Data-wrangling on R

Rémi Mahmoud

Who is talking ?

What you have to do:

Open Rstudio

What you have to do:

Open Rstudio

Open the file data_wrangling_slides_INRAE.pdf

What you have to do:

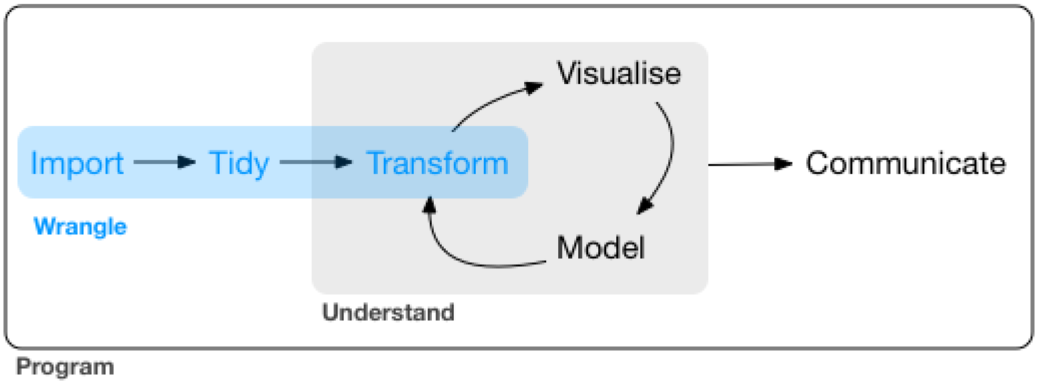
Open Rstudio

Open the file data_wrangling_slides_INRAE.pdf

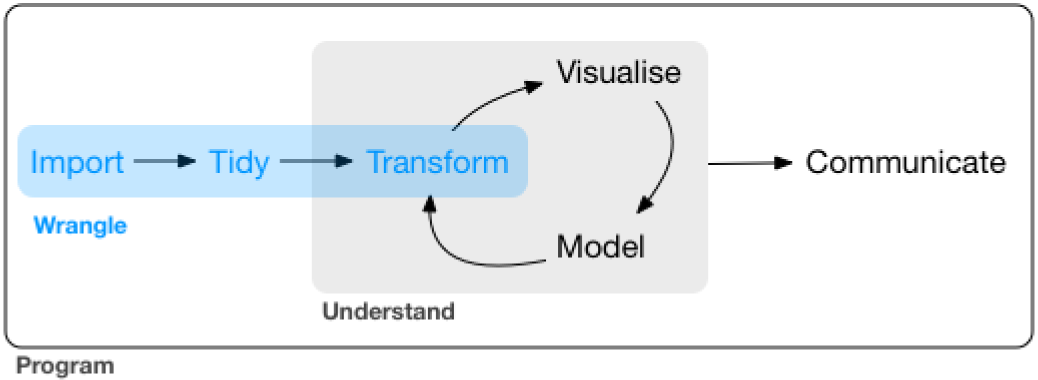
Stop me if anything is unclear

What is data-wrangling ?

What is data-wrangling ?



What is data-wrangling ?



- data-wrangling is the set of operations on **raw** data that leads to **non messy** (tidy) data.

What we will talk about today

What we will talk about today

- Data importation

What we will talk about today

- Data importation
- Manipulate data (filtering, arranging data etc.)

What we will talk about today

- Data importation
- Manipulate data (filtering, arranging data etc.)
- Tidy data

What we will talk about today

- Data importation
- Manipulate data (filtering, arranging data etc.)
- Tidy data

What we will NOT talk about today

What we will talk about today

- Data importation
- Manipulate data (filtering, arranging data etc.)
- Tidy data

What we will NOT talk about today

- Dealing with missing values / outliers

Framework

All manipulations will be done in the `tidyverse` framework.

Framework

All manipulations will be done in the `tidyverse` framework.

Hence, you should, if not already done, run the following command in R **NOW**

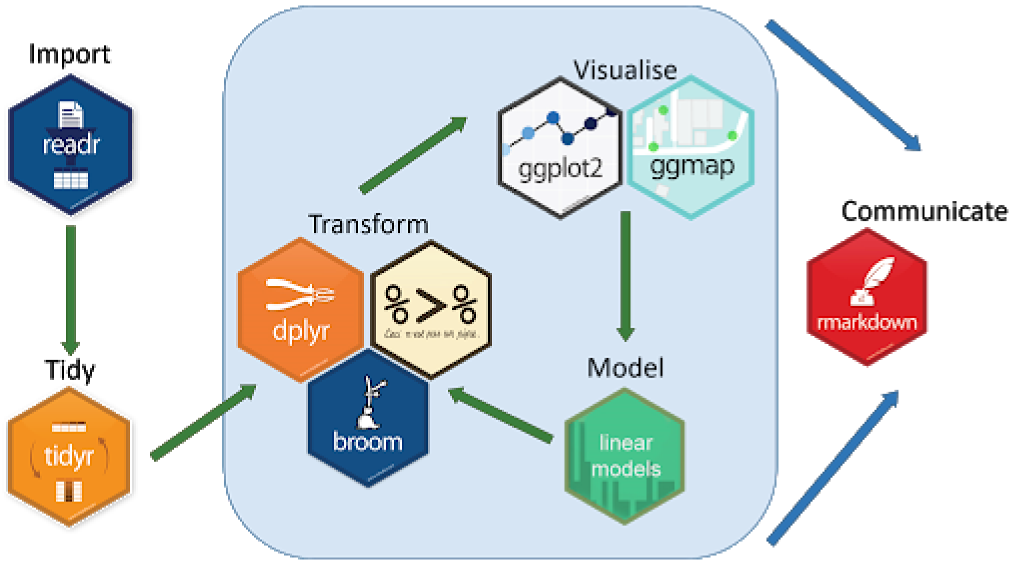
```
install.packages("tidyverse")
```

Tidyverse = Tidy universe

Tidyverse is a set of packages with differents purposes, that share the same syntax and that are designed to work in a complementary way

