

# Getting to use data in R

Alexandre Courtiol & Colin Vulllioud

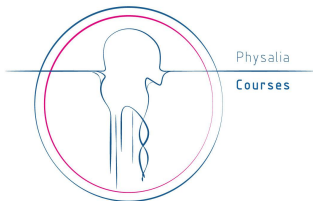
Leibniz Institute of Zoo and Wildlife Research

June 2018



**Leibniz Institute for Zoo  
and Wildlife Research**

IN THE FORSCHUNGSVERBUND BERLIN E.V.



# Getting started with R

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays
- 4 List
- 5 Data frames
- 6 Importing & exporting data
- 7 tidyverse

# 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

# 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!

Note: if you don't master at least vectors and data frames, you will never get very far using **R**.

# 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

- lists

- each element can have its own length
- each element can have its own type

- data frames (and tibbles)

- all rows & columns have same length
- each column can have its own type

# Getting started with R

- 1 Introduction
- 2 **Vectors**
- 3 Matrices and arrays
- 4 List
- 5 Data frames
- 6 Importing & exporting data
- 7 tidyverse

# Vector

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

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

## 1 Introduction

## 2 Vectors

- general properties
  - types & classes
  - factors
  - functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features



## 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)] ## useful 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)]  ## useful 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
foo[2] + foo[1]
## colin
##    3
```

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

Double squared brackets can also be used for subsetting of single elements; they drop names (and other attributes):

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

## Vector: general properties

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

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

## $names
## [1] "alex" "colin"
```



## Vector: general properties

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

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

## $names
## [1] "alex" "colin"
```

```
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(alex = 1, colin = 2)
attributes(foo)

## $names
## [1] "alex" "colin"
```

```
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:

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

## $names
## [1] "alex" "colin"
```

```
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"
```

```
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

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# 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** automatically detects 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(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 really help to understand many 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: coercion

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

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

## Vector: coercion

Vectors must contain elements of the same type (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: coercion

Vectors must contain elements of the same type (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.
- find out which date is internally stored as 0?
- try to check how the automatic coercion occurs by mixing different classes in different ways (logical, integers, numeric, characters, factors).



## Vector: coercion

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
```

# Vector: coercion

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: coercion

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
```

# Vector: coercion

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

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- **factors**
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# 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"
```

# 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
class(x = sex)
## [1] "factor"
```

# 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
class(x = sex)
## [1] "factor"
```

Better code:

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



# 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
class(x = sex)
## [1] "factor"
```

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(x = c("a", "b"))
foo
## [1] a b
## Levels: a b
```

```
bar <- factor(x = 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(x = c("a", "b"))
foo
## [1] a b
## Levels: a b
```

```
bar <- factor(x = c("b", "c"))
bar
## [1] b c
## Levels: b c
```

## Problem:

```
foo_bar <- c(foo, bar)
foo_bar
## [1] 1 2 1 2
class(x = foo_bar)
## [1] "integer"
```

# Combining vectors with different levels

We want to merge the two following vectors:

```
foo <- factor(x = c("a", "b"))
foo
## [1] a b
## Levels: a b
```

```
bar <- factor(x = c("b", "c"))
bar
## [1] b c
## Levels: b c
```

## Problem:

```
foo_bar <- c(foo, bar)
foo_bar
## [1] 1 2 1 2
class(x = foo_bar)
## [1] "integer"
```

## Solution:

```
foo_bar <- factor(x = c(as.character(x = foo), as.character(x = bar))
foo_bar
## [1] a b b c
## Levels: a b c
class(x = foo_bar)
## [1] "factor"
```

# Dropping unused levels

By default **R** keeps unused levels:

```
foo <- factor(x = 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(x = 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(x = 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:

```
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:

```
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
```

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

# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- **functions**

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# 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 silly 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 silly 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 silly 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 silly 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`.

## 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 silly 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`.

Challenge: can you think of an alternative to do that without using `sapply()`?

# Getting started with R

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays**
- 4 List
- 5 Data frames
- 6 Importing & exporting data
- 7 tidyverse

# Matrices & arrays

The matrices and arrays are direct extensions 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 extensions 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 building 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

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# 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
```

# Matrices: general properties

In the background, a matrix is a vector with dimensions defined as an attribute:

```
attributes(my_matrix)
## $dim
## [1] 3 4
##
## $dimnames
## $dimnames[[1]]
## [1] "a" "b" "c"
##
## $dimnames[[2]]
## [1] "A" "B" "C" "D"

str(my_matrix)
## 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"
```

## Matrices: general properties

In the background, a matrix is a vector with dimensions defined as an attribute:

```
attributes(my_matrix)
## $dim
## [1] 3 4
##
## $dimnames
## $dimnames[[1]]
## [1] "a" "b" "c"
##
## $dimnames[[2]]
## [1] "A" "B" "C" "D"

str(my_matrix)
## 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"
```

Thus, you can also subset a matrix considering it is a vector:

```
my_matrix[5] == my_matrix[2, 2]
## [1] TRUE
```



# Matrices: general properties

In the background, a matrix is a vector with dimensions defined as an attribute:

```
attributes(my_matrix)
## $dim
## [1] 3 4
##
## $dimnames
## $dimnames[[1]]
## [1] "a" "b" "c"
##
## $dimnames[[2]]
## [1] "A" "B" "C" "D"

str(my_matrix)
## 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"
```

Thus, you can also subset a matrix considering it is a vector:

```
my_matrix[5] == my_matrix[2, 2]
## [1] TRUE
```

Note: by default, a matrix is always filled column by column (top → bottom then left → right):

```
(my_matrix4 <- matrix(data = 1:4, ncol = 2, nrow = 2)) ## but you can use the option byrow = TRUE to do fill matrices row by row instead!
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# 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
```

## 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

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays
- 4 List**
- 5 Data frames
- 6 Importing & exporting data
- 7 tidyverse

# 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 necessary because no function can output more than one object in **R**!



# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

## 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
```

## 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
```

# 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
```

## 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

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- **functions**

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

## Some simple functions for lists

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

```
str(object = list_full) ## using str() on a list is really useful to understand the output of complex functions
## 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

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays
- 4 List
- 5 Data frames**
- 6 Importing & exporting data
- 7 tidyverse

# Data frames

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

## Example:

```
height <- c(172, 178, 182, 162, 175, 168)
sex     <- factor(c("female", "male", "male", "female", "male", "male"))

dataframe_ht <- data.frame(Height = height, Sex = sex)
dataframe_ht

##   Height   Sex
## 1    172 female
## 2    178  male
## 3    182  male
## 4    162 female
## 5    175  male
## 6    168  male

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:

```
height <- c(172, 178, 182, 162, 175, 168)
sex    <- factor(c("female", "male", "male", "female", "male", "male"))

dataframe_ht <- data.frame(Height = height, Sex = sex)
dataframe_ht

##   Height   Sex
## 1    172 female
## 2    178  male
## 3    182  male
## 4    162 female
## 5    175  male
## 6    168  male

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

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- **general properties**
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# Data frames: general properties

The usage borrows from both matrices and lists:

As for matrices:

```
(dataframe_ht2 <- cbind(dataframe_ht,  
                        newcol = 1:nrow(dataframe_ht)))
```

##	Height	Sex	newcol
## 1	172	female	1
## 2	178	male	2
## 3	182	male	3
## 4	162	female	4
## 5	175	male	5
## 6	168	male	6

# Data frames: general properties

The usage borrows from both matrices and lists:

As for matrices:

```
(dataframe_ht2 <- cbind(dataframe_ht,  
                        newcol = 1:nrow(dataframe_ht)))
```

```
##   Height   Sex newcol  
## 1    172 female     1  
## 2    178  male     2  
## 3    182  male     3  
## 4    162 female     4  
## 5    175  male     5  
## 6    168  male     6
```

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

```
## [1] female male  male  female male  male  
## Levels: female male
```

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

```
## [1] male  
## Levels: female male
```

# Data frames: general properties

The usage borrows from both matrices and lists:

As for matrices:

```
(dataframe_ht2 <- cbind(dataframe_ht,
                        newcol = 1:nrow(dataframe_ht)))

##   Height   Sex newcol
## 1    172 female     1
## 2    178  male     2
## 3    182  male     3
## 4    162 female     4
## 5    175  male     5
## 6    168  male     6
```

```
dataframe_ht[, "Sex"]
## [1] female male   male   female male   male
## Levels: female male

dataframe_ht[2, 2]
## [1] male
## Levels: female male
```

As for lists:

```
dataframe_ht$Height
## [1] 172 178 182 162 175 168

str(dataframe_ht)
## 'data.frame': 6 obs. of 2 variables:
## $ Height: num 172 178 182 162 175 168
## $ Sex : Factor w/ 2 levels "female","male": 1 2 2 1 2 2
```



# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- **challenge**
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

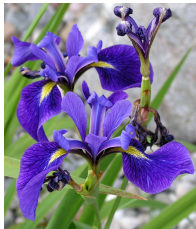
# Data frames: challenge

The iris data set ([https://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](https://en.wikipedia.org/wiki/Iris_flower_data_set)):



*Iris setosa*

©Miya.m



*Iris versicolor*

©D.G.E. Robertson



*Iris virginica*

©F. Mayfield

```
head(x = iris, n = 2L) ## this function displays by default the first 6 rows but here we display only the first 2
```

```
##   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
```

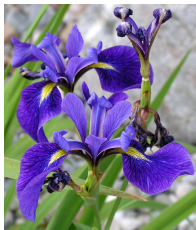
# Data frames: challenge

The iris data set ([https://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](https://en.wikipedia.org/wiki/Iris_flower_data_set)):



*Iris setosa*

©Miya.m



*Iris versicolor*

©D.G.E. Robertson



*Iris virginica*

©F. Mayfield

```
head(x = iris, n = 2L) ## this function displays by default the first 6 rows but here we display only the first 2
## 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
```

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] 172
```

# 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] 172
```

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

```
##   Height    Sex linenumber
## 1    172 female          1
## 2    178   male          2
## 3    182   male          3
## 4    162 female          4
## 5    175   male          5
## 6    168   male          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] 172
```

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

```
##   Height   Sex linenumber
## 1    172 female         1
## 2    178   male         2
## 3    182   male         3
## 4    162 female         4
## 5    175   male         5
## 6    168   male         6
```

