R pour l'analyse de données

Pierre Gloaguen

November, 2020

Contents

Prérequis supposés

Prerequis

- Savoir lire l'anglais
- Bases de `R`
  - Créer des objets
  - Opérations de base
- Objets de base en `R`
  - Vecteurs
  - Listes
  - Data.frame
- Fonctions de bases
- Import de données

Références utiles

- R for Data Science (*G. Grolemund and H. Wickham*)
- R Markdown: The definitive guide (*Y. Xie, J.J. Allaire and G. Grolemund*)
- **ggplot2: Elegant Graphics for Data Analysis** (*H. Wickham*)

Les deux premiers sont disponible en ligne

Tidyverse

What is tidyverse?

- Set of packages for tidy and unified data processing, visualization and modelling.
- Consists in a set of method having unified structure.
- Involves slight modifications in classical `R` grammar.
- This grammar becomes dominant, and highly documented.

Loading tidyverse

```
library(tidyverse)
```

- Load a lot of packages
- Some of them are often updated! **Watch for updates!**

The `tibble` format

`tibble`, a nicer `data.frame`

- The `tibble` is a "modern" version of the `data.frame`

```
iris_tibble <- as_tibble(iris) # Natural conversion
```

- The printing is naturally cropped, giving types, number of rows, cols

```
iris_tibble
```

```
# 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
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
# ... with 140 more rows
```

Creating a tibble

The function `tibble` creates naturally a tibble.

```
my_tibble <- tibble(Nom = c("Alice", "Bob", "Claire"),
                    Age = c(10, 25, 30))
```

It can be neatly printed with `knitr::kable`, like a data.frame

```
my_tibble
```

```
# A tibble: 3 x 2
  Nom      Age
  <chr>  <dbl>
1 Alice     10
2 Bob       25
3 Claire    30
```

Extracting in tibble

Exactly as in `data.frame`

```
my_tibble[1] # Returns a tibble
my_tibble[, "Nom"]# Returns a tibble. Equivalent to my_tibble["Nom"]
# For a data.frame, this last line would return a vector, not a data.frame
```

To extract columns, the `pull` function is introduced (also works with `data.frame`).

```
dplyr::pull(my_tibble, Nom)# The pull function extracts a column to a vector
```

```
[1] "Alice"  "Bob"    "Claire"
```

```
my_tibble$Nom # But the old fashion way still works
```

```
[1] "Alice"  "Bob"    "Claire"
```

Row names in `tibble`

For programming reasons, `tibble` can't have row names

```
head(swiss[, 1:3], n = 2)
```

```
           Fertility Agriculture Examination
Courtelary      80.2        17.0          15
Delemont        83.1        45.1           6
```

```
print(as_tibble(swiss[, 1:3]), n = 2)
```

```
# A tibble: 47 x 3
  Fertility Agriculture Examination
      <dbl>       <dbl>       <int>
1      80.2        17            15
2      83.1        45.1           6
# ... with 45 more rows
```

```
print(as_tibble(swiss[, 1:3], rownames = "Province"), n = 2) # Creating a new column
```

```
# A tibble: 47 x 4
  Province   Fertility Agriculture Examination
  <chr>          <dbl>       <dbl>       <int>
1 Courtelary      80.2        17            15
2 Delemont        83.1        45.1           6
# ... with 45 more rows
```

%>% For sequential data processing

diamonds data set

First, we load a toy data set

```
data(diamonds) # Load the diamond data set
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
```

```
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Example of sequential processing

- Imagine we want to compute the mean (omitting `NA`s) price of diamonds having a `Good` cut and a color of type `D`.

There is the one-line way, rather cumbersome

```
mean(diamonds[diamonds$cut == "Good" & diamonds$color == "D", ]$price, na.rm = T)
```

```
[1] 3405.382
```

There is the way in multiple lines, creating intermediary objects

```
condition <- diamonds$cut == "Good" & diamonds$color == "D"
sub_prices <- diamonds[condition, ]$price # could have use pull
mean(sub_prices, na.rm = T)
```

Tidier sequential processing using %>%

The "pipe" instruction `%>%` allows to write this **sequential** instruction in an *easy to read* way, *without creating intermediary objects*.

- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`
- When you read code, `%>%` is pronounced "then"

```
# Same example as before
diamonds %>% # We take the data, then
  dplyr::filter(cut == "Good", color == "D") %>% # Subsetting (using dplyr)
  dplyr::pull(price) %>% # extract the price
  mean(na.rm = T) # compute the mean, omitting the NAs
```

```
[1] 3405.382
```

- The result of the previous treatment is set as the first default argument of the next function
- No redundancy of the diamonds using `$`
- The keyboard shortcut in `Rstudio` for `%>%` is `Ctrl + Maj + M`
- Note that the indenting naturally allows commenting

More about the %>%

If you need to specify specifically which argument you want your input to go to , you can use the dot "."