Tidyverse = Tidy universe

Tidyverse is a set of packages with differents purposes, that share the same syntax and that are designed to work in a complementary way



You can list the packages available in the `tidyverse` by running the following command:

```
tidyverse::tidyverse_packages()
```

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
##  [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "tidyverse"
```

You can list the packages available in the `tidyverse` by running the following command:

```
tidyverse::tidyverse_packages()
```

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
##  [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "tidyverse"
```

You can see that `ggplot2` that you may already know belongs to the `tidyverse`. But there are *many other packages* !

You can list the packages available in the `tidyverse` by running the following command:

```
tidyverse::tidyverse_packages()
```

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
##  [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "tidyverse"
```

You can see that `ggplot2` that you may already know belongs to the `tidyverse`. But there are *many other packages* !

For instance, the `forcats` package allows to work in a convenient way with factors, `lubridate` with dates etc. .

You can list the packages available in the `tidyverse` by running the following command:

```
tidyverse::tidyverse_packages()
```

```
##  [1] "broom"      "cli"        "crayon"     "dbplyr"     "dplyr"
##  [6] "forcats"    "ggplot2"    "haven"      "hms"        "httr"
## [11] "jsonlite"   "lubridate"  "magrittr"   "modelr"     "pillar"
## [16] "purrr"      "readr"      "readxl"     "reprex"     "rlang"
## [21] "rstudioapi" "rvest"      "stringr"    "tibble"     "tidyr"
## [26] "xml2"       "tidyverse"
```

You can see that `ggplot2` that you may already know belongs to the `tidyverse`. But there are *many other packages* !

For instance, the `forcats` package allows to work in a convenient way with factors, `lubridate` with dates etc. .

For now, we will take a closer look to the `readr` and to a lesser extent `readxl` packages. These packages are useful to **import** data.

Import data with `readr`

The `read_csv` function of `readr` allows to read csv files.

Import data with `readr`

The `read_csv` function of `readr` allows to read csv files.

```
data_work <- readr::read_csv('data/iris.csv')
```

```
## Parsed with column specification:
## cols(
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

`read_csv` is faster than base R `read.csv` and it parses well different types of columns.

This function has also other arguments that may be useful for you when using it:

This function has also other arguments that may be useful for you when using it:

- `skip` to specify the number of lines to skip before reading the file
- `na` to specify what should be considered as `NA` (for ex: you could put `na = "Not answered"`)
- `col_names` to specify the names of the columns you want to have in your dataset.
- See `?read_csv` for other arguments.

Tibble vs dataframe

Let us take a look at the data we have imported.

```
head(data_work)
```

```
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>  
## 1          5.1         3.5          1.4         0.2 setosa 
## 2          4.9         3            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           3.6          1.4         0.2 setosa 
## 6          5.4         3.9          1.7         0.4 setosa
```

Tibble vs dataframe

Let us take a look at the data we have imported.

```
head(data_work)
```

```
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
```

Note the particular output of the print:

- *A tibble*
- The type of each column is written under each col_name

The tibble is an alternative to the classical `data.frame` of base R

The tibble is an alternative to the classical `data.frame` of base R

As part of the tidyverse, it is mainly used in the tidyverse' packages.

The tibble is an alternative to the classical `data.frame` of base R

As part of the tidyverse, it is mainly used in the tidyverse' packages.

The difference should not worry you: the main difference with a classical dataframe is the nicer output when printing (run `iris` in R to see the difference).

The tibble is an alternative to the classical `data.frame` of base R

As part of the tidyverse, it is mainly used in the tidyverse' packages.

The difference should not worry you: the main difference with a classical dataframe is the nicer output when printing (run `iris` in R to see the difference).

By the way, note that a tibble **is a data.frame**

```
is.data.frame(data_work)
```

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

An other package of the `tidyverse`: `readxl` to read .xlsx files

The function `read_xlsx` allows you to read .xlsx files.

An other package of the `tidyverse`: `readxl` to read .xlsx files

The function `read_xlsx` allows you to read .xlsx files.

Some arguments are useful for you:

- `sheet` : name of the sheet of the file you want to read (if you provide a string), or position of the sheet you want to read (if you provide an integer)
- same arguments as `read_csv` (`na`, `skip` etc.)
- see `?read_xlsx` for details.

Is everything ok until now ?

Exercises

~ 30 minutes

Exercises

~ 30 minutes

Download the zip Github depository, open "Import_data_exercises.Rmd" with Rstudio

Manipulate your data using `dplyr`

`dplyr` is a package of the `tidyverse` designed to manipulate your data easily.

Manipulate your data using `dplyr`

`dplyr` is a package of the `tidyverse` designed to manipulate your data easily.

> what do we mean by manipulating the data easily ?

Manipulate your data using `dplyr`

`dplyr` is a package of the `tidyverse` designed to manipulate your data easily.

> what do we mean by manipulating the data easily ?