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

```
##   Height   Sex
## 1    172 female
## 2    178   male
## 3    182   male
## 4    162 female
## 5    175   male
## 6    168   male
```

# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- **functions**

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

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

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

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

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

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

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

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

```
length(x = iris) ## as in list, 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" "Petal.Width" "
```



## 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()`.

# Getting started with R

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays
- 4 List
- 5 Data frames
- 6 Importing & exporting data**
- 7 tidyverse

## 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.pdf"      "usingdata.pdf.asis" "usingdata.Rnw"
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(x = my_iris)
## Error in head(x = 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(x = my_iris)

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

```
load(file = "my_iris.rda")
head(x = 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(x = 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(x = 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(file = "my_iris.csv") ## or read.table() with adequate options!
head(x = 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(x = 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(file = "my_iris.csv") ## or read.table() with adequate options!
head(x = 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**.

# Getting started with R

- 1 Introduction
- 2 Vectors
- 3 Matrices and arrays
- 4 List
- 5 Data frames
- 6 Importing & exporting data
- 7 tidyverse**

# Meeting the tidyverse world

Two different ways to solve the problem: *what is the average length and width for each iris species?*

Standard R approach:

```
tapply(X = iris$Sepal.Length, INDEX = iris$Species, FUN = mean)
##      setosa versicolor virginica
##      5.006      5.936      6.588

tapply(X = iris$Sepal.Width, INDEX = iris$Species, FUN = mean)
##      setosa versicolor virginica
##      3.428      2.770      2.974
```

# Meeting the tidyverse world

Two different ways to solve the problem: *what is the average length and width for each iris species?*

Standard R approach:

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

##      setosa versicolor virginica
##      5.006      5.936      6.588

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

##      setosa versicolor virginica
##      3.428      2.770      2.974
```

The same operation using the package dplyr from tidyverse:

```
library(tidyverse) ## or just load dplyr

iris %>%
  group_by(Species) %>%
  summarize(mean_sepal_length = mean(Sepal.Length), mean_sepal_width = mean(Sepal.Width))

## # A tibble: 3 x 3
##   Species    mean_sepal_length mean_sepal_width
##   <fct>          <dbl>          <dbl>
## 1 setosa              5.01              3.43
## 2 versicolor          5.94              2.77
## 3 virginica           6.59              2.97
```

# Meeting the tidyverse world

Two different ways to solve the problem: *what is the average length and width for each iris species?*

Standard R approach:

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

##      setosa versicolor  virginica
##      5.006      5.936      6.588

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

##      setosa versicolor  virginica
##      3.428      2.770      2.974
```

The same operation using the package dplyr from tidyverse:

```
library(tidyverse) ## or just load dplyr

iris %>%
  group_by(Species) %>%
  summarize(mean_sepal_length = mean(Sepal.Length), mean_sepal_width = mean(Sepal.Width))

## # A tibble: 3 x 3
##   Species    mean_sepal_length mean_sepal_width
##   <fct>          <dbl>          <dbl>
## 1 setosa              5.01              3.43
## 2 versicolor         5.94              2.77
## 3 virginica          6.59              2.97
```

Note: which one do you prefer?



# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

# Some words about the tidyverse

The tidyverse packages (<https://www.tidyverse.org/>) are developed by RStudio:



# Some words about the tidyverse

The tidyverse packages (<https://www.tidyverse.org/>) are developed by RStudio:



## R is from the R core team who

- 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?)

## tidyverse is from the RStudio people who

- build RStudio IDE, tidyverse and more
- tidyverse philosophy: 1 function = 1 action
- backward compatibility is not the priority
- ~ 80 employees + tons of volunteers
- free + commercial

# Some words about the tidyverse

The tidyverse packages (<https://www.tidyverse.org/>) are developed by RStudio:



## R is from the R core team who

- 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?)

## tidyverse is from the RStudio people who

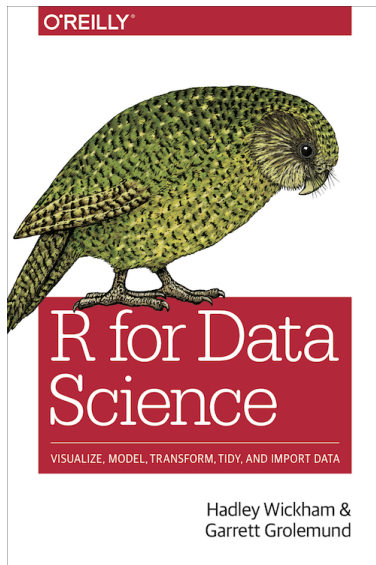
- build RStudio IDE, tidyverse and more
- tidyverse philosophy: 1 function = 1 action
- backward compatibility is not the priority
- ~ 80 employees + tons of volunteers
- free + commercial

Note 1: that has led to two quite distinct **R** dialects.

Note 2: more and more users rely on tidyverse and many recent packages use the tidyverse grammar.

Note 3: we will see a bit of both dialects.

# Getting started with tidyverse



Note: there are also multiple tutorials on the web  
(e.g. <https://www.r-bloggers.com/lesser-known-dplyr-tricks/>).

# Key tidyverse principles

- one verb = one action = one function
- 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 (exception: `ggplot2` that uses `+"` instead)
- the outcome of a given function is always of the same class

# Key tidyverse principles

- one verb = one action = one function
- 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 (exception: `ggplot2` that uses `"+"` instead)
- the outcome of a given function is always of the same class

## Pros

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

# Key tidyverse principles

- one verb = one action = one function
- 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 (exception: `ggplot2` that uses "`+`" instead)
- the outcome of a given function is always of the same class

## 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. help not looked at)
- wider gap between users and developers



# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- **dplyr**
- tibbles
- advanced but useful features

# dplyr verbs

Useful dplyr functions:

- add column with `mutate()`