```
# Adjusting a linear regression on diamonds price w.r.t. carat
diamonds %>%
  lm(data = ., formula = price ~ carat) # The "." refers to entering argument
```

Some advices (from the bible of tidyverse)

- Use `%>%` to emphasize a sequence of actions, rather than the object that the actions are being performed on.

- Avoid using the pipe when:
  - You need to manipulate more than one object at a time. Reserve pipes for a sequence of steps applied to one primary object.
  - There are meaningful intermediate objects that could be given informative names.

Manipulating data

The `dplyr` package

```
library(dplyr)
data(diamonds, package = "ggplot2") # data set used
```

`dplyr` is a package (part of `tidyverse`) which allows you to solve the vast majority of your data-manipulation challenge:

- create variables
- pick variables
- reorder observations
- pick observations
- create summaries
- ...

Functions in this package are verbs and have consistent structures.

Extracting lines on number: `slice()`

- You can extract lines (or **slicing** the data) using `slice()`
- The results remains a `tibble` (or a `data.frame`)

```
slice(diamonds, # Table in which we select
      c(3, 8)) # Select lines 3 and 8
```

```
# A tibble: 2 x 10
  carat cut       color clarity depth table price     x     y     z
  <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
2  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
```

Conditional extraction: `filter()`

- `dplyr::filter` allows to return rows with matching conditions

```
dplyr::filter(diamonds, #  Table in which we select
              color == "D", clarity == "SI1")# the , is equivalent to an "&"
```

```
# A tibble: 2,083 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.3  Premium   D     SI1      62.6    59   552  4.23  4.27  2.66
 2  0.3  Ideal     D     SI1      62.5    57   552  4.29  4.32  2.69
 3  0.3  Ideal     D     SI1      62.1    56   552  4.3   4.33  2.68
 4  0.75 Very Good D     SI1      63.2    56  2760  5.8   5.75  3.65
 5  0.71 Very Good D     SI1      63.6    58  2764  5.64  5.68  3.6
```

```
 6  0.71 Ideal     D     SI1      61.9    59  2764  5.69  5.72  3.53
 7  0.73 Very Good D     SI1      60.2    56  2768  5.83  5.87  3.52
 8  0.7  Very Good D     SI1      61.1    58  2768  5.66  5.73  3.48
 9  0.72 Ideal     D     SI1      60.8    57  2782  5.76  5.75  3.5
10  0.72 Premium   D     SI1      62.7    59  2782  5.73  5.69  3.58
# ... with 2,073 more rows
```

```
# Equivalent to the one before
dplyr::filter(diamonds,
              color == "D" & clarity == "SI1")
```

```
# A tibble: 2,083 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.3  Premium   D     SI1      62.6    59   552  4.23  4.27  2.66
 2  0.3  Ideal     D     SI1      62.5    57   552  4.29  4.32  2.69
 3  0.3  Ideal     D     SI1      62.1    56   552  4.3   4.33  2.68
 4  0.75 Very Good D     SI1      63.2    56  2760  5.8   5.75  3.65
 5  0.71 Very Good D     SI1      63.6    58  2764  5.64  5.68  3.6
 6  0.71 Ideal     D     SI1      61.9    59  2764  5.69  5.72  3.53
 7  0.73 Very Good D     SI1      60.2    56  2768  5.83  5.87  3.52
 8  0.7  Very Good D     SI1      61.1    58  2768  5.66  5.73  3.48
 9  0.72 Ideal     D     SI1      60.8    57  2782  5.76  5.75  3.5
10  0.72 Premium   D     SI1      62.7    59  2782  5.73  5.69  3.58
# ... with 2,073 more rows
```

```
# One with a "or" (using the in)
dplyr::filter(diamonds,
              color %in% c("D", "E"), clarity == "SI1")
```

```
# A tibble: 4,509 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 2  0.3  Premium   D     SI1      62.6    59   552  4.23  4.27  2.66
 3  0.3  Ideal     D     SI1      62.5    57   552  4.29  4.32  2.69
 4  0.3  Ideal     D     SI1      62.1    56   552  4.3   4.33  2.68
 5  0.7  Ideal     E     SI1      62.5    57  2757  5.7   5.72  3.57
 6  0.73 Very Good E     SI1      61.6    59  2760  5.77  5.78  3.56
 7  0.75 Very Good D     SI1      63.2    56  2760  5.8   5.75  3.65
 8  0.75 Premium   E     SI1      59.9    54  2760  6     5.96  3.58
 9  0.74 Ideal     E     SI1      62.3    54  2762  5.8   5.83  3.62
10  0.71 Very Good D     SI1      63.6    58  2764  5.64  5.68  3.6
# ... with 4,499 more rows
```

Choosing among columns `select()`

Selecting on column number

- The basic `select` works on column number.

```
select(diamonds, # Data in which we select columns
       1, 4) # The position of the select column (1st and 4th)
```

```
# A tibble: 53,940 x 2
   carat clarity
   <dbl> <ord>
```

```
 1  0.23 SI2
 2  0.21 SI1
 3  0.23 VS1
 4  0.29 VS2
 5  0.31 SI2
 6  0.24 VVS2
 7  0.24 VVS1
 8  0.26 SI1
 9  0.22 VS2
10  0.23 VS1
# ... with 53,930 more rows
```

Selecting on names

- The basic `select` works on column names.

```
select(diamonds, # dat in which we select
       carat, cut, depth) # Column names (no need for "")
```

```
# A tibble: 53,940 x 3
   carat cut       depth
   <dbl> <ord>     <dbl>
 1  0.23 Ideal      61.5
 2  0.21 Premium    59.8
 3  0.23 Good       56.9
 4  0.29 Premium    62.4
 5  0.31 Good       63.3
 6  0.24 Very Good  62.8
 7  0.24 Very Good  62.3
 8  0.26 Very Good  61.9
 9  0.22 Fair       65.1
10  0.23 Very Good  59.4
# ... with 53,930 more rows
```

Removing column

One can remove column using `select`, by adding `-` in front of the column name:

```
select(diamonds, # Data in which we select
       -carat, -cut) # Select all columms but carat and cut
```

