0

typeof(x = foo)

## [1] "integer"
```

- doubles

```
(foo <- c(1, 1.2, pi))

## [1] 1.000000 1.200000 3.141593

typeof(x = foo)

## [1] "double"
```

- characters

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

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

typeof(x = foo)

## [1] "character"
```

Vector: types

Types refer to the internal representation of the objects:

- logicals

```
(foo <- c(TRUE, FALSE, F, T))

## [1]  TRUE FALSE FALSE  TRUE

typeof(x = foo)

## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))

## [1] 1 5 7 0

typeof(x = foo)

## [1] "integer"
```

- doubles

```
(foo <- c(1, 1.2, pi))

## [1] 1.000000 1.200000 3.141593

typeof(x = foo)

## [1] "double"
```

- characters

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

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

typeof(x = foo)

## [1] "character"
```

Note: **R** detects automatically the type of input and creates the right type of vector for you! Challenge: compare `typeof()` with `mode()`.

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(x = foo)

## [1] "logical"
```

- integers

```
(foo <- c(1L, 5L, 7L, 0L))

## [1] 1 5 7 0

class(x = foo)

## [1] "integer"
```

- numerics (from the type doubles)

```
(foo <- c(1, 1.2, pi))

## [1] 1.000000 1.200000 3.141593

class(x = foo)

## [1] "numeric"
```

- characters

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

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

class(x = 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(x = foo)

## [1] "factor"

typeof(x = foo)

## [1] "integer"

levels(x = foo)

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

levels(x = foo) <- c(levels(x = 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

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

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

```
class(x = foo)
```

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

```
typeof(x = foo)
```

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

```
levels(x = foo)
```

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

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

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

- dates

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

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

```
class(x = foo)
```

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

```
typeof(x = foo)
```

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

```
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(x = foo)

## [1] "factor"

typeof(x = foo)

## [1] "integer"

levels(x = foo)

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

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

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

- dates

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

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

class(x = foo)

## [1] "Date"

typeof(x = 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?

Vector: classes

Some coercions are straightforward:

```
as.integer(x = 1.2)

## [1] 1

as.integer(x = 1.9)

## [1] 1

as.integer(x = -2.1)

## [1] -2
```

```
foo <- factor(x = 10:20)
foo

##  [1] 10 11 12 13 14 15 16 17 18 19 20
## Levels: 10 11 12 13 14 15 16 17 18 19 20

as.character(x = foo)

##  [1] "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "20"
```

Vector: classes

Some coercions are straightforward:

```
as.integer(x = 1.2)

## [1] 1

as.integer(x = 1.9)

## [1] 1

as.integer(x = -2.1)

## [1] -2
```

```
foo <- factor(x = 10:20)
foo

##  [1] 10 11 12 13 14 15 16 17 18 19 20
## Levels: 10 11 12 13 14 15 16 17 18 19 20

as.character(x = foo)

##  [1] "10" "11" "12" "13" "14" "15" "16" "17" "18" "19" "20"
```

But not all:

```
as.numeric(x = foo)

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

as.numeric(as.character(x = foo))

##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
foo <- as.Date(x = "20180618", format = "%Y%m%d")
as.integer(x = foo)

## [1] 17700

as.integer(x = gsub(pattern = "-", replacement = "", x = as.character(foo)))

## [1] 20180618
```

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(x = sex)

## [1] "character"
```

```
sex <- factor(x = 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(x = sex)
```

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

```
sex <- factor(x = sex)
sex
```

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

Better code:

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

Even better code:

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

Note: more on programming style later!