```
iris %>%  
  mutate(double_SL = 2*Sepal.Length) %>%  
  head(n = 3)
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	double_SL
## 1	5.1	3.5	1.4	0.2	setosa	10.2
## 2	4.9	3.0	1.4	0.2	setosa	9.8
## 3	4.7	3.2	1.3	0.2	setosa	9.4

# dplyr verbs

## Useful dplyr functions:

- add column with `mutate()`

```
iris %>%  
  mutate(double_SL = 2*Sepal.Length) %>%  
  head(n = 3)
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	double_SL
## 1	5.1	3.5	1.4	0.2	setosa	10.2
## 2	4.9	3.0	1.4	0.2	setosa	9.8
## 3	4.7	3.2	1.3	0.2	setosa	9.4

- create and keep only new columns with `transmute()`

```
iris %>%  
  transmute(double_SL = 2*Sepal.Length) %>%  
  head(n = 3)
```

##	double_SL
## 1	10.2
## 2	9.8
## 3	9.4

# dplyr verbs

Useful dplyr functions:

- select columns with `select()`

```
iris %>%  
  select(Sepal.Length) %>%  
  head(n = 3)
```

```
##   Sepal.Length  
## 1           5.1  
## 2           4.9  
## 3           4.7
```

## dplyr verbs

Useful dplyr functions:

- select columns with `select()`

```
iris %>%  
  select(Sepal.Length) %>%  
  head(n = 3)  
  
##   Sepal.Length  
## 1           5.1  
## 2           4.9  
## 3           4.7
```

- select rows with `filter()`

```
iris %>%  
  filter(Species == "virginica") %>%  
  head(n = 3)  
  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width  Species  
## 1           6.3          3.3          6.0          2.5 virginica  
## 2           5.8          2.7          5.1          1.9 virginica  
## 3           7.1          3.0          5.9          2.1 virginica
```

# dplyr verbs

Useful dplyr functions:

- select columns with `select()`

```
iris %>%  
  select(Sepal.Length) %>%  
  head(n = 3)  
  
##   Sepal.Length  
## 1           5.1  
## 2           4.9  
## 3           4.7
```

- select rows with `filter()`

```
iris %>%  
  filter(Species == "virginica") %>%  
  head(n = 3)  
  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width  Species  
## 1           6.3         3.3         6.0         2.5 virginica  
## 2           5.8         2.7         5.1         1.9 virginica  
## 3           7.1         3.0         5.9         2.1 virginica
```

- sort rows with `arrange()`

```
iris %>%  
  arrange(Petal.Length) %>% ## arrange(desc(Petal.Length)) for the other direction  
  head(n = 3)  
  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1           4.6         3.6         1.0         0.2  setosa  
## 2           4.3         3.0         1.1         0.1  setosa  
## 3           5.8         4.0         1.2         0.2  setosa
```

# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- **tibbles**
- advanced but useful features

## What are tibbles?

Tibbles are modified data frames that can be used and (sometimes automatically) produced by tidyverse packages:

```
iris %>%
  group_by(Species) %>%
  mutate(Sepal.Length.meam = mean(Sepal.Length)) -> iris_tbl ## note the inverted arrow!
iris_tbl
```

## # A tibble: 150 x 6

## # Groups: Species [3]

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
##	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<dbl>
## 1	5.1	3.5	1.4	0.2	setosa	5.01
## 2	4.9	3	1.4	0.2	setosa	5.01
## 3	4.7	3.2	1.3	0.2	setosa	5.01
## 4	4.6	3.1	1.5	0.2	setosa	5.01
## 5	5	3.6	1.4	0.2	setosa	5.01
## 6	5.4	3.9	1.7	0.4	setosa	5.01
## 7	4.6	3.4	1.4	0.3	setosa	5.01
## 8	5	3.4	1.5	0.2	setosa	5.01
## 9	4.4	2.9	1.4	0.2	setosa	5.01
## 10	4.9	3.1	1.5	0.1	setosa	5.01

## # ... with 140 more rows

Note: most of what works on objects of class `data.frame` works on objects of class `tbl` (but not all as they don't consider row names).



# From tibbles to data frames and back

You can easily convert one into the other:

```
head(data.frame(iris_tbl))
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
## 1	5.1	3.5	1.4	0.2	setosa	5.006
## 2	4.9	3.0	1.4	0.2	setosa	5.006
## 3	4.7	3.2	1.3	0.2	setosa	5.006
## 4	4.6	3.1	1.5	0.2	setosa	5.006
## 5	5.0	3.6	1.4	0.2	setosa	5.006
## 6	5.4	3.9	1.7	0.4	setosa	5.006