```
# A tibble: 53,940 x 8
   color clarity depth table price     x     y     z
   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1 E     SI2      61.5    55   326  3.95  3.98  2.43
 2 E     SI1      59.8    61   326  3.89  3.84  2.31
 3 E     VS1      56.9    65   327  4.05  4.07  2.31
 4 I     VS2      62.4    58   334  4.2   4.23  2.63
 5 J     SI2      63.3    58   335  4.34  4.35  2.75
 6 J     VVS2     62.8    57   336  3.94  3.96  2.48
 7 I     VVS1     62.3    57   336  3.95  3.98  2.47
 8 H     SI1      61.9    55   337  4.07  4.11  2.53
 9 E     VS2      65.1    61   337  3.87  3.78  2.49
10 H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Selecting on a condition `select_if()`

- One can select on column condition with `select_if`
- Put in argument a function on column that returns a TRUE/FALSE

```
select_if(diamonds, # Data in which we select
          .predicate = is.factor) # Select only factor columns, the .predicate is optional
```

```
# A tibble: 53,940 x 3
   cut       color clarity
   <ord>     <ord> <ord>
 1 Ideal     E     SI2
 2 Premium   E     SI1
 3 Good      E     VS1
 4 Premium   I     VS2
 5 Good      J     SI2
 6 Very Good J     VVS2
 7 Very Good I     VVS1
 8 Very Good H     SI1
 9 Fair      E     VS2
10 Very Good H     VS1
# ... with 53,930 more rows
```

```
select_if(diamonds, # Data in which we select
          is.numeric) # Select only numeric columns
```

```
# A tibble: 53,940 x 7
   carat depth table price     x     y     z
   <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23  61.5    55   326  3.95  3.98  2.43
 2  0.21  59.8    61   326  3.89  3.84  2.31
 3  0.23  56.9    65   327  4.05  4.07  2.31
 4  0.29  62.4    58   334  4.2   4.23  2.63
 5  0.31  63.3    58   335  4.34  4.35  2.75
 6  0.24  62.8    57   336  3.94  3.96  2.48
 7  0.24  62.3    57   336  3.95  3.98  2.47
 8  0.26  61.9    55   337  4.07  4.11  2.53
 9  0.22  65.1    61   337  3.87  3.78  2.49
10  0.23  59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Selecting on a characteristic `select_at()`

- One can select on position using `select_at`. It uses the `dplyr` function `vars` that allows to enclose text expressions.

```
select_at(diamonds, # Data in which we select
  .vars = vars(starts_with("c"))) # Select columns starting with "c"
```

```
# A tibble: 53,940 x 4
   carat cut       color clarity
   <dbl> <ord>     <ord> <ord>
 1  0.23 Ideal     E     SI2
 2  0.21 Premium   E     SI1
 3  0.23 Good      E     VS1
 4  0.29 Premium   I     VS2
 5  0.31 Good      J     SI2
```

```
 6  0.24 Very Good J     VVS2
 7  0.24 Very Good I     VVS1
 8  0.26 Very Good H     SI1
 9  0.22 Fair      E     VS2
10  0.23 Very Good H     VS1
# ... with 53,930 more rows
```

Extracting columns with `pull()`

- Function to extract a column (by its name or number) **as a vector**

```
pull(diamonds, # Data from which we pull a column
     cut) %>%  # extract the cut variable by name as a vector
  head(n = 5) # only print first 5 values
```

```
[1] Ideal   Premium Good    Premium Good
Levels: Fair < Good < Very Good < Premium < Ideal
```

```
pull(diamonds, # Data from which we pull
     2) %>% # extract the cut by number variable as a vector
  head(n = 5)
```

```
[1] Ideal   Premium Good    Premium Good
Levels: Fair < Good < Very Good < Premium < Ideal
```

Renaming columns `rename()`

Basic renaming

- You might want to rename column(s)

```
rename(diamonds, # Data in which we rename
       length = x, # Always New = old
       width = y) # You can rename multiple columns at once
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price length width     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int>  <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326   3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326   3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327   4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334   4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335   4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336   3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336   3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337   4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337   3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338   4     4.05  2.39
# ... with 53,930 more rows
```

Renaming on condition `rename_if()`

- In the same way, a `rename_if` base on a condition
- Require as argument a function to apply to each selected column.

```
# Add the "_fact" suffix to every factor column name
rename_if(diamonds, # Data in which we rename
          .predicate = is.factor, .funs = function(x) paste0(x, "_fact"))
```

```
# A tibble: 53,940 x 10
   carat cut_fact  color_fact clarity_fact depth table price     x     y     z
   <dbl> <ord>     <ord>      <ord>        <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E          SI2           61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E          SI1           59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E          VS1           56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I          VS2           62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J          SI2           63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J          VVS2          62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I          VVS1          62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H          SI1           61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E          VS2           65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H          VS1           59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Renaming on a characteristic `rename_at()`

- As well as a `rename_at()`
- Require as argument a function to apply to each selected column.

```
# Putting in upper case columns starting with "c"
rename_at(diamonds, # Data in which we rename
          .vars = vars(starts_with("c")), .funs = toupper)
```

```
# A tibble: 53,940 x 10
   CARAT CUT       COLOR CLARITY depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Create or modify a column: `mutate()`

Basic modication

- You can modify a column

```
mutate(diamonds, # Data in which we modify the column
       cut = factor(cut, labels = LETTERS[5:1])) # modifying labels of cut
```

```
# A tibble: 53,940 x 10
   carat cut   color clarity depth table price     x     y     z
   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 A     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 B     E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 D     E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 B     I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 D     J     SI2      63.3    58   335  4.34  4.35  2.75
```

