Manipulate your data using dplyr

Rémi Mahmoud

Who is talking ?

INRAE

What you have to do:

Open Rstudio

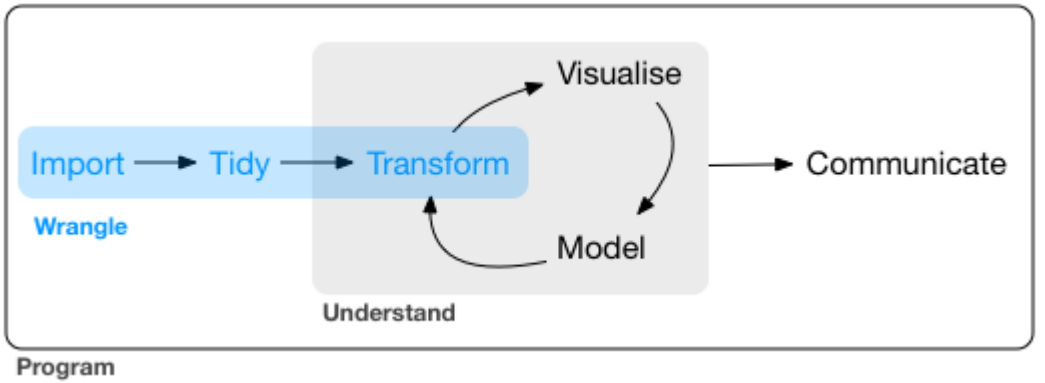
What you have to do:

Open Rstudio

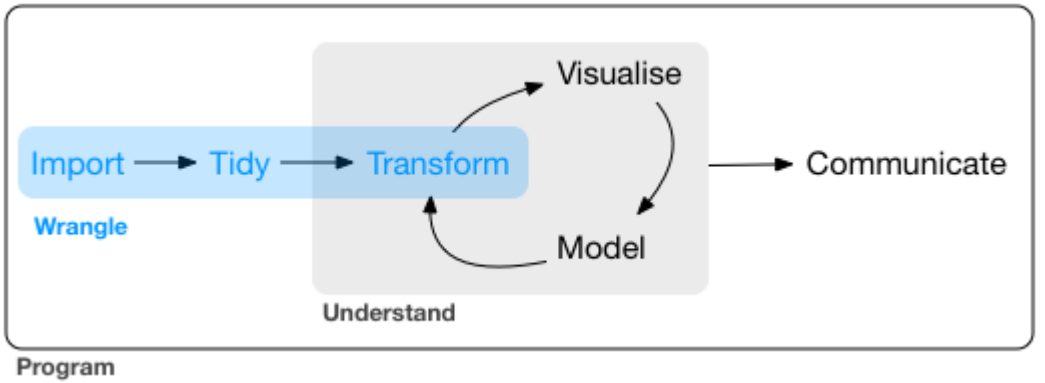
Stop me if anything is unclear

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.

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**

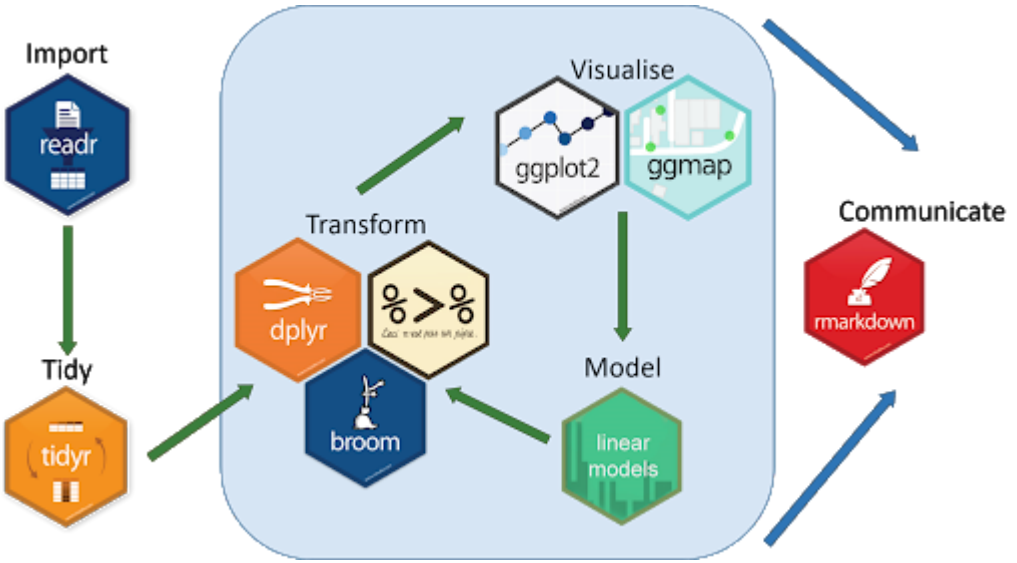
```
install.packages("tidyverse")
```

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



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 ?