Combining vectors with different levels

We want to merge the two following vectors:

```
foo <- factor(c("a", "b"))
foo
```

```
## [1] a b
## Levels: a b
```

```
bar <- factor(c("b", "c"))
bar
```

```
## [1] b c
## Levels: b c
```

Combining vectors with different levels

We want to merge the two following vectors:

```
foo <- factor(c("a", "b"))
foo

## [1] a b
## Levels: a b
```

```
bar <- factor(c("b", "c"))
bar

## [1] b c
## Levels: b c
```

Problem:

```
foo_bar <- c(foo, bar)
foo_bar

## [1] 1 2 1 2

class(foo_bar)

## [1] "integer"
```

Combining vectors with different levels

We want to merge the two following vectors:

```
foo <- factor(c("a", "b"))
foo
```

```
## [1] a b
## Levels: a b
```

```
bar <- factor(c("b", "c"))
bar
```

```
## [1] b c
## Levels: b c
```

Problem:

```
foo_bar <- c(foo, bar)
foo_bar
```

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

```
class(foo_bar)
```

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

Solution:

```
foo_bar <- factor(c(as.character(foo), as.character(bar)))
foo_bar
```

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

```
class(foo_bar)
```

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

Dropping unused levels

By default **R** keeps unused levels:

```
foo <- factor(c("a", "a", "b", "c"))
foo

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

```
table(foo)

## foo
## a b c
## 2 1 1
```

```
bar <- foo[-4]
table(bar)

## bar
## a b c
## 2 1 0
```

Dropping unused levels

By default **R** keeps unused levels:

```
foo <- factor(c("a", "a", "b", "c"))
foo
```

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

```
table(foo)
```

```
## foo
## a b c
## 2 1 1
```

```
bar <- foo[-4]
table(bar)
```

```
## bar
## a b c
## 2 1 0
```

If you want to update the levels you need to use the function `droplevels`:

```
new_bar <- droplevels(bar)
table(new_bar)
```

```
## new_bar
## a b
## 2 1
```

Or use the argument `drop`:

```
bar <- foo[-4, drop = TRUE]
table(bar)
```

```
## bar
## a b
## 2 1
```

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(x = my_factor1, levels = 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(x = my_factor1, levels = levels(my_factor1)[c(3, 2, 1)])
my_factor2

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

Or if you only care about the first level:

```
my_factor3 <- relevel(x = 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(x = my_factor1, levels = levels(my_factor1)[c(3, 2, 1)])
my_factor2
```

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

Or if you only care about the first level:

```
my_factor3 <- relevel(x = 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(x = my_factor1)

## [1] "A" "B" "C"

my_factor2 <- my_factor1
levels(x = 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(x = my_factor1)
```

```
## [1] "A" "B" "C"
```

```
my_factor2 <- my_factor1
levels(x = 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(.x = my_factor1, A = "A", B = "A", C = "D")
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(x = my_factor1)
## [1] "A" "B" "C"
my_factor2 <- my_factor1
levels(x = 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(.x = my_factor1, A = "A", B = "A", C = "D")
my_factor2
## [1] A A A A D
## Levels: A D
```

Note: if you want more modern functions to manipulate factors, look at the package `forcats` from `tidyverse`.

Getting started with **R**

Some simple functions for vectors

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

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

Some simple functions for vectors

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

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

```
any(is.na(x = bar))
## [1] TRUE
unique(x = bar)
## [1] 1.000000 1.200000 3.141593       NA
length(x = bar)
## [1] 4
str(object = bar)
##  num [1:4] 1 1.2 3.14 NA
summary(object = 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(x = "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(x = "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(x = "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 1: 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!

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(x = "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 1: 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!
Note 2: if you want more modern functions more consistent than the *apply() ones, look at the package purrr from tidyverse.

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(x = "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 1: 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!
Note 2: if you want more modern functions more consistent than the *apply() ones, look at the package purrr from tidyverse.
Challenge: can you think of an alternative to do that without using sapply()?