```
 6  0.24 C     J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 C     I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 C     H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 E     E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 C     H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

- The same function also creates columns

```
mutate(diamonds, # Data in which we create the column
       z_square = z^2) # creates a new column as function of an existing one
```

```
# A tibble: 53,940 x 11
   carat cut       color clarity depth table price     x     y     z z_square
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>    <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43     5.90
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31     5.34
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31     5.34
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63     6.92
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75     7.56
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48     6.15
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47     6.10
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53     6.40
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49     6.20
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39     5.71
# ... with 53,930 more rows
```

- You can create/modify multiple columns at once
- This will be done **sequentially!**

```
mutate(diamonds, # Data in which we modify/create columns
       cut = factor(cut, labels = LETTERS[5:1]), # modifying labels of cut
       y = 2 * y, # doubling the width
       z_square = z^2, # creating a column z_square
       cut = z_square) # This modification overrides the first one
```

```
# A tibble: 53,940 x 11
   carat   cut color clarity depth table price     x     y     z z_square
   <dbl> <dbl> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>    <dbl>
 1  0.23  5.90 E     SI2      61.5    55   326  3.95  7.96  2.43     5.90
 2  0.21  5.34 E     SI1      59.8    61   326  3.89  7.68  2.31     5.34
 3  0.23  5.34 E     VS1      56.9    65   327  4.05  8.14  2.31     5.34
 4  0.29  6.92 I     VS2      62.4    58   334  4.2   8.46  2.63     6.92
 5  0.31  7.56 J     SI2      63.3    58   335  4.34  8.7   2.75     7.56
 6  0.24  6.15 J     VVS2     62.8    57   336  3.94  7.92  2.48     6.15
 7  0.24  6.10 I     VVS1     62.3    57   336  3.95  7.96  2.47     6.10
 8  0.26  6.40 H     SI1      61.9    55   337  4.07  8.22  2.53     6.40
 9  0.22  6.20 E     VS2      65.1    61   337  3.87  7.56  2.49     6.20
10  0.23  5.71 H     VS1      59.4    61   338  4     8.1   2.39     5.71
# ... with 53,930 more rows
```

Modifying columns on a condition `mutate_if()`

- Again, you can change on a condition `mutate_if`

```
mutate_if(diamonds, # Data in which we modify column
          is.factor, # Condition on which a column is modified
```

```
          as.character) # Function to apply (transform factor columns to character)
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <chr>     <chr> <chr>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

- Check the type of `cut` `color` and `clarity`!

Modifying columns on a characteristic `mutate_at()`

- And on position, `mutate_at`

```
mutate_at(diamonds, # Data in which we modify the column
          vars(starts_with("c")), # Names of columns modified
          .funs = function(x) 0) # Put columns to 0
```

```
# A tibble: 53,940 x 10
   carat   cut color clarity depth table price     x     y     z
   <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1     0     0     0       0  61.5    55   326  3.95  3.98  2.43
 2     0     0     0       0  59.8    61   326  3.89  3.84  2.31
 3     0     0     0       0  56.9    65   327  4.05  4.07  2.31
 4     0     0     0       0  62.4    58   334  4.2   4.23  2.63
 5     0     0     0       0  63.3    58   335  4.34  4.35  2.75
 6     0     0     0       0  62.8    57   336  3.94  3.96  2.48
 7     0     0     0       0  62.3    57   336  3.95  3.98  2.47
 8     0     0     0       0  61.9    55   337  4.07  4.11  2.53
 9     0     0     0       0  65.1    61   337  3.87  3.78  2.49
10     0     0     0       0  59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Sorting data: `arrange()`

```
arrange(diamonds, # data set on which we sort
        carat, depth) # Sorting on carat, then on depth
```

```
# A tibble: 53,940 x 10
   carat cut     color clarity depth table price     x     y     z
   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1   0.2 Premium E     VS2      59      60   367  3.81  3.78  2.24
 2   0.2 Premium E     VS2      59.7    62   367  3.84  3.8   2.28
 3   0.2 Ideal   E     VS2      59.7    55   367  3.86  3.84  2.3
 4   0.2 Premium E     VS2      59.8    62   367  3.79  3.77  2.26
 5   0.2 Premium E     SI2      60.2    62   345  3.79  3.75  2.27
```

```
 6   0.2 Premium E     VS2      61.1    59   367  3.81  3.78  2.32
 7   0.2 Ideal   D     VS2      61.5    57   367  3.81  3.77  2.33
 8   0.2 Premium D     VS2      61.7    60   367  3.77  3.72  2.31
 9   0.2 Ideal   E     VS2      62.2    57   367  3.76  3.73  2.33
10   0.2 Premium D     VS2      62.3    60   367  3.73  3.68  2.31
# ... with 53,930 more rows
```

```
# Decreasing sort uses the function desc()
arrange(diamonds, # data on which we sort
        desc(carat, depth)) # Decreasing on carat, increasing on depth
```

```
Error in `arrange()`:
! `desc()` must be called with exactly one argument.
```

Summarising data `summarise()`

- Useful to summarize a column to a single number.
- Returns a `tibble`.

```
summarise(diamonds, # Data on which we summarise
          mean_carat = mean(carat), # New name = transformation
          var_carat = var(carat),
          number_diamonds = n(), # dplyr::n() equivalent to nrow(.)
          number_distinct_carats = n_distinct(carat))# ddplyr::n_distinct()
```

```
# A tibble: 1 x 4
  mean_carat var_carat number_diamonds number_distinct_carats
       <dbl>     <dbl>           <int>                  <int>
1      0.798     0.225           53940                    273
```