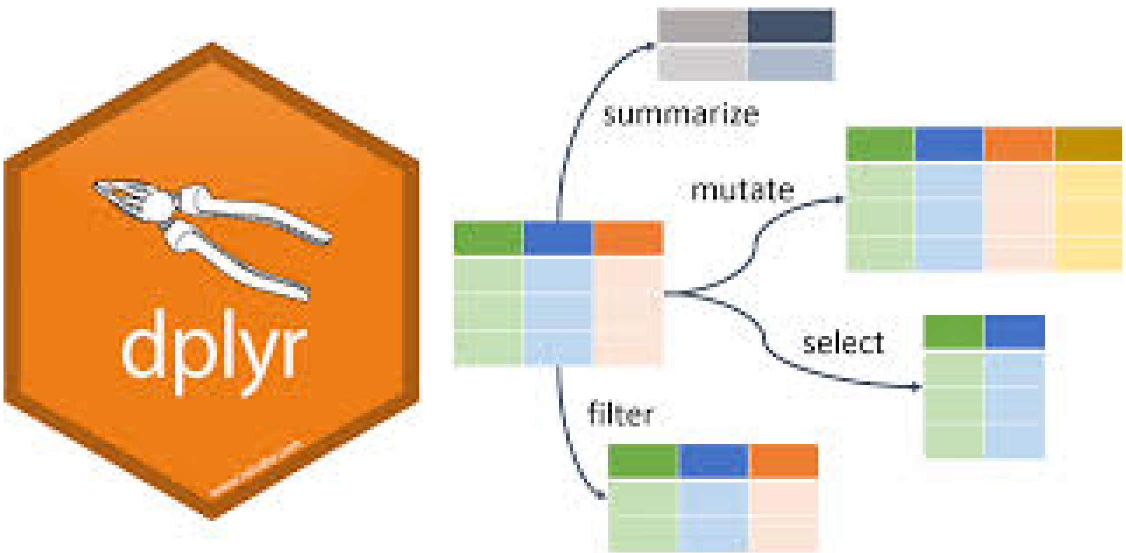
Select columns, filter their rows, create new columns etc.

Manipulate your data using `dplyr`

`dplyr` is a package of the `tidyverse` designed to manipulate your data easily.

> what do we mean by manipulating the data easily ?

Select columns, filter their rows, create new columns etc.



Let us consider the dataset `data_work` previously introduced (it is simply the well know `iris` dataset turned into tibble).

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Let us see the manipulations we can do on this dataset.

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
    - *select* the 3rd column

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
    - *select* the 3rd column

```
select(data_work, 3)
```

```
## # A tibble: 150 x 1
##   Petal.Length
##          <dbl>
## 1          1.4
## 2          1.4
## 3          1.3
## 4          1.5
## 5          1.4
## 6          1.7
## # ... with 144 more rows
```

`dplyr::select`

In a data analysis, we could be interested in:

- *select* some columns, for instance:
    - *select* column Sepal.Width

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
  - *select* column Sepal.Width

```
select(data_work, Sepal.Width)
# Note the absence of " around Sepal .Width
```

```
## # A tibble: 150 x 1
##   Sepal.Width
##         <dbl>
## 1         3.5
## 2         3
## 3         3.2
## 4         3.1
## 5         3.6
## 6         3.9
## # ... with 144 more rows
```

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
    - *select* all columns except Sepal.width and Sepal.Length)

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
  - *select* all columns except Sepal.width and Sepal.Length)

```
select(data_work, - c(Sepal.Width, Sepal.Length))
```

```
## # A tibble: 150 x 3
##   Petal.Length Petal.Width Species
##          <dbl>       <dbl> <chr>
## 1          1.4         0.2 setosa
## 2          1.4         0.2 setosa
## 3          1.3         0.2 setosa
## 4          1.5         0.2 setosa
## 5          1.4         0.2 setosa
## 6          1.7         0.4 setosa
## # ... with 144 more rows
```

dplyr::select

In a data analysis, we could be interested in:

- *select* some columns, for instance:
    - *select* all columns except Sepal.width and Sepal.Length)

```
select(data_work, - c(Sepal.Width, Sepal.Length))
```

```
## # A tibble: 150 x 3
##   Petal.Length Petal.Width Species
##          <dbl>       <dbl> <chr>
## 1          1.4         0.2 setosa
## 2          1.4         0.2 setosa
## 3          1.3         0.2 setosa
## 4          1.5         0.2 setosa
## 5          1.4         0.2 setosa
## 6          1.7         0.4 setosa
## # ... with 144 more rows
```

Note the absence of " around Sepal.Width and Sepal.Length, and the **-** that means **except**

`dplyr::select`: helpers()

`select()` is provided with many *functions helpers* that you can use to select columns, for instance:

- `select(data_work, contains("pal"))`: all columns of `data_work` with a name containing "pal"

`dplyr::select`: helpers()

`select()` is provided with many *functions helpers* that you can use to select columns, for instance:

- `select(data_work, contains("pal"))`: all columns of `data_work` with a name containing "pal"

- `select(data_work, starts_with("Se"))`: *can you guess it ?*

`dplyr::select`: helpers()

`select()` is provided with many *functions helpers* that you can use to select columns, for instance:

- `select(data_work, contains("pal"))`: all columns of `data_work` with a name containing "pal"
- `select(data_work, starts_with("Se"))`: *can you guess it ?*
- `select(data_work, ends_with("th"))`: *can you guess it ?*