Getting started with **R**

Matrices & arrays

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

Example of a matrix:

```
my_matrix <- matrix(data = 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(x = my_matrix)
```

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

```
typeof(x = 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 dimension (1 or 2 dimensions for matrices, any for arrays).

Example of a matrix:

```
my_matrix <- matrix(data = 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(x = my_matrix)
## [1] "matrix"
typeof(x = my_matrix) ## behind the curtain, matrices are stored as vectors!
## [1] "integer"
```

Note 1: since there are a kind of vectors, the same restrictions apply: 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(data = 13:18, ncol = 2, nrow = 3))
```

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

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

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

Matrices: general properties

They can be combined:

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

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

(my_3rd_matrix <- matrix(data = 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
```

Matrices: general properties

They can be combined:

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

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

```
(my_3rd_matrix <- matrix(data = 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
```

Matrices: 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
```

Matrices: 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(x = my_matrix) <- c("A", "B", "C", "D")
rownames(x = 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(x = my_matrix)
## [1] 3 4
ncol(x = my_matrix)
## [1] 4
nrow(x = my_matrix)
## [1] 3
length(x = my_matrix)
## [1] 12
```

Names:

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

Linear algebra:

```
t(x = 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(x = 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
```

Arrays?

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

```
foo <- array(data = 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**

Lists

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

Example of a list:

```
list_height <- list(height_girls, height_boys)
list_height
## [[1]]
## [1] 178 175 159 164 183 192
##
## [[2]]
## [1] 181 189 174 177
class(x = list_height)
## [1] "list"
typeof(x = list_height)
## [1] "list"
```

Note 1: list elements can be anything!

Note 2: lists are very important because no function can output more than one object!

Getting started with **R**

Lists: general properties

They can be combined:

```
list_full <- c(list_height, list(my_matrix))
list_full
```

```
## [[1]]
## [1] 178 175 159 164 183 192
##
## [[2]]
## [1] 181 189 174 177
##
## [[3]]
##   A B C  D
## a 1 4 7 10
## b 2 5 8 11
## c 3 6 9 12
```

Lists: general properties

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

```
list_height <- list(girls = height_girls, boys = height_boys)  ## create a list with names
list_height
```

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

```
list_height$girls
```

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

```
list_height["boys"] ## still a list
```

```
## $boys
## [1] 181 189 174 177
```

```
list_height[["boys"]] ## vector
```

```
## [1] 181 189 174 177
```

```
list_height[[2]][3]
```

```
## [1] 174
```

Getting started with **R**

Some simple functions for lists

```
length(x = list_full) ## number of elements
## [1] 3
```

```
str(object = list_full)  ## this function is really useful!
## List of 3
##  $ : num [1:6] 178 175 159 164 183 192
##  $ : num [1:4] 181 189 174 177
##  $ : int [1:3, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "a" "b" "c"
##   .. ..$ : chr [1:4] "A" "B" "C" "D"
```

Challenge: run the examples from `lm()` and explore the list `lm.D9`.

A more complex function: `lapply()`

`lapply()` is a function to apply a function on each element of a list:

```
lapply(X = list_full, FUN = mean)
```

```
## [[1]]
## [1] 175.1667
##
## [[2]]
## [1] 180.25
##
## [[3]]
## [1] 6.5
```

Getting started with **R**

Data frames

Data frames allow the organisation of vectors of the same length as a matrix-like structure:

Example:

```
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
```

```
class(dataframe_ht)
```

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

```
typeof(dataframe_ht)
```

```
## [1] "list"
```

Data frames

Data frames allow the organisation of vectors of the same length as a matrix-like structure:

Example:

```
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
```

```
class(dataframe_ht)
```

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

```
typeof(dataframe_ht)
```

```
## [1] "list"
```

Note 1: this is the best choice of representation for datasets!

Note 2: it is safer to work on data frames than on floating vectors!

Getting started with **R**

Data frames: general properties

They borough from both matrices and lists:

As for matrices:

```
(dataframe_ht_double <- cbind(dataframe_ht, newcol = 1:10))
```

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

Data frames: general properties

They borough from both matrices and lists:

As for matrices:

```
(dataframe_ht_double <- cbind(dataframe_ht, newcol = 1:10))
```

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

```
dataframe_ht[, "Sex"]
```

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

```
dataframe_ht[2, 2]
```

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

Data frames: general properties

They borough from both matrices and lists:

As for matrices:

```
(dataframe_ht_double <- cbind(dataframe_ht, newcol = 1:10))
```

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

```
dataframe_ht[, "Sex"]
```

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

```
dataframe_ht[2, 2]
```

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

As for lists:

```
dataframe_ht$Height
```

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

```
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
```

Getting started with **R**

Data frames: challenge

The `iris` data set:

```
head(iris) ## this function displays the first 6 rows
```

```
##   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
```

Using the `iris` data frame, find out:

- what is the average sepal length across all flowers?
- what is the median sepal length across *Iris versicolor*?