Summarising columns on condition `summarise_if()`

- Of course, `summarise_if()` and `summarise_at()` exists!

```
# Compute the mean of all numeric variables
summarise_if(diamonds,
             is.numeric, mean)
```

```
# A tibble: 1 x 7
  carat depth table price     x     y     z
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 0.798  61.7  57.5 3933.  5.73  5.73  3.54
```

```
summarise_at(diamonds,
             .vars = vars(starts_with("c")), # For columns starting by c
             # count number of unique elements in the column
             .funs = list(n_unique = function(x) length(unique(x))))
```

```
# A tibble: 1 x 4
  carat_n_unique cut_n_unique color_n_unique clarity_n_unique
           <int>        <int>          <int>            <int>
1            273            5              7                8
```

Grouping data `group_by()`

- Often, you might compute statistics depending on groups.
- `group_by` wiill allow to perform the summarise per factor of a group

```r
diamonds %>%
  group_by(cut) %>% # depending on the cut
  summarise(Count = n(),# Number of diamonds per group
            Mean_price = mean(price),# Mean price per group
            Mean_carat = var(carat))
```

```
# A tibble: 5 x 4
  cut       Count Mean_price Mean_carat
  <ord>     <int>      <dbl>      <dbl>
1 Fair       1610      4359.      0.267
2 Good       4906      3929.      0.206
3 Very Good 12082      3982.      0.211
4 Premium   13791      4584.      0.265
5 Ideal     21551      3458.      0.187
```

- In practice `group_by` changes the way `dplyr` functions operate on the data. They will operate groupwise.

Exercise

In the `diamonds` data set

1. Rename the `x`, `y`, `z` variables as `length`, `width`, `height`.
2. Put their values in cm instead of mm.
3. Create an object `diamonds_modif` containing these three columns and the `clarity`, `cut`, `carat` and `price`, when for the cut is `Premium` or `Ideal`.
4. For each combination `cut/color`, compute the mean of the three variables `length`, `width`, `height`. Keep this summary in an object `dim_summary`.

Exercise (Solution)

- First we create a `diamonds_modif` as wanted.

```r
diamonds_modif <- diamonds %>%
  rename(length = x, width = y, height = z) %>%
  mutate_at(.vars = c("length", "width", "height"), # Change this column
            .funs = function(x) x / 10) %>% # Applying them the same function
  filter(cut %in% c("Premium", "Ideal")) %>% # Filtering on the cut
  select(-depth, -color, -table) # Exclude non wanted columns
```

Then, we use `summarise()` and a `group_by`!

```r
dim_summary <- diamonds_modif %>%
  group_by(cut, clarity) %>%
  summarise_at(c("length", "width", "height"), mean)
```

```
# A tibble: 16 x 5
# Groups:   cut [2]
   cut     clarity length width height
   <ord>   <ord>    <dbl> <dbl>  <dbl>
 1 Premium I1       0.684 0.679  0.413
 2 Premium SI2      0.657 0.654  0.400
 3 Premium SI1      0.605 0.601  0.369
 4 Premium VS2      0.583 0.581  0.357
 5 Premium VS1      0.574 0.571  0.351
 6 Premium VVS2     0.539 0.537  0.330
 7 Premium VVS1     0.504 0.502  0.308
```

```
 8 Premium IF       0.523 0.522  0.319
 9 Ideal   I1       0.675 0.674  0.416
10 Ideal   SI2      0.626 0.627  0.387
11 Ideal   SI1      0.578 0.579  0.357
12 Ideal   VS2      0.543 0.543  0.335
13 Ideal   VS1      0.545 0.547  0.337
14 Ideal   VVS2     0.521 0.523  0.322
15 Ideal   VVS1     0.496 0.498  0.306
16 Ideal   IF       0.483 0.485  0.298
```

Manipulating multiple tables

Multiple tables

- Consider two different sources of data
- Don't hesitate to run the code to see what they look like!

```
# If required
# install.packages("nycflights13")
# Data on New York city flights and airports
library(nycflights13) # New York city flights of 2013 data set
```

```
Error in library(nycflights13): aucun package nommé 'nycflights13' n'est trouvé
```

```
table_1 <- airports %>%
  select(name, lon, lat) %>% # Select those 3 columns
  slice(1:2) # 2 first rows
```

```
Error in select(., name, lon, lat): objet 'airports' introuvable
```

```
table_2 <- airports %>%
  select(name, lon, lat) %>% # Same columns as table_1, different rows
  slice(6:7)
```

```
Error in select(., name, lon, lat): objet 'airports' introuvable
```

```
table_3 <- airports %>%
  select(alt, tz) %>%
  slice(1:2) # Same rows as table_1, different columns
```

```
Error in select(., alt, tz): objet 'airports' introuvable
```

Binding tables `bind_rows()` and `bind_cols()`

```
table_1  %>% # Binding table_2 below table_1
  bind_rows(table_2) # Column names must match!
```

```
Error in list2(...): objet 'table_1' introuvable
```

```
table_1 %>% # Binding table_3 next to table_1
  bind_cols(table_3) # Number of rows must match!
```

```
Error in list2(...): objet 'table_1' introuvable
```