# From tibbles to data frames and back

You can easily convert one into the other:

```
head(data.frame(iris_tbl))
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
## 1	5.1	3.5	1.4	0.2	setosa	5.006
## 2	4.9	3.0	1.4	0.2	setosa	5.006
## 3	4.7	3.2	1.3	0.2	setosa	5.006
## 4	4.6	3.1	1.5	0.2	setosa	5.006
## 5	5.0	3.6	1.4	0.2	setosa	5.006
## 6	5.4	3.9	1.7	0.4	setosa	5.006

```
as_tibble(data.frame(iris_tbl)) ## no need for head() when using tibbles!
```

```
## # A tibble: 150 x 6
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
##	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<dbl>
## 1	5.1	3.5	1.4	0.2	setosa	5.01
## 2	4.9	3	1.4	0.2	setosa	5.01
## 3	4.7	3.2	1.3	0.2	setosa	5.01
## 4	4.6	3.1	1.5	0.2	setosa	5.01
## 5	5	3.6	1.4	0.2	setosa	5.01
## 6	5.4	3.9	1.7	0.4	setosa	5.01
## 7	4.6	3.4	1.4	0.3	setosa	5.01
## 8	5	3.4	1.5	0.2	setosa	5.01
## 9	4.4	2.9	1.4	0.2	setosa	5.01
## 10	4.9	3.1	1.5	0.1	setosa	5.01

```
## # ... with 140 more rows
```

# From tibbles to data frames and back

You can easily convert one into the other:

```
head(data.frame(iris_tbl))
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
## 1	5.1	3.5	1.4	0.2	setosa	5.006
## 2	4.9	3.0	1.4	0.2	setosa	5.006
## 3	4.7	3.2	1.3	0.2	setosa	5.006
## 4	4.6	3.1	1.5	0.2	setosa	5.006
## 5	5.0	3.6	1.4	0.2	setosa	5.006
## 6	5.4	3.9	1.7	0.4	setosa	5.006

```
as_tibble(data.frame(iris_tbl)) ## no need for head() when using tibbles!
```

```
## # A tibble: 150 x 6
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Sepal.Length.meam
##	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<dbl>
## 1	5.1	3.5	1.4	0.2	setosa	5.01
## 2	4.9	3	1.4	0.2	setosa	5.01
## 3	4.7	3.2	1.3	0.2	setosa	5.01
## 4	4.6	3.1	1.5	0.2	setosa	5.01
## 5	5	3.6	1.4	0.2	setosa	5.01
## 6	5.4	3.9	1.7	0.4	setosa	5.01
## 7	4.6	3.4	1.4	0.3	setosa	5.01
## 8	5	3.4	1.5	0.2	setosa	5.01
## 9	4.4	2.9	1.4	0.2	setosa	5.01
## 10	4.9	3.1	1.5	0.1	setosa	5.01

```
## # ... with 140 more rows
```

Note: for linear models, it is safer to convert everything into data frames!

# How to influence the display of tibbles?

You can change the number of rows being displayed:

```
print(iris_tbl, n = 2)

## # A tibble: 150 x 6
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length.meam
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>         <dbl>
## 1         5.1           3.5           1.4           0.2 setosa         5.01
## 2         4.9           3             1.4           0.2 setosa         5.01
## # ... with 148 more rows
```

```
print(iris_tbl, n = 8)

## # A tibble: 150 x 6
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length.meam
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>         <dbl>
## 1         5.1           3.5           1.4           0.2 setosa         5.01
## 2         4.9           3             1.4           0.2 setosa         5.01
## 3         4.7           3.2           1.3           0.2 setosa         5.01
## 4         4.6           3.1           1.5           0.2 setosa         5.01
## 5         5             3.6           1.4           0.2 setosa         5.01
## 6         5.4           3.9           1.7           0.4 setosa         5.01
## 7         4.6           3.4           1.4           0.3 setosa         5.01
## 8         5             3.4           1.5           0.2 setosa         5.01
## # ... with 142 more rows
```

# How to influence the display of tibbles?

You can change the number of rows being displayed:

```
print(iris_tbl, n = 2)

## # A tibble: 150 x 6
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length.meam
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>         <dbl>
## 1         5.1           3.5           1.4           0.2 setosa         5.01
## 2         4.9           3             1.4           0.2 setosa         5.01
## # ... with 148 more rows
```

```
print(iris_tbl, n = 8)

## # A tibble: 150 x 6
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.Length.meam
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>         <dbl>
## 1         5.1           3.5           1.4           0.2 setosa         5.01
## 2         4.9           3             1.4           0.2 setosa         5.01
## 3         4.7           3.2           1.3           0.2 setosa         5.01
## 4         4.6           3.1           1.5           0.2 setosa         5.01
## 5         5             3.6           1.4           0.2 setosa         5.01
## 6         5.4           3.9           1.7           0.4 setosa         5.01
## 7         4.6           3.4           1.4           0.3 setosa         5.01
## 8         5             3.4           1.5           0.2 setosa         5.01
## # ... with 142 more rows
```

Note: to always display all row you can set options(dplyr.print\_min = Inf).