Data frames: general properties

Data frames can easily be edited:

```
backup <- dataframe_ht[1, 1]
dataframe_ht[1, 1] <- 171.3
dataframe_ht[1, 1]
```

```
## [1] 171.3
```

```
dataframe_ht[1, 1] <- backup
dataframe_ht[1, 1]
```

```
## [1] 181
```

Data frames: general properties

Data frames can easily be edited:

```
backup <- dataframe_ht[1, 1]
dataframe_ht[1, 1] <- 171.3
dataframe_ht[1, 1]

## [1] 171.3

dataframe_ht[1, 1] <- backup
dataframe_ht[1, 1]

## [1] 181
```

```
dataframe_ht$linenumber <- 1:nrow(x = dataframe_ht)  # add column
head(x = dataframe_ht)

##   Height  Sex linenumber
## 1    181 girl          1
## 2    189 girl          2
## 3    174 girl          3
## 4    177 girl          4
## 5    178 girl          5
## 6    175 girl          6
```

Data frames: general properties

Data frames can easily be edited:

```
backup <- dataframe_ht[1, 1]
dataframe_ht[1, 1] <- 171.3
dataframe_ht[1, 1]

## [1] 171.3

dataframe_ht[1, 1] <- backup
dataframe_ht[1, 1]

## [1] 181
```

```
dataframe_ht$linenumber <- 1:nrow(x = dataframe_ht)  # add column
head(x = dataframe_ht)

##   Height  Sex linenumber
## 1    181 girl          1
## 2    189 girl          2
## 3    174 girl          3
## 4    177 girl          4
## 5    178 girl          5
## 6    175 girl          6
```

```
dataframe_ht$linenumber <- NULL  # remove column
head(x = dataframe_ht)

##   Height  Sex
## 1    181 girl
## 2    189 girl
## 3    174 girl
## 4    177 girl
## 5    178 girl
## 6    175 girl
```

Getting started with **R**

Some simple functions for data frames

```
head(x = iris)  ## try also tail()
```

```
##   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
```

```
summary(object = iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length
##  Min.   :4.300   Min.   :2.000   Min.   :1.000
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
##  Median :5.800   Median :3.000   Median :4.350
##  Mean   :5.843   Mean   :3.057   Mean   :3.758
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
##  Max.   :7.900   Max.   :4.400   Max.   :6.900
##   Petal.Width          Species
##  Min.   :0.100   setosa    :50
##  1st Qu.:0.300   versicolor:50
##  Median :1.300   virginica :50
##  Mean   :1.199
##  3rd Qu.:1.800
##  Max.   :2.500
```

```
dim(x = iris)
```

```
## [1] 150   5
```

```
ncol(x = iris)
```

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

```
nrow(x = iris)
```

```
## [1] 150
```

```
length(x = iris) ## not as in matrix!!
```

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

```
rownames(x = iris)[1:10]
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

```
colnames(x = iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
## [4] "Petal.Width"  "Species"
```

A more complex function: `tapply()`

`tapply()` is a function to apply a function on subsets of a given column from the data frame:

```
tapply(X = iris$Sepal.Length, INDEX = iris$Species, FUN = mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

A more complex function: `tapply()`

`tapply()` is a function to apply a function on subsets of a given column from the data frame:

```
tapply(X = iris$Sepal.Length, INDEX = iris$Species, FUN = mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Or similarly:

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

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

A more complex function: `tapply()`

`tapply()` is a function to apply a function on subsets of a given column from the data frame:

```
tapply(X = iris$Sepal.Length, INDEX = iris$Species, FUN = mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Or similarly:

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

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

Or similarly:

```
by(data = iris, INDICES = iris$Species, FUN = function(x) mean(x$Sepal.Length))
```

```
## iris$Species: setosa
## [1] 5.006
## ---------------------------------------------
## iris$Species: versicolor
## [1] 5.936
## ---------------------------------------------
## iris$Species: virginica
## [1] 6.588
```

Note: `by()` is more powerful but more complex than `tapply()`.