Joining tables

Left jointure `left_join()`

- Suppose we have the two following tables

```
drivers <- tibble(name = c("Sue", "Sue", "Marc", "Gunter", "Rayan", "Rayan"),
                  car = c("Clio", "ZX", "AX", "Lada", "Twingo", "Clio"))
vehicles <- tibble(car = c("Twingo", "Ferrari", "Clio", "Lada", "ZX"),
                   speed = c("140", "280", "160", "85", "160"))
drivers
```

```
# A tibble: 6 x 2
  name   car
  <chr>  <chr>
1 Sue    Clio
2 Sue    ZX
3 Marc   AX
4 Gunter Lada
5 Rayan  Twingo
6 Rayan  Clio
```

```
vehicles
```

```
# A tibble: 5 x 2
  car     speed
  <chr>   <chr>
1 Twingo  140
2 Ferrari 280
3 Clio    160
4 Lada    85
5 ZX      160
```

- We would like to joint the tables to obtain the speed for each driver

```
left_join(drivers, vehicles, by = "car")
```

```
# A tibble: 6 x 3
  name   car    speed
  <chr>  <chr>  <chr>
1 Sue    Clio   160
2 Sue    ZX     160
3 Marc   AX     <NA>
4 Gunter Lada   85
5 Rayan  Twingo 140
6 Rayan  Clio   160
```

- As no record was given for the speed of an AX, an `NA` is inserted.
- The Ferrari car is omitted as it was not present in the first table
- **Watch out**:`left_join()` is not symmetric

```
left_join(vehicles, drivers, by = "car")
```

```
# A tibble: 6 x 3
  car     speed name
  <chr>   <chr> <chr>
1 Twingo  140   Rayan
2 Ferrari 280   <NA>
3 Clio    160   Sue
```

```
4 Clio    160   Rayan
5 Lada    85    Gunter
6 ZX      160   Sue
```

- `left_join(x, y)` is equivalent to `right_join(y, x)`

Inner jointures `inner_join()`

- `inner_join()` only keeps lines where info is present on both tables

```
inner_jointure <- inner_join(drivers, vehicles, by = "car")
```

- Both the `Marc` and `Ferrari` lines are missing.

Full jointures `full_join()`

- `full_join()` keeps all lines, putting `NA` if necessary

```
full_jointure <- full_join(drivers, vehicles, by = "car")
```

- Both the `Marc`and `Ferrari` lines are present, with `NA`.

Exercise

For the `diamonds_modif` object, compute, for each combination `cut/clarity`, the number of observations per group. Include it in the table `dim_summary` created in the previous exercise (previous section).

Exercise (Solution)

```
diamonds_modif %>%
  group_by(cut, clarity) %>%
  summarise(Nb_obs = n()) %>%
  left_join(dim_summary, by = c("cut", "clarity"))
```

```
# A tibble: 16 x 6
# Groups:   cut [2]
   cut     clarity Nb_obs length width height
   <ord>   <ord>    <int>  <dbl> <dbl>  <dbl>
 1 Premium I1         205  0.684 0.679  0.413
 2 Premium SI2       2949  0.657 0.654  0.400
 3 Premium SI1       3575  0.605 0.601  0.369
 4 Premium VS2       3357  0.583 0.581  0.357
 5 Premium VS1       1989  0.574 0.571  0.351
 6 Premium VVS2       870  0.539 0.537  0.330
 7 Premium VVS1       616  0.504 0.502  0.308
 8 Premium IF         230  0.523 0.522  0.319
 9 Ideal   I1         146  0.675 0.674  0.416
10 Ideal   SI2       2598  0.626 0.627  0.387
11 Ideal   SI1       4282  0.578 0.579  0.357
12 Ideal   VS2       5071  0.543 0.543  0.335
13 Ideal   VS1       3589  0.545 0.547  0.337
14 Ideal   VVS2      2606  0.521 0.523  0.322
15 Ideal   VVS1      2047  0.496 0.498  0.306
16 Ideal   IF        1212  0.483 0.485  0.298
```

Cleaning and transforming data

When you have a dataset, there are three interrelated rules which make your dataset tidy:

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

This mean that you must have a clear idea of what is a variable, an observation, or even a value!

A toy example: grades dataset

```
grades <- tibble(
  Name = c("Tommy", "Mary", "Gary", "Cathy"),
  Sex_age = c("m_15", "f_15", "m_16", "f_14"),
  Test1 = c(10, 15, 16, 14),
  Test2 = c(11, 13, 10, 12),
  Test3 = c(12, 13, 17, 10)
  )
```

Name

Sex_age

Test1

Test2

Test3

Tommy

m_15

10

11

12

Mary

f_15

15

13

13

Gary

m_16

16

10

17

Cathy

f_14

14

12

10

One can spot multiple potential problems:

- The `Sex_age` column gathers two variables
- There are three variables tests, which can be what we want, but we might want a variable `Grade` whose value is a grade (and might depend on which test was performed!)

Splitting a column into 2: `separate()`

- The `Sex_age` column should be two columns (two variables)

```
grades_s_a <- # grades with two columns sex age
  grades %>%
  separate(Sex_age, into = c("Sex", "Age"), sep = "_")
print(grades_s_a)
```

```
# A tibble: 4 x 6
  Name  Sex   Age   Test1 Test2 Test3
  <chr> <chr> <chr> <dbl> <dbl> <dbl>
1 Tommy m     15       10    11    12
2 Mary  f     15       15    13    13
3 Gary  m     16       16    10    17
4 Cathy f     14       14    12    10
```

- The `sep` argument can handle regular expressions.
- Not trivial but really powerful tool for string manipulation.