`dplyr::select`: helpers()

`select()` is provided with many *functions helpers* that you can use to select columns, for instance:

- `select(data_work, contains("pal"))`: all columns of `data_work` with a name containing "pal"
- `select(data_work, starts_with("Se"))`: *can you guess it ?*
- `select(data_work, ends_with("th"))`: *can you guess it ?*
- `select(data_work, matches("*th"))`: *can you guess it ?* (select columns with a name matching a regular expression)

`dplyr::select`: helpers()

`select()` is provided with many *functions helpers* that you can use to select columns, for instance:

- `select(data_work, contains("pal"))`: all columns of `data_work` with a name containing "pal"
- `select(data_work, starts_with("Se"))`: *can you guess it ?*
- `select(data_work, ends_with("th"))`: *can you guess it ?*
- `select(data_work, matches("*th"))`: *can you guess it ?* (select columns with a name matching a regular expression)

> That's one of the assets of the dplyr syntax: it looks like almost natural language.

`dplyr::filter`

In a data analysis, we could be interested in:

- *filter* rows based on the values of some columns (predicates), for instance:
    - *filter* rows of `data_work` with individuals having their length of Sepal greater than 4

dplyr::filter

In a data analysis, we could be interested in:

- *filter* rows based on the values of some columns (predicates), for instance:
    - *filter* rows of `data_work` with individuals having their length of Sepal greater than 4

```
filter(data_work, Sepal.Length > 4)
```

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

`dplyr::filter`

In a data analysis, we could be interested in:

- *filter* rows based on the values of some columns (predicates), for instance:
    - *filter* rows of `data_work` of species "virginica"

dplyr::filter

In a data analysis, we could be interested in:

- *filter* rows based on the values of some columns (predicates), for instance:
    - *filter* rows of `data_work` of species "virginica"

```
filter(data_work, Species == "virginica")
```

```
## # A tibble: 50 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          6.3         3.3          6           2.5 virginica
## 2          5.8         2.7          5.1         1.9 virginica
## 3          7.1         3            5.9         2.1 virginica
## 4          6.3         2.9          5.6         1.8 virginica
## 5          6.5         3            5.8         2.2 virginica
## 6          7.6         3            6.6         2.1 virginica
## # ... with 44 more rows
```

`dplyr::filter`

You can put multiple conditions, for instance:

- *filter* rows based on the values of some columns (predicates), for instance:
    - *filter* rows of `data_work` of species "virginica" and with their Width of Petal smaller than 2

dplyr::filter

You can put multiple conditions, for instance:

- *filter* rows based on the values of some columns (predicates), for instance:
  - *filter* rows of `data_work` of species "virginica" and with their Width of Petal smaller than 2

```
filter(data_work, Species == "virginica", Petal.Width < 2)
```

```
## # A tibble: 21 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.8         2.7          5.1         1.9 virginica
## 2          6.3         2.9          5.6         1.8 virginica
## 3          4.9         2.5          4.5         1.7 virginica
## 4          7.3         2.9          6.3         1.8 virginica
## 5          6.7         2.5          5.8         1.8 virginica
## 6          6.4         2.7          5.3         1.9 virginica
## # ... with 15 more rows
```

> Again, it looks like the natural language !

> Again, it looks like the natural language !
>
> *That's one of the nicer things in the `dplyr` syntax.*

dplyr::mutate

dplyr::mutate

`mutate` is the verb used to create new columns.

dplyr::mutate

`mutate` is the verb used to create new columns.

For instance, suppose we want to compute the sum of the lengths of the Sepal and the Petal in our dataset.

dplyr::mutate

`mutate` is the verb used to create new columns.

For instance, suppose we want to compute the sum of the lengths of the Sepal and the Petal in our dataset.

```
mutate(data_work, sum_lengths = Sepal.Length + Petal.Length)
```

```
## # A tibble: 150 x 6
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species sum_le
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `arrange`: `arrange(data_work, Species, desc(Petal.Length))`

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `arrange`: `arrange(data_work, Species, desc(Petal.Length))`

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          4.8         3.4          1.9         0.2 setosa
## 2          5.1         3.8          1.9         0.4 setosa
## 3          5.4         3.9          1.7         0.4 setosa
## 4          5.7         3.8          1.7         0.3 setosa
## 5          5.4         3.4          1.7         0.2 setosa
## 6          5.1         3.3          1.7         0.5 setosa
## # ... with 144 more rows
```

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `distinct`: `distinct(data_work, Species)`

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `distinct`: `distinct(data_work, Species)`

```
## # A tibble: 3 x 1
##   Species
##   <chr>
## 1 setosa
## 2 versicolor
## 3 virginica
```

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `rename`: `rename(data_work, S.Width = Sepal.Width, S.Length = Sepal.Length)`

`dplyr`: other useful functions

`dplyr` provides many useful functions. You can guess their purposes just by their name:

- `rename`: `rename(data_work, S.Width = Sepal.Width, S.Length = Sepal.Length)`

```
## # A tibble: 150 x 5
##   S.Length S.Width Petal.Length Petal.Width Species
##      <dbl>   <dbl>        <dbl>       <dbl> <chr>
## 1      5.1     3.5          1.4         0.2 setosa
## 2      4.9     3            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       3.6          1.4         0.2 setosa
## 6      5.4     3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Is everything clear ?


Break Time

~ 15 minutes

Chain commands using %>% (pipe) operator

The %>% (pronounce pipe) provides a convenient way to code, as it allows the code to be written in chain.

Chain commands using %>% (pipe) operator

The %>% (pronounce pipe) provides a convenient way to code, as it allows the code to be written in chain.

> **IMPORTANT**: the keyboard shortcut for %>% is *ctrl* + *shift* + *M*

Chain commands using %>% (pipe) operator

The %>% (pronounce pipe) provides a convenient way to code, as it allows the code to be written in chain.

> **IMPORTANT**: the keyboard shortcut for %>% is *ctrl* + *shift* + *M*
>
> Try it !

For instance, suppose we want to:

1. *filter* rows of `data_work` of species "virginica" and with their Width of Petal smaller than 2
2. *then* compute the sum of the lengths of the Sepal and the Petal in our dataset.
3. *then* select the columns with their name starting with an S
4. *then* arrange the result by length of Sepal.Length

We would write

For instance, suppose we want to:

1. *filter* rows of `data_work` of species "virginica" and with their Width of Petal smaller than 2
2. *then* compute the sum of the lengths of the Sepal and the Petal in our dataset.
3. *then* select the columns with their name starting with an S
4. *then* arrange the result by length of Sepal.Length

We would write

```
arrange(select(mutate(filter(data_work, Species == 'virginica', Petal
```

For instance, suppose we want to:

1. *filter* rows of `data_work` of species "virginica" and with their Width of Petal smaller than 2
2. *then* compute the sum of the lengths of the Sepal and the Petal in our dataset.
3. *then* select the columns with their name starting with an S
4. *then* arrange the result by length of Sepal.Length

We would write

```
arrange(select(mutate(filter(data_work, Species == 'virginica', Petal
```

> Isn't it unreadable ?!

Let's write it using the %>% operator:

Let's write it using the %>% operator:

```
data_work %>%
  filter(Species == 'virginica', Petal.Width < 2) %>%
  mutate(Sum_lengths = Sepal.Length + Petal.Length) %>%
  select(starts_with("S")) %>%
  arrange(Sepal.Length)
```

Let's write it using the %>% operator:

```
data_work %>%
  filter(Species == 'virginica', Petal.Width < 2) %>%
  mutate(Sum_lengths = Sepal.Length + Petal.Length) %>%
  select(starts_with("S")) %>%
  arrange(Sepal.Length)
```

You see how clearer it looks ?

Let's write it using the %>% operator:

```
data_work %>%
  filter(Species == 'virginica', Petal.Width < 2) %>%
  mutate(Sum_lengths = Sepal.Length + Petal.Length) %>%
  select(starts_with("S")) %>%
  arrange(Sepal.Length)
```

You see how clearer it looks ?

If I run this: `x %>% sum` it is strictly equivalent to `sum(x)`.

Let's write it using the %>% operator:

```
data_work %>%
  filter(Species == 'virginica', Petal.Width < 2) %>%
  mutate(Sum_lengths = Sepal.Length + Petal.Length) %>%
  select(starts_with("S")) %>%
  arrange(Sepal.Length)
```

You see how clearer it looks ?

If I run this: `x %>% sum` it is strictly equivalent to `sum(x)`.

It means: take `x` and pass it through the function `sum`.

Another example to see the power of %>%. Suppose I want to carry out the following steps:

Another example to see the power of %>%. Suppose I want to carry out the following steps:

1. Take `data_work`
2. Select variables containing "Sepal", and "Petal.Width" and "Species"
3. Filter rows with length of Sepal greater than 5
4. Fit a linear model of Petal.Width vs Sepal.Width + Sepal.Length + Species
5. Print a summary of the model

Another example to see the power of %>%. Suppose I want to carry out the following steps:

1. Take `data_work`
2. Select variables containing "Sepal", and "Petal.Width" and "Species"
3. Filter rows with length of Sepal greater than 5
4. Fit a linear model of Petal.Width vs Sepal.Width + Sepal.Length + Species
5. Print a summary of the model

```
data_work %>% #Step 1
  select(contains("Sepal") ,
         Petal.Width, Species) %>% # Step 2
  filter(Sepal.Length >5) %>% # Step 3
  lm(Petal.Width ~ Sepal.Width + Sepal.Length + Species,
     data= .) %>% # Step 4: NOTE THE .
  summary # Step 5
```

```
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Width + Sepal.Length + Species,
##     data = .)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.48660 -0.10718 -0.00351  0.12237  0.46503
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)        -1.14888    0.24851  -4.623 1.01e-05 ***
```

```
data_work %>% #Step 1
  select(contains("Sepal") ,
         Petal.Width, Species) %>% # Step 2
  filter(Sepal.Length > 5) %>% # Step 3
  lm(Petal.Width ~ Sepal.Width + Sepal.Length + Species,
     data= .) %>% # Step 4: NOTE THE .
  summary # Step 5
```

```r
data_work %>% #Step 1
  select(contains("Sepal") ,
         Petal.Width, Species) %>% # Step 2
  filter(Sepal.Length > 5) %>% # Step 3
  lm(Petal.Width ~ Sepal.Width + Sepal.Length + Species,
     data= .) %>% # Step 4: NOTE THE .
  summary # Step 5
```

In this example, it is also important to notice the `.` . When using the pipe, the "." is the object refering to what's before the last %>% .