The dyplr alternative to tapply()

The same operation in dyplr looks very different:

```
iris %>%
  group_by(Species) %>%
  summarize(mean_sepal_length = mean(Sepal.Length),
            mean_sepal_width = mean(Sepal.Width)) %>%
  as.data.frame()  ## optional but otherwise tibble and not data frame
```

```
##      Species mean_sepal_length mean_sepal_width
## 1     setosa             5.006            3.428
## 2 versicolor             5.936            2.770
## 3  virginica             6.588            2.974
```

Note: this replaces two tapply() calls and remains easy to read.

Getting started with R

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** and the original **R** GUI
- maintain CRAN
- backward compatibility is the priority
- limited man power (20 selected volunteers)
- not commercial (but Microsoft may creep in?)

RStudio

- build RStudio IDE, `tidyverse` and more
- `tidyverse` philosophy: 1 function = 1 action
- 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

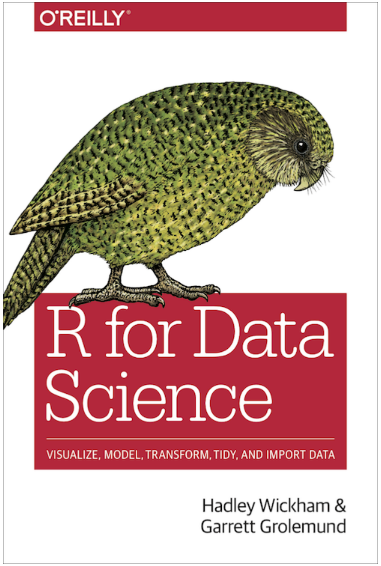
- build the core of **R** and the original **R** GUI
- maintain CRAN
- backward compatibility is the priority
- limited man power (20 selected volunteers)
- not commercial (but Microsoft may creep in?)

RStudio

- build RStudio IDE, `tidyverse` and more
- `tidyverse` philosophy: 1 function = 1 action
- backward compatibility is not the priority
- 1 leader (Hadley Wickham) + $\sim$ 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

Some words about `dplyr` & co.



dplyr

- in dplyr one verb = one action = one function (tidyverse philosophy)
- operations can be chained with the pipe operator %>% (from package magrittr), which considers the output from one function as the input of the next function

dplyr

- in `dplyr` one verb = one action = one function (`tidyverse` philosophy)
- operations can be chained with the pipe operator %>% (from package `magrittr`), which considers the output from one function as the input of the next function

Pros

- clear code
- consistent
- powerful
- fast
- many tutorials

dplyr

- in dplyr one verb = one action = one function (tidyverse philosophy)
- operations can be chained with the pipe operator %>% (from package magrittr), which considers the output from one function as the input of the next function

Pros

- clear code
- consistent
- powerful
- fast
- many tutorials

Cons

- different & redundant
- buggy (but less & less so)
- poor traditional documentation
- lead to bad habits (e.g. arguments not named, help not looked at)
- broaden the gap between users and programmers

dplyr verbs

Useful dplyr functions:

- add column with mutate()

```
dataframe_ht <- dataframe_ht %>% mutate(ID = 1:nrow(dataframe_ht))
head(x = dataframe_ht, n = 3)
```

```
##   Height  Sex ID
## 1    181 girl  1
## 2    189 girl  2
## 3    174 girl  3
```

dplyr verbs

Useful dplyr functions:

- add column with mutate()

```
dataframe_ht <- dataframe_ht %>% mutate(ID = 1:nrow(dataframe_ht))
head(x = dataframe_ht, n = 3)

##   Height  Sex ID
## 1    181 girl  1
## 2    189 girl  2
## 3    174 girl  3
```