Merging two columns: `unite()`

- The inverse of separate is `unite()`

```
grades_s_a  %>%
  unite(col = "Sex_age", Sex, Age, sep = "_") %>%
  knitr::kable(format = "html")
```

Name

Sex_age

Test1

Test2

Test3

Tommy

m_15

10

11

12

Mary

f_15

15

13

13

Gary

m_16

16

10

17

Cathy

f_14

14

12

10

Gathering columns: `pivot_longer()`

- Suppose that the observation of interest is the grade
- In `grade_s_a`, the a same variable (the grade) is coded in three columns.
- We might **gather** them in one column, and keep test number in one column.
- The table will be a in long format that might be better suited for our use.
- We use the function `pivot_longer`.

```
grades_long <- grades_s_a %>%
  pivot_longer(starts_with("Test"), # Gathered columns
               names_to = "Test", # Column gathering old column names
               values_to = "Grade") # Column gathering values
grades_long
```

```
# A tibble: 12 x 5
   Name  Sex   Age   Test  Grade
   <chr> <chr> <chr> <chr> <dbl>
 1 Tommy m     15    Test1    10
 2 Tommy m     15    Test2    11
 3 Tommy m     15    Test3    12
 4 Mary  f     15    Test1    15
 5 Mary  f     15    Test2    13
 6 Mary  f     15    Test3    13
 7 Gary  m     16    Test1    16
 8 Gary  m     16    Test2    10
 9 Gary  m     16    Test3    17
10 Cathy f     14    Test1    14
11 Cathy f     14    Test2    12
12 Cathy f     14    Test3    10
```

- This format is often better for visualisation with `ggplot`.
- The dimension of `grades_gathered` is now 12×5.
- Alternative code not enumerating all columns to gather, but excluding non gathered ones

```
grades_long <- grades_s_a %>%
  pivot_longer(cols = -c("Name", "Age", "Sex"), # non gathered columns
               names_to = "Test", # Column gathering old column names
```

```
               values_to = "Grade") # Column gathering values
grades_long
```

```
# A tibble: 12 x 5
   Name  Sex   Age   Test  Grade
   <chr> <chr> <chr> <chr> <dbl>
 1 Tommy m     15    Test1    10
 2 Tommy m     15    Test2    11
 3 Tommy m     15    Test3    12
 4 Mary  f     15    Test1    15
 5 Mary  f     15    Test2    13
 6 Mary  f     15    Test3    13
 7 Gary  m     16    Test1    16
 8 Gary  m     16    Test2    10
 9 Gary  m     16    Test3    17
10 Cathy f     14    Test1    14
11 Cathy f     14    Test2    12
12 Cathy f     14    Test3    10
```

Why longer format?

- The longer format is (often) more convenient for summarizing.
- If we want the mean grade per test, for each sex

```
grades_long %>% # Initial data
  group_by(Sex, Test) %>% # Grouping of interest
  summarise(Mean = mean(Grade)) # Get the summary of interest, per group
```

```
# A tibble: 6 x 3
# Groups:   Sex [2]
  Sex   Test   Mean
  <chr> <chr> <dbl>
1 f     Test1  14.5
2 f     Test2  12.5
3 f     Test3  11.5
4 m     Test1  13
5 m     Test2  10.5
6 m     Test3  14.5
```

Get a wider table `pivot_wider()`

- The inverse of `pivot_longer` is `pivot_wider()`
- It spreads a column into multiples, according to a key column

```
# The sep argument is used to give correct names to column
grades_long %>%
  pivot_wider(names_from = "Test", values_from = "Grade")
```

```
# A tibble: 4 x 6
  Name  Sex   Age   Test1 Test2 Test3
  <chr> <chr> <chr> <dbl> <dbl> <dbl>
1 Tommy m     15       10    11    12
2 Mary  f     15       15    13    13
3 Gary  m     16       16    10    17
4 Cathy f     14       14    12    10
```

Input missing data: `complete()`

- Sometimes, when a value is missing, it is not recorded

```
notes <- tibble(Nom = c("Alain", "Alain", "Benoit", "Claire"),
                Discipline = c("Maths", "Francais", "Maths", "Francais"),
                Note = c(16, 9, 17, 11),
                Present = rep("oui", 4))
```

Nom

Discipline

Note

Present

Alain

Maths

16

oui

Alain

Francais

9

oui

Benoit

Maths

17

oui

Claire

Francais

11

oui

- Benoit has no note in French , Claire has no note in maths.
- You might want to **complete** the data such that each Name/Discipline has a note.

```
notes %>%
  complete(Nom, Discipline)
```

```
# A tibble: 6 x 4
  Nom    Discipline  Note Present
  <chr>  <chr>      <dbl> <chr>
1 Alain  Francais       9 oui
2 Alain  Maths         16 oui
3 Benoit Francais      NA <NA>
4 Benoit Maths         17 oui
5 Claire Francais      11 oui
6 Claire Maths         NA <NA>
```

- The default assignment is `NA`.

- You can assign any value to missing data

```
notes_completes <- notes %>%
  complete(Nom, Discipline,
           fill = list(Note = 0, Present = "non")) # Custom the value per column
```

Nom

Discipline

Note

Present

Alain

Francais

9

oui

Alain

Maths

16

oui

Benoit

Francais

0

non

Benoit

Maths

17

oui

Claire

Francais

11

oui

Claire

Maths

0

non

Handling missing data: `replace_na`

Consider the following table which as `NA`

```r
donnees_na <- tibble(Groupe = rep(c("A", "B"), each = 3),
       Nom = c("Al", "Bob", NA, "Dave", "Elle", "Fanch"),
       Note = c(NA, 8, 7, 4.5, 1, 4))
donnees_na
```

```
# A tibble: 6 x 3
  Groupe Nom    Note
  <chr>  <chr> <dbl>
1 A      Al       NA
2 A      Bob       8
3 A      <NA>      7
4 B      Dave      4.5
5 B      Elle      1
6 B      Fanch     4
```