```
data_work %>% #Step 1
  select(contains("Sepal") ,
         Petal.Width, Species) %>% # Step 2
  filter(Sepal.Length > 5) %>% # Step 3
  lm(Petal.Width ~ Sepal.Width + Sepal.Length + Species,
     data= .) %>% # Step 4: NOTE THE .
  summary # Step 5
```

In this example, it is also important to notice the `.` . When using the pipe, the "." is the object refering to what's before the last %>% .

It is important to specify it when the argument that needs the object before the last %>% is not the first argument.That's why we had to specify it in the `lm` function and not in the `select` function.

Group operations

An important features of `dplyr` is its ability to *group* tibbles and compute operations on these *grouped* tibbles.

Group operations

An important features of `dplyr` is its ability to *group* tibbles and compute operations on these *grouped* tibbles.

A *key* function of `dplyr` is `group_by`.

dplyr::group_by

dplyr::group_by

```r
data_work_by_species <- data_work %>%
  group_by(Species)
# Equivalent to data_work_by_species <- group_by(data_work, Species)

data_work_by_species
```

```
## # A tibble: 150 x 5
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

dplyr::group_by

```r
data_work_by_species <- data_work %>%
  group_by(Species)
# Equivalent to data_work_by_species <- group_by(data_work, Species)

data_work_by_species
```

```
## # A tibble: 150 x 5
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Note the `# Groups: Species [3]`. It means that operations on this dataset will be done for each group.

For example, suppose we want to compute the median of the width of the Sepal for each species.

```
data_work_by_species %>%
  mutate(median_sepal_width = median(Sepal.Width)) %>%
  select(starts_with("S"), median_sepal_width)
```

```
## # A tibble: 150 x 4
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Species median_sepal_width
##          <dbl>       <dbl> <chr>                <dbl>
## 1          5.1         3.5 setosa                 3.4
## 2          4.9         3   setosa                 3.4
## 3          4.7         3.2 setosa                 3.4
## 4          4.6         3.1 setosa                 3.4
## 5          5           3.6 setosa                 3.4
## 6          5.4         3.9 setosa                 3.4
## # ... with 144 more rows
```

It's nice, but we may also need to summarise the table, just keep a summary of the Species and the median.

`dplyr::summarise`

It is easily done by the function `summarise`

`dplyr::summarise`

It is easily done by the function `summarise`

```
data_work_by_species %>%
  summarise(median_sepal_width = median(Sepal.Width))
```

```
## # A tibble: 3 x 2
##   Species    median_sepal_width
##   <chr>                   <dbl>
## 1 setosa                    3.4
## 2 versicolor                2.8
## 3 virginica                 3
```

If you want to take out the grouped structure of your tibble, you just have to use the function `ungroup`

```
data_work_by_species %>% ungroup
```

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Functions to join tables

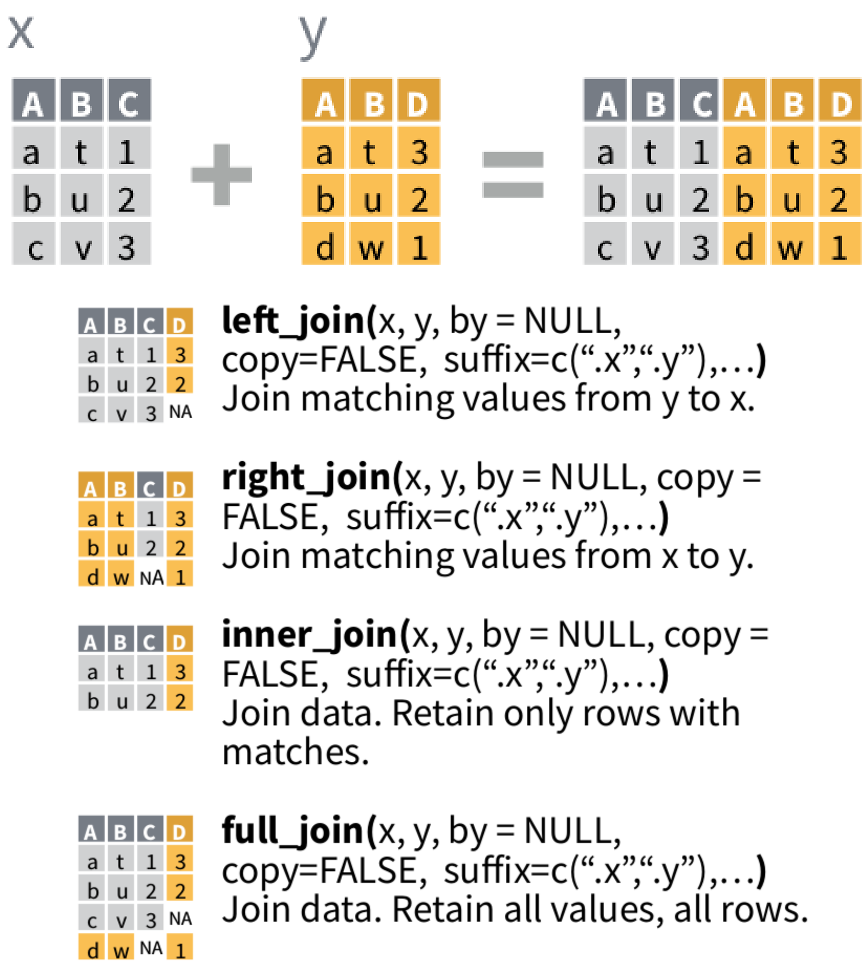


Image from RStudio data transformation cheatsheet

Since `dplyr` 1.0: function `dplyr::across`

Since `dplyr` 1.0: function `dplyr::across`

To help apply a function over multiple columns, `dplyr` came with the function `across`

Since `dplyr` 1.0: function `dplyr::across`

To help apply a function over multiple columns, `dplyr` came with the function `across`

General syntax is: `across(.cols = THE COLUMNS ON WHICH YOU WANT TO APPLY THE FUNCTION(S), .fns = THE FUNCTION YOU WANT TO APPLY, ...)`

Since `dplyr` 1.0: function `dplyr::across`

To help apply a function over multiple columns, `dplyr` came with the function `across`

General syntax is: `across(.cols = THE COLUMNS ON WHICH YOU WANT TO APPLY THE FUNCTION(S), .fns = THE FUNCTION YOU WANT TO APPLY, ...)`

See `?dplyr::across` for more details and further arguments

Combined with the `where` function, it allows to apply a function on specific columns