- change column with transmute()

```
dataframe_ht2 <- dataframe_ht %>% transmute(double_height = 2*height)
head(x = dataframe_ht2, n = 3)

##   double_height
## 1           362
## 2           378
## 3           348
```

dplyr verbs

Useful dplyr functions:

- select columns with select()

```
dataframe_ht_sex <- dataframe_ht %>% select(Sex)
head(x = dataframe_ht_sex, n = 3)
```

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

dplyr verbs

Useful dplyr functions:

- select columns with select()

```
dataframe_ht_sex <- dataframe_ht %>% select(Sex)
head(x = 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    181 girl  1
## 2    189 girl  2
## 3    174 girl  3
```

dplyr verbs

Useful dplyr functions:

- select columns with select()

```
dataframe_ht_sex <- dataframe_ht %>% select(Sex)
head(x = 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    181 girl  1
## 2    189 girl  2
## 3    174 girl  3
```

- sort rows with arrange()

```
dataframe_ht_sorted <- dataframe_ht %>% arrange(Height)  ## add arrange(desc(Height)) for the other direction
head(dataframe_ht_sorted, n = 3)
```

```
##   Height  Sex ID
## 1    159  boy  7
## 2    164  boy  8
## 3    174 girl  3
```

Around dplyr verbs

These main `dplr` functions have derivatives and some of them can be useful:
e.g. `mutate_if` performs mutation if a condition is fulfilled, which could be useful for example if you want to change all numeric variables into character variables:

you have:

```
## 'data.frame': 150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ..
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ..
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1
```

you want:

```
## 'data.frame': 150 obs. of  5 variables:
##  $ Sepal.Length: chr  "5.1" "4.9" "4.7" "4.6" ...
##  $ Sepal.Width : chr  "3.5" "3" "3.2" "3.1" ...
##  $ Petal.Length: chr  "1.4" "1.4" "1.3" "1.5" ...
##  $ Petal.Width : chr  "0.2" "0.2" "0.2" "0.2" ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1
```

you do:

```
iris_numeric <- iris %>%
  mutate_if(is.numeric, ~ as.character(.))
```

group_by()

The `group_by()` function allows you to perform operation on gouped data.

group_by()

The group_by() function allows you to perform operation on gouped data.

It is very powerful when combined to:

- summarize() $\rightarrow$ one value per group

group_by()

The group_by() function allows you to perform operation on gouped data.

It is very powerful when combined to:

- summarize() → one value per group
- mutate() or transmute() → one value per observation

group_by()

The group_by() function allows you to perform operation on gouped data.

It is very powerful when combined to:

- summarize() $\rightarrow$ one value per group
- mutate() or transmute() $\rightarrow$ one value per observation
- slice() $\rightarrow$ select some rows for each "group"

group_by() with summarize()

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

```
dataframe_ht %>%
  group_by(Sex) %>%
  summarize(mean_height = mean(Height),
            median_height = median(Height),
            n = n()) %>%
  as.data.frame()
```

```
##    Sex mean_height median_height n
## 1  boy       174.5         173.5 4
## 2 girl       179.0         177.5 6
```

group_by() with mutate()

Same as before, but we want to repeat the value for each individual:

```
dataframe_ht %>%
  group_by(Sex) %>%
  mutate(mean_height = mean(Height),
            median_height = median(Height),
            n = n()) %>%
  as.data.frame()
```

```
##    Height  Sex ID mean_height median_height n
## 1     181 girl  1       179.0         177.5 6
## 2     189 girl  2       179.0         177.5 6
## 3     174 girl  3       179.0         177.5 6
## 4     177 girl  4       179.0         177.5 6
## 5     178 girl  5       179.0         177.5 6
## 6     175 girl  6       179.0         177.5 6
## 7     159  boy  7       174.5         173.5 4
## 8     164  boy  8       174.5         173.5 4
## 9     183  boy  9       174.5         173.5 4
## 10    192  boy 10       174.5         173.5 4
```

Note: many other functions than `n()` can be used, see `?summarise` !

group_by() with slice()