# How to influence the display of tibbles?

You can change the number of digits being displayed:

Default:

```
x <- as_tibble(data.frame(pi = pi))
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1  3.14
```

# How to influence the display of tibbles?

You can change the number of digits being displayed:

Default:

```
x <- as_tibble(data.frame(pi = pi))
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1  3.14
```

Changing setting:

```
old_opt <- options(pillar.sigfig = 10)
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1 3.141592654
```

# How to influence the display of tibbles?

You can change the number of digits being displayed:

Default:

```
x <- as_tibble(data.frame(pi = pi))
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1  3.14
```

Changing setting:

```
old_opt <- options(pillar.sigfig = 10)
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1 3.141592654
```

Resetting setting:

```
options(old_opt)
x
## # A tibble: 1 x 1
##       pi
##   <dbl>
## 1  3.14
```



# Getting started with R

## 1 Introduction

## 2 Vectors

- general properties
- types & classes
- factors
- functions

## 3 Matrices and arrays

- general properties
- functions

## 4 List

- general properties
- functions

## 5 Data frames

- general properties
- challenge
- functions

## 6 Importing & exporting data

## 7 tidyverse

- introduction
- dplyr
- tibbles
- advanced but useful features

## group\_by()

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

## group\_by()

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

It is very powerful when combined to:

- `summarize()` → one value per group

## group\_by()

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

It is very powerful when combined to:

- `summarize()` → one value per group
- `mutate()` or `transmute()` → one value per observation
- `slice()` → select some rows for each "group"

## group\_by()

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

It is very powerful when combined to:

- `summarize()` → one value per group
- `mutate()` or `transmute()` → one value per observation
- `slice()` → select some rows for each "group"
- `do()` → for applying custom functions on each "group"  
(but advanced and perhaps soon deprecated)

## group\_by() with summarize()

Example: for each species you want the mean petal length, the sd and the number of observations:

```
iris %>%  
  group_by(Species) %>%  
  summarize(mean_PL = mean(Petal.Length),  
            sd_PL = sd(Petal.Length),  
            n = n())  
  
## # A tibble: 3 x 4  
##   Species    mean_PL sd_PL      n  
##   <fct>      <dbl> <dbl> <int>  
## 1 setosa      1.46 0.174    50  
## 2 versicolor  4.26 0.470    50  
## 3 virginica   5.55 0.552    50
```

## group\_by() with mutate()

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

```
iris %>%
  group_by(Species) %>%
  mutate(mean_PL = mean(Petal.Length),
         sd_PL = sd(Petal.Length),
         n = n())
```

## # A tibble: 150 x 8

## # Groups: Species [3]

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	mean_PL	sd_PL	n
##	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<dbl>	<dbl>	<int>
## 1	5.1	3.5	1.4	0.2	setosa	1.46	0.174	50
## 2	4.9	3	1.4	0.2	setosa	1.46	0.174	50
## 3	4.7	3.2	1.3	0.2	setosa	1.46	0.174	50
## 4	4.6	3.1	1.5	0.2	setosa	1.46	0.174	50
## 5	5	3.6	1.4	0.2	setosa	1.46	0.174	50
## 6	5.4	3.9	1.7	0.4	setosa	1.46	0.174	50
## 7	4.6	3.4	1.4	0.3	setosa	1.46	0.174	50
## 8	5	3.4	1.5	0.2	setosa	1.46	0.174	50
## 9	4.4	2.9	1.4	0.2	setosa	1.46	0.174	50
## 10	4.9	3.1	1.5	0.1	setosa	1.46	0.174	50

## # ... with 140 more rows

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



## group\_by() with slice()

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

```
iris %>%  
  group_by(Species) %>%  
  slice(1:2)
```

## # A tibble: 6 x 5

## # Groups: Species [3]

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
##	<dbl>	<dbl>	<dbl>	<dbl>	<fct>
## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3	1.4	0.2	setosa
## 3	7	3.2	4.7	1.4	versicolor
## 4	6.4	3.2	4.5	1.5	versicolor
## 5	6.3	3.3	6	2.5	virginica
## 6	5.8	2.7	5.1	1.9	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` (careful, quite large!) and figure out:

- the death toll for all individuals below 15 yrs for each year
- which were the top 3 detailed causes of death before 1930 for each of the 8 broader categories (most difficult)

## Possible solution

Which were the top 3 detailed causes of death before 1930 for each of the 8 broader categories?

```
str(deaths_UK)

## 'data.frame': 1445659 obs. of  8 variables:
##  $ ICD      : chr  "0010" "0010" "0020" "0020" ...
##  $ Year      : num  1901 1901 1901 1901 1901 ...
##  $ Sex       : Factor w/ 2 levels "Males","Females": 1 2 1 2 1 2 1 2 1 2 ...
##  $ Deaths   : num   2 7 4 7 4 3 5 11 34 27 ...
##  $ Age_cat   : Factor w/ 6 levels "Infant (< 1 yr)",...: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Desc      : Factor w/ 6673 levels "\"Pink\" disease",...: 5995 5995 5994 5994 5993 5993 1563 1563 1141 1141 ...
##  $ Cause     : Factor w/ 21 levels "Certain conditions originating in the perinatal period",...: 2 2 2 2 2 2 2 2 2 2 ...
##  $ Cause_simple: Factor w/ 8 levels "Infectious and parasitic diseases",...: 1 1 1 1 1 1 1 1 1 1 ...
```