```
data_work %>%
  mutate(across(where(is.character), as.factor)) # apply function as
```

```
## # A tibble: 150 x 5
##   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          4.7         3.2          1.3         0.2 setosa
## 4          4.6         3.1          1.5         0.2 setosa
## 5          5           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

You can specify the name and a specific functions using the `~.x` syntax. `.names` is used to specify the names of the new columns. It has a specific syntax (see `?dplyr::across`).

You can specify the name and a specific functions using the `~.x` syntax. `.names` is used to specify the names of the new columns. It has a specific syntax (see `?dplyr::across`).

```
data_work %>%
  mutate(across(where(is.numeric), ~ .x*0.01, .names = "{.col}_in_met
```

```
## # A tibble: 150 x 9
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal.
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            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           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows, and 3 more variables: Sepal.Width_in_mete
## #   Petal.Length_in_meters <dbl>, Petal.Width_in_meters <dbl>
```

Do you have questions ?

Exercises

~ 20 minutes

Exercises

~ 20 minutes

Open "Manipulate_and_tidy_your_data_exercises.Rmd" with Rstudio

Exercises

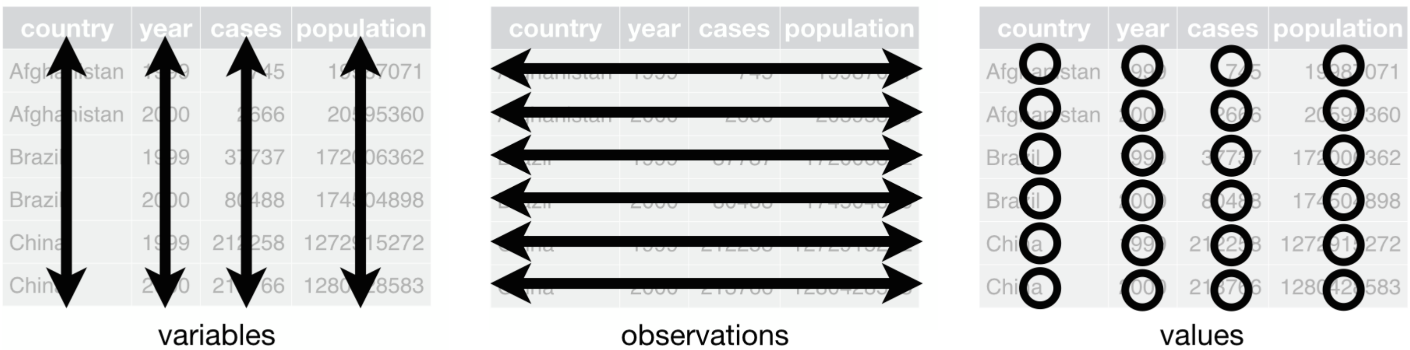
~ 20 minutes

Open "Manipulate_and_tidy_your_data_exercises.Rmd" with Rstudio

Answer to the questions until section "Tidy your data"

Tidy data

1. Each variable must have its own column
2. Each observation must have its row
3. Each value must have its own cell



For instance, imagine this dataset, giving the population of different countries in 2002 and 2007:

```
d %>% head
```

```
## # A tibble: 6 x 3
##   country         2002    2007
##   <chr>          <dbl>   <dbl>
## 1 Belgium     10311970 10392226
## 2 France      59925035 61083916
## 3 Germany     82350671 82400996
## 4 Italy       57926999 58147733
## 5 Spain       40152517 40448191
## 6 Switzerland  7361757  7554661
```

For instance, imagine this dataset, giving the population of different countries in 2002 and 2007:

```
d %>% head
```

```
## # A tibble: 6 x 3
##   country         `2002`   `2007`
##   <chr>            <dbl>    <dbl>
## 1 Belgium     10311970 10392226
## 2 France      59925035 61083916
## 3 Germany     82350671 82400996
## 4 Italy       57926999 58147733
## 5 Spain       40152517 40448191
## 6 Switzerland  7361757  7554661
```

- Is this dataset tidy ?

For instance, imagine this dataset, giving the population of different countries in 2002 and 2007:

```
d %>% head
```

```
## # A tibble: 6 x 3
##   country        `2002`   `2007`
##   <chr>           <dbl>    <dbl>
## 1 Belgium      10311970 10392226
## 2 France       59925035 61083916
## 3 Germany      82350671 82400996
## 4 Italy        57926999 58147733
## 5 Spain        40152517 40448191
## 6 Switzerland   7361757  7554661
```

- Is this dataset tidy ?

This is dataset is **not** tidy, as the population, which is an observed variable is not in a distinct column (principle 1.). Year is also a variable, so it should have its column too.