Example: you want the first two rows of each species of irices:

```
iris %>%
  group_by(Species) %>%
  slice(1:2) %>%
  as.data.frame()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          7.0         3.2          4.7         1.4
## 4          6.4         3.2          4.5         1.5
## 5          6.3         3.3          6.0         2.5
## 6          5.8         2.7          5.1         1.9
##      Species
## 1     setosa
## 2     setosa
## 3 versicolor
## 4 versicolor
## 5  virginica
## 6  virginica
```

Challenge

Use the dataset called `population_UK` and compute the total population size for:

- 1915
- 2015
- all years in the dataset
- all years between 1915 and 2015

Use the dataset called `deaths_UK` and figure out:

- which were the top 3 detailed causes of death before 1930 for each of the 21 broader categories
- the death toll for all individuals below 15 yrs for each year

Using dplyr to merge datasets

Data frame #1:

```
my_df1 <- iris %>%
  filter(Species == "setosa") %>%
  select(Sepal.Length, Petal.Length, Species) %>%
  slice(1:4)
my_df1[4, 1] <- NA
my_df1
```

```
##   Sepal.Length Petal.Length Species
## 1          5.1          1.4  setosa
## 2          4.9          1.4  setosa
## 3          4.7          1.3  setosa
## 4           NA          1.5  setosa
```

Data frame #2:

```
my_df2 <- iris %>%
  filter(Species == "virginica") %>%
  select(Sepal.Length, Petal.Width, Species) %>%
  slice(1:4)
my_df2[3, 2] <- NA
my_df2
```

```
##   Sepal.Length Petal.Width   Species
## 1          6.3         2.5 virginica
## 2          5.8         1.9 virginica
## 3          7.1          NA virginica
## 4          6.3         1.8 virginica
```

We will see how to merge these two data frames!

Using dplyr to merge datasets

There are several options but `full_join()` is the most effective one: it keeps all the rows of the two data frames and adds NA when no data are present!

```
my_df3 <- full_join(my_df1, my_df2)
## Joining, by = c("Sepal.Length", "Species")
my_df3
##   Sepal.Length Petal.Length   Species Petal.Width
## 1          5.1          1.4    setosa          NA
## 2          4.9          1.4    setosa          NA
## 3          4.7          1.3    setosa          NA
## 4           NA          1.5    setosa          NA
## 5          6.3           NA virginica         2.5
## 6          5.8           NA virginica         1.9
## 7          7.1           NA virginica          NA
## 8          6.3           NA virginica         1.8
```

Note: you can also do that without `dplyr` but the outcome is a bit more messy:

```
merge(my_df1, my_df2, all = TRUE)
```

Challenge

Use the datasets called population_UK and deaths_UK to compute yearly mortality rates for:

- all individuals
- individuals below 15 yrs only

Getting started with **R**

Reshaping data frame

For most data analyses, you need:

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

Unfortunatelly, it is often not the way people input data!

Reshaping data frame

For most data analyses, you need:

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

Unfortunatelly, it is often not the way people input data!

The `tidyverse` package `tidyr` offers solutions:

- `gather()` turns wide data into long
- `spread()` turns long data into wide

From wide to long

you have:

```
head(my_df1)
```

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

```
dim(my_df1)
```

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

you want:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

```
dim(my_df2)
```

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

From wide to long

you have:

```
head(my_df1)
```

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

```
dim(my_df1)
```

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

you want:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

```
dim(my_df2)
```

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

you do:

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

From wide to long

you have:

```
head(my_df1)
```

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

```
dim(my_df1)
```

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

you want:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

```
dim(my_df2)
```

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

you do:

```
library(tidyr)
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)
```

From long to wide

you have:

```
head(my_df2)
```

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

```
dim(my_df2)
```

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

you want:

```
head(my_df1)
```

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

```
dim(my_df1)
```

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

From long to wide

you have:

```
head(my_df2)
```

```
##   ID  Sex   Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

```
dim(my_df2)
```

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

you want:

```
head(my_df1)
```

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

```
dim(my_df1)
```

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

you do:

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