## Possible solution

Which were the top 3 detailed causes of death before 1930 for each of the 8 broader categories?

```
str(deaths_UK)

## 'data.frame': 1445659 obs. of 8 variables:
## $ ICD      : chr  "0010" "0010" "0020" "0020" ...
## $ Year     : num  1901 1901 1901 1901 1901 ...
## $ Sex      : Factor w/ 2 levels "Males","Females": 1 2 1 2 1 2 1 2 1 2 ...
## $ Deaths  : num   2  7  4  7  4  3  5 11 34 27 ...
## $ Age_cat  : Factor w/ 6 levels "Infant (< 1 yr)",...: 1 1 1 1 1 1 1 1 1 1 ...
## $ Desc     : Factor w/ 6673 levels "\"Pink\" disease",...: 5995 5995 5994 5994 5993 5993 1563 1563 1141 1141 ...
## $ Cause    : Factor w/ 21 levels "Certain conditions originating in the perinatal period",...: 2 2 2 2 2 2 2 2 2 2 ...
## $ Cause_simple: Factor w/ 8 levels "Infectious and parasitic diseases",...: 1 1 1 1 1 1 1 1 1 1 ...
```

```
deaths_UK %>%
  filter(Year <= 1930) %>%
  group_by(Cause_simple, Desc) %>%
  summarize(tot_deaths = sum(Deaths)) %>%
  arrange(Cause_simple, desc(tot_deaths)) %>%
  slice(1:3) %>%
  View() ## for nice display as a spreadsheet
```

## Possible solution

Which were the top 3 detailed causes of death before 1930 for each of the 8 broader categories?

```
str(deaths_UK)

## 'data.frame': 1445659 obs. of  8 variables:
##  $ ICD          : chr  "0010" "0010" "0020" "0020" ...
##  $ Year          : num  1901 1901 1901 1901 1901 ...
##  $ Sex           : Factor w/ 2 levels "Males","Females": 1 2 1 2 1 2 1 2 1 2 ...
##  $ Deaths       : num   2  7  4  7  4  3  5 11 34 27 ...
##  $ Age_cat       : Factor w/ 6 levels "Infant (< 1 yr)",...: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Desc          : Factor w/ 6673 levels "\"Pink\" disease",...: 5995 5995 5994 5994 5993 5993 1563 1563 1141 1141 ...
##  $ Cause         : Factor w/ 21 levels "Certain conditions originating in the perinatal period",...: 2 2 2 2 2 2 2 2 2 2 ...
##  $ Cause_simple  : Factor w/ 8 levels "Infectious and parasitic diseases",...: 1 1 1 1 1 1 1 1 1 1 ...
```

```
deaths_UK %>%
  filter(Year <= 1930) %>%
  group_by(Cause_simple, Desc) %>%
  summarize(tot_deaths = sum(Deaths)) %>%
  arrange(Cause_simple, desc(tot_deaths)) %>%
  slice(1:3) %>%
  View() ## for nice display as a spreadsheet
```

Note: to clearly understand, check the output after each step!

## group\_by() with do()

Advanced: you want to apply a function that returns several elements, such as the range of `Petal.Length` for each species of iris:

```
iris %>%  
  group_by(Species) %>%  
  do(tibble(range = range(.$Petal.Length))) ## must be turned into a tibble or data.frame to work  
  
## # A tibble: 6 x 2  
## # Groups:   Species [3]  
##   Species    range  
##   <fct>     <dbl>  
## 1 setosa      1  
## 2 setosa     1.9  
## 3 versicolor 3  
## 4 versicolor 5.1  
## 5 virginica  4.5  
## 6 virginica  6.9
```

## group\_by() with do()

Advanced: you want to apply a function that returns several elements, such as the range of `Petal.Length` for each species of iris:

```
iris %>%
  group_by(Species) %>%
  do(tibble(range = range(.$Petal.Length))) ## must be turned into a tibble or data.frame to work

## # A tibble: 6 x 2
## # Groups:   Species [3]
##   Species   range
##   <fct>     <dbl>
## 1 setosa     1
## 2 setosa    1.9
## 3 versicolor 3
## 4 versicolor 5.1
## 5 virginica 4.5
## 6 virginica 6.9
```

More sophisticated alternative (combining tidyverse and standard R):

```
Range <- function(x, ...) c(min = min(x, ...), max = max(x, ...)) ## same as range() but output a named vector!
iris %>%
  group_by(Species) %>%
  do(data.frame(as.list(Range(.$Petal.Length)))) ## for this specific example we could of course use summarize with min and max

## # A tibble: 3 x 3
## # Groups:   Species [3]
##   Species   min   max
##   <fct>     <dbl> <dbl>
## 1 setosa     1     1.9
## 2 versicolor 3     5.1
## 3 virginica 4.5    6.9
```