Instead, we should have:

```
## # A tibble: 6 x 3
##   country year  population
##   <chr>   <chr>      <dbl>
## 1 Belgium 2002    10311970
## 2 Belgium 2007    10392226
## 3 France  2002    59925035
## 4 France  2007    61083916
## 5 Germany 2002    82350671
## 6 Germany 2007    82400996
```

Make tidy data

Two key functions, of package `tidyr` are used to tidy the data:

Make tidy data

Two key functions, of package `tidyr` are used to tidy the data:

- `tidyr::pivot_longer` is used to make your dataset *longer* (what a surprise ! :O)
- `tidyr::pivot_wider` is used to make your dataset *wider* (what a surprise ! :O)

In practice:

```r
#d is the dataset with the populations of the countries
data_tidy <- d %>% pivot_longer(cols =  c("2002","2007"))

head(data_tidy)
```

```
## # A tibble: 6 x 3
##   country name     value
##   <chr>   <chr>    <dbl>
## 1 Belgium 2002  10311970
## 2 Belgium 2007  10392226
## 3 France  2002  59925035
## 4 France  2007  61083916
## 5 Germany 2002  82350671
## 6 Germany 2007  82400996
```

In practice:

```r
#d is the dataset with the populations of the countries
data_tidy <- d %>% pivot_longer(cols =  c("2002","2007"))

head(data_tidy)
```

```
## # A tibble: 6 x 3
##   country name     value
##   <chr>   <chr>    <dbl>
## 1 Belgium 2002  10311970
## 2 Belgium 2007  10392226
## 3 France  2002  59925035
## 4 France  2007  61083916
## 5 Germany 2002  82350671
## 6 Germany 2007  82400996
```

The first argument is the dataset to tidy (which is not necessary to complete because of the %>%).

In practice:

```r
#d is the dataset with the populations of the countries
data_tidy <- d %>% pivot_longer(cols =  c("2002","2007"))

head(data_tidy)
```

```
## # A tibble: 6 x 3
##   country name     value
##   <chr>   <chr>    <dbl>
## 1 Belgium 2002  10311970
## 2 Belgium 2007  10392226
## 3 France  2002  59925035
## 4 France  2007  61083916
## 5 Germany 2002  82350671
## 6 Germany 2007  82400996
```

The first argument is the dataset to tidy (which is not necessary to complete because of the %>%).

The second is the name of the columns to gather.

We can also provide another names to the new columns using the arguments `values_to` and `names_to`.

We can also provide another names to the new columns using the arguments `values_to` and `names_to`.

The columns to gather can also be selected using the select helpers that we've seen previously:

We can also provide another names to the new columns using the arguments `values_to` and `names_to`.

The columns to gather can also be selected using the select helpers that we've seen previously:

```
#d is the dataset with the populations of the countries
data_tidy <- d %>%
  pivot_longer(cols =  -country,
               names_to = "Year", # New name
               #for the columns 2002 and 2007
               values_to = "Population")
    #New name for the values

head(data_tidy)
```

```
## # A tibble: 6 x 3
##   country Year  Population
##   <chr>   <chr>      <dbl>
## 1 Belgium 2002    10311970
## 2 Belgium 2007    10392226
## 3 France  2002    59925035
## 4 France  2007    61083916
## 5 Germany 2002    82350671
## 6 Germany 2007    82400996
```

Conversely, `pivot_wider` allows us to come back to the first dataset.

Conversely, `pivot_wider` allows us to come back to the first dataset.

```
data_tidy %>% pivot_wider(names_from = Year, values_from = Populatior
```

```
## # A tibble: 6 x 3
##   country        `2002`   `2007`
##   <chr>           <dbl>    <dbl>
## 1 Belgium      10311970 10392226
## 2 France       59925035 61083916
## 3 Germany      82350671 82400996
## 4 Italy        57926999 58147733
## 5 Spain        40152517 40448191
## 6 Switzerland   7361757  7554661
```

Is everything clear ?

Exercises

Exercises

~ 20 minutes

Exercises

~ 20 minutes

Answer to the questions of the section "Tidy your data"

Conclusion

The `tidyverse` provides many tools to work with data.

Conclusion

The `tidyverse` provides many tools to work with data.

Many topics have not been presented today:

- manipulate factors using `forcats`
- manipulate dates using `lubridate`
- manipulate dates using `stringr`
- ...

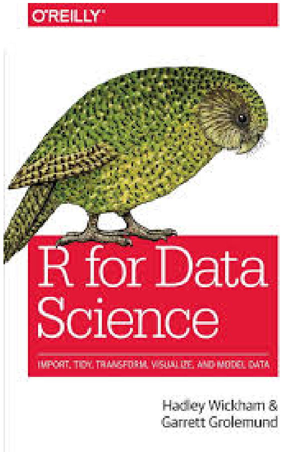
Conclusion

The `tidyverse` provides many tools to work with data.

Many topics have not been presented today:

- manipulate factors using `forcats`
- manipulate dates using `lubridate`
- manipulate dates using `stringr`
- ...

Feel free to consult this book (available for free online at this adress: https://r4ds.had.co.nz/):



THANKS

THANKS

Any remark, questions ?

THANKS

Any remark, questions ?

remi.mahmoud@inrae.fr

THANKS

Any remark, questions ?

remi.mahmoud@inrae.fr

Lesson contents available at:
https://github.com/RemiMahmoud/data_wrangling