```
##   ID  Sex age1 age2 age3 age4
## 1  1 girl   81  156  171  181
```

Note: but why on Earth would you need that?!

Some other useful functions from `tidyr`

`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     81
## 2  1_girl age2    156
## 3  1_girl age3    171
## 4  1_girl age4    181
```

Some other useful functions from `tidyr`

`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     81
## 2  1_girl age2    156
## 3  1_girl age3    171
## 4  1_girl age4    181
```

`separate()` splits 2 columns of a data frame:

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

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

Some other useful functions from `tidyr`

`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     81
## 2  1_girl age2    156
## 3  1_girl age3    171
## 4  1_girl age4    181
```

`separate()` splits 2 columns of a data frame:

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

```
##   ID  Sex  Age Height
## 1  1 girl age1     81
## 2  1 girl age2    156
## 3  1 girl age3    171
## 4  1 girl age4    181
```

Note: the **R** base equivalent are `paste()` and `strsplit()` but they are a bit more tedious to use.

Getting started with **R**

Working directory

Before anything, you must know where you read & write on your hard drive!

```
getwd() ## get the working directory, to change it use setwd()
## [1] "/home/alex/Dropbox/Boulot/Mes_projets_de_recherche/R_packages/BeginR_project/BeginR/sources_vignettes/usingdata"
dir() ## list 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")  ## list all files with the extension csv
## character(0)
```

Note: you can also set this up with RStudio but it won't be saved unless you set up a project file.

Exporting and importing data in the **R** binary format

R can write and read binary formats that take by convention the extensions *.rda or *.RData.

Example:

```
my_iris <- iris
save(my_iris, file = "my_iris.rda") ## check the help for compression
```

Exporting and importing data in the **R** binary format

R can write and read binary formats that take by convention the extensions *.rda or *.RData.

Example:

```
my_iris <- iris
save(my_iris, file = "my_iris.rda") ## check the help for compression
```

```
rm(list = ls()) ## removes everything!
head(my_iris)
## Error in head(my_iris):  object 'my_iris' not found
```

Exporting and importing data in the **R** binary format

R can write and read binary formats that take by convention the extensions *.rda or *.RData.

Example:

```
my_iris <- iris
save(my_iris, file = "my_iris.rda") ## check the help for compression
```

```
rm(list = ls()) ## removes everything!
head(my_iris)
```

```
## Error in head(my_iris):  object 'my_iris' not found
```

```
load(file = "my_iris.rda")
head(my_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
```

Note: this is useful and best for **R** to **R** exchanges (but it is useless without **R**).

Exporting and importing data sets in plain text

- **R** cannot read/write .xls(x) files out of the box
- Several packages can do that, but it is safer to use .csv or .txt files
- Excel can read and write .csv & .txt files!

Writing a data set:

```
write.csv(my_iris, file = "my_iris.csv", row.names = FALSE)
```

Exporting and importing data sets in plain text

- **R** cannot read/write .xls(x) files out of the box
- Several packages can do that, but it is safer to use .csv or .txt files
- Excel can read and write .csv & .txt files!

Writing a data set:

```
write.csv(my_iris, file = "my_iris.csv", row.names = FALSE)
```

Reading a data set:

```
rm(my_iris) ## delete the object my_iris
my_iris <- read.csv("my_iris.csv")
head(my_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
```

Exporting and importing data sets in plain text

- **R** cannot read/write .xls(x) files out of the box
- Several packages can do that, but it is safer to use .csv or .txt files
- Excel can read and write .csv & .txt files!

Writing a data set:

```
write.csv(my_iris, file = "my_iris.csv", row.names = FALSE)
```

Reading a data set:

```
rm(my_iris) ## delete the object my_iris
my_iris <- read.csv("my_iris.csv")
head(my_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
```

Note 1: always check your file in a text editor before importing it or use RStudio "File/Import Datasets GUI".

Note 2: you will have often to change the arguments `sep` (and `dec` if you are german).

Note 3: setting `stringsAsFactors = FALSE` can avoid a lot of troubles!

Challenge

Create a data frame using your favorite spreadsheet software (or choose an existing one) and import it in **R**.