## group\_by() with do()

Advanced: you want to apply a function that returns several elements, such as the range of `Petal.Length` for each species of iris:

```
iris %>%
  group_by(Species) %>%
  do(tibble(range = range(.$Petal.Length))) ## must be turned into a tibble or data.frame to work

## # A tibble: 6 x 2
## # Groups:   Species [3]
##   Species   range
##   <fct>     <dbl>
## 1 setosa     1
## 2 setosa    1.9
## 3 versicolor 3
## 4 versicolor 5.1
## 5 virginica 4.5
## 6 virginica 6.9
```

More sophisticated alternative (combining tidyverse and standard R):

```
Range <- function(x, ...) c(min = min(x, ...), max = max(x, ...)) ## same as range() but output a named vector!
iris %>%
  group_by(Species) %>%
  do(data.frame(as.list(Range(.$Petal.Length)))) ## for this specific example we could of course use summarize with min and max

## # A tibble: 3 x 3
## # Groups:   Species [3]
##   Species   min   max
##   <fct>     <dbl> <dbl>
## 1 setosa     1     1.9
## 2 versicolor 3     5.1
## 3 virginica 4.5    6.9
```

Alternatively, stick to standard R and the function `tapply()`.



# Using dplyr to merge datasets

## Data frame #1:

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

```
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           4.6           1.5  setosa
```

## Data frame #2:

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

```
my_df2  
##   Sepal.Length Petal.Width  Species  
## 1           6.3           2.5 virginica  
## 2           5.8           1.9 virginica  
## 3           7.1           2.1 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         4.6         1.5    setosa         NA
## 5         6.3          NA virginica         2.5
## 6         5.8          NA virginica         1.9
## 7         7.1          NA virginica         2.1
## 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 individuals below 15 yrs only.

# Possible solution

```
deaths_UK %>%
  filter(Age_cat %in% levels(Age_cat)[1:3]) %>%
  group_by(Year) %>%
  summarize(tot_Deaths = sum(Deaths)) -> deaths_processed
```

```
population_UK %>%
  group_by(Year) %>%
  summarize(tot_Pop = sum(Pop)) -> pop_processed
```

```
full_join(deaths_processed, pop_processed) %>%
  mutate(rel_Deaths = 1000*tot_Deaths/tot_Pop)
```

```
## Joining, by = "Year"
```

```
## # A tibble: 116 x 4
```

```
##   Year tot_Deaths tot_Pop rel_Deaths
##   <dbl>      <dbl>   <dbl>    <dbl>
## 1  1901      223738 32612100     6.86
## 2  1902      206866 32950800     6.28
## 3  1903      199771 33293400     6.00
## 4  1904      219726 33731300     6.51
## 5  1905      193913 33988900     5.71
## 6  1906      198937 34342000     5.79
## 7  1907      182930 34699000     5.27
## 8  1908      183493 35155400     5.22
## 9  1909      169945 35423700     4.80
## 10 1910      157712 35792000     4.41
## # ... with 106 more rows
```

## Reshaping data frame

For most data analyses, you need:

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

Unfortunately, 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

Unfortunately, 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 (useless?)

# From wide to long

you have:

```
iris_wide ## note: stored in BeginR!  
##      Species day_1 day_2 day_3 day_4 day_5  
## 1    setosa   5.1   4.9   4.7   4.6   5.0  
## 2 versicolor  7.0   6.4   6.9   5.5   6.5
```

you want:

```
iris_long ## note: stored in BeginR!  
##      Species day Sepal.Length  
## 1    setosa   1         5.1  
## 2    setosa   2         4.9  
## 3    setosa   3         4.7  
## 4    setosa   4         4.6  
## 5    setosa   5         5.0  
## 6 versicolor  1         7.0  
## 7 versicolor  2         6.4  
## 8 versicolor  3         6.9  
## 9 versicolor  4         5.5  
## 10 versicolor 5         6.5
```

# From wide to long

you have:

```
iris_wide ## note: stored in BeginR!

##      Species day_1 day_2 day_3 day_4 day_5
## 1    setosa   5.1   4.9   4.7   4.6   5.0
## 2 versicolor 7.0   6.4   6.9   5.5   6.5
```

you want:

```
iris_long ## note: stored in BeginR!

##      Species day Sepal.Length
## 1    setosa   1      5.1
## 2    setosa   2      4.9
## 3    setosa   3      4.7
## 4    setosa   4      4.6
## 5    setosa   5      5.0
## 6 versicolor 1      7.0
## 7 versicolor 2      6.4
## 8 versicolor 3      6.9
## 9 versicolor 4      5.5
## 10 versicolor 5      6.5
```

you do:

```
iris_wide %>%
  gather(key = "day", value = "Sepal.Length", -Species) %>% ## after, optional
  separate(day, c("obs", "day")) %>%
  select(-obs) %>%
  as_tibble() %>%
  mutate(day = as.numeric(day)) %>%
  arrange(Species, day)

## # A tibble: 10 x 3
##   Species    day Sepal.Length
##   <fct>    <dbl>         <dbl>
## 1 setosa      1           5.1
## 2 setosa      2           4.9
## 3 setosa      3           4.7
## 4 setosa      4           4.6
## 5 setosa      5           5
## 6 versicolor 1           7
## 7 versicolor 2           6.4
## 8 versicolor 3           6.9
## 9 versicolor 4           5.5
## 10 versicolor 5           6.5
```

Note: run the code step by step to understand everything that is happening!