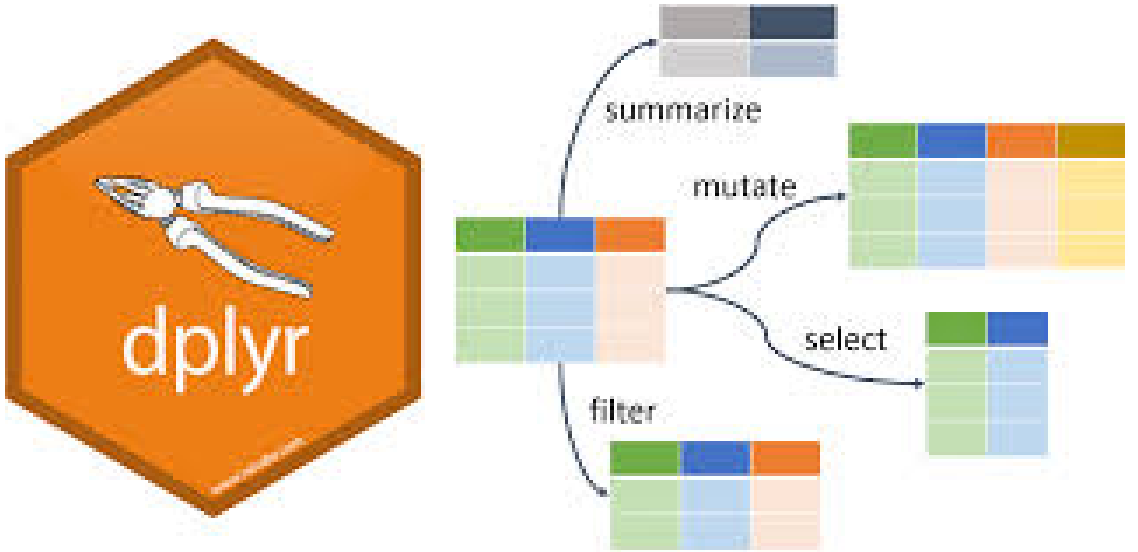
Select columns, filter their rows, create new columns etc.

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.



Tibble vs dataframe

Let us take a the well-known iris dataset, turned into a *tibble*.

```
data_work <- as_tibble(iris)

head(data_work)
```

```
## # A tibble: 6 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
```

Tibble vs dataframe

Let us take a the well-known iris dataset, turned into a *tibble*.

```
data_work <- as_tibble(iris)

head(data_work)
```

```
## # A tibble: 6 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
```

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
```

Let us consider the dataset `data_work`.

```
data_work
```

```
## # 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
```

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> <fct>
## 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> <fct>
## 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> <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
```

`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> <fct>
## 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> <fct>
## 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> <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
```

`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> <fct>
## 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
##   <fct>
## 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> <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
```

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
```

```
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

```
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> <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
```

dplyr::group_by

```
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> <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
```

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> <fct>                 <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))
```

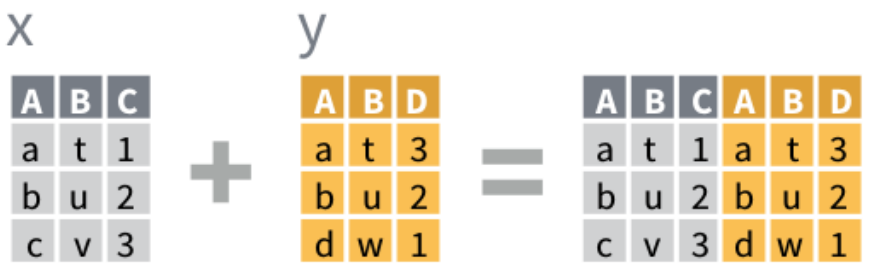
```
## # A tibble: 3 x 2
##   Species    median_sepal_width
##   <fct>                   <dbl>
## 1 setosa                    3.4
## 2 versicolor                2.8
## 3 virginica                 3
```

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
```

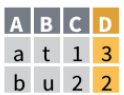
```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## 3          4.7         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
```

Functions to join tables



left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…**)**
Join matching values from y to x.

right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…**)**
Join matching values from x to y.



inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…**)**
Join data. Retain only rows with matches.

full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…**)**
Join data. Retain all values, all rows.

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.L
##          <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, and 3 more variables: Sepal.Width_in_mete
## #   Petal.Length_in_meters <dbl>, Petal.Width_in_meters <dbl>
```

Do you have questions ?

Exercises

~ 45 minutes

Exercises

~ 45 minutes

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

Exercises

~ 45 minutes

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

Answer to the questions

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`
- ...

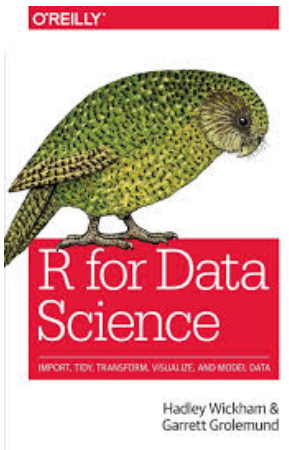
Conclusion

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