- Suppose we want to assign values to `NA`.
- Use `replace_na`.

```r
donnees_na %>%
  replace_na(replace = list(Nom ="Unknown",
                            Note = 0)) # Assign same value for all columns
```

```
# A tibble: 6 x 3
  Groupe Nom       Note
  <chr>  <chr>    <dbl>
1 A      Al         0
2 A      Bob        8
3 A      Unknown    7
4 B      Dave       4.5
5 B      Elle       1
6 B      Fanch      4
```

- This syntax might not be well suited if many columns are present
- By combining with `mutate_if`, one can do several things

Replace NA based on a condition

```r
# Replace NA only in numeric column
donnees_na %>%
  mutate_if(is.numeric, # We change only if it's numeric
            # The treatment is made column wise
            function(colonne) replace_na(colonne, replace = 0))
```

```
# A tibble: 6 x 3
  Groupe Nom    Note
  <chr>  <chr> <dbl>
1 A      Al       0
2 A      Bob      8
3 A      <NA>     7
4 B      Dave     4.5
5 B      Elle     1
6 B      Fanch    4
```

Replace NA by mean of the column

- It is standard to replace by the mean of the column

```r
# Replace NA by the mean of the numeric column
donnees_na %>%
  mutate_if(is.numeric, # We change only if it's numeric
            # The treatment is made column wise
            function(colonne) replace_na(colonne,
                                         replace = mean(colonne, na.rm = TRUE)))
```

```
# A tibble: 6 x 3
  Groupe Nom    Note
  <chr>  <chr> <dbl>
1 A      Al      4.9
2 A      Bob     8
3 A      <NA>    7
4 B      Dave    4.5
5 B      Elle    1
6 B      Fanch   4
```

- Of course, by combining with `group_by`, you can replace the `NA` by the mean of the group (here, the group A)

```r
# Replace NA by the group mean
donnees_na %>%
  group_by(Groupe) %>% # We first group by the required group
  # And then apply the same treatment
  mutate_if(is.numeric, # We change only if it's numeric
            # The treatment is made column wise
            function(colonne) replace_na(colonne,
                                         replace = mean(colonne,
                                                        na.rm = TRUE))) %>%
  ungroup() # We ungroup for further treatment
```

```
# A tibble: 6 x 3
  Groupe Nom    Note
  <chr>  <chr> <dbl>
1 A      Al      7.5
2 A      Bob     8
3 A      <NA>    7
4 B      Dave    4.5
5 B      Elle    1
6 B      Fanch   4
```

Removing missing data `na.omit`

A straightforward way of getting rid of `NA` values is simply to exclude them from the data set. However, note that this imply to remove the all corresponding row!

```r
na.omit(donnees_na) # Two row are removed
```

```
# A tibble: 4 x 3
  Groupe Nom    Note
  <chr>  <chr> <dbl>
1 A      Bob     8
2 B      Dave    4.5
3 B      Elle    1
4 B      Fanch   4
```

Exercise

From the `diamonds_modif` table of the previous exercise, create the `diamonds_gathered` table having the following the following column structure:

```
# A tibble: 5 x 5
  price cut     clarity Dimension Value
  <int> <ord>   <ord>   <chr>     <dbl>
1 18034 Premium SI2     length    0
2  3959 Premium SI2     height    0.395
3  1155 Ideal   VVS2    length    0.494
4  2722 Ideal   VVS2    width     0.538
5 10398 Premium SI1     width     0.746
```

Exercise (solution)

```
diamonds_gathered <- diamonds_modif %>%
  select(price, cut, clarity, length, width, height) %>%
  gather(key = "Dimension", value = "Value", length, width, height)
```

Dealing with characters

`stringr` package

The `stringr` provides powerful functions to handle characters.

For instance, let's consider the following data.

For a full cover of all these possibilities, there are are lot of dedicated pages, for instance, this one.

```
example <- tibble(Nom = c("Mr_Al_Bob", "Mr_Bob_Col"),
                  Age = c(42, 41))
```

Replacing character pattern

- Say we want to replace `Mr` by a `M.`
- Use the `str_replace` function

```
example %>%
  mutate(Nom = str_replace(Nom, pattern = "Mr", replacement = "M."))
```

```
# A tibble: 2 x 2
  Nom          Age
  <chr>      <dbl>
1 M._Al_Bob     42
2 M._Bob_Col    41
```

- If we want to replace **all** the `_` by a space

```
example %>%
  mutate(Nom = str_replace_all(Nom, pattern = c("_"), replacement = c(" ")))
```

```
# A tibble: 2 x 2
  Nom          Age
  <chr>      <dbl>
1 Mr Al Bob     42
2 Mr Bob Col    41
```

- We can of course remove the `Mr` in front, with `str_remove`

```
example %>%
  mutate(Nom = str_replace_all(Nom, pattern = c("_"), replacement = c(" ")),
         Nom = str_remove(Nom, pattern = "Mr "))
```

```
# A tibble: 2 x 2
  Nom       Age
  <chr>   <dbl>
1 Al Bob     42
2 Bob Col    41
```

Dealing with factors

Transforming numeric in factor

Let's consider the following table

```
athletes <- tibble(Name = c("Alice", "Bob", "Charles", "Dan", "Elsa", "Fanch"),
                   Performance = c(15, 10, 5, 18, 11, 4),
                   Aptitude = c("Strong", "Intermediate", "Weak",
                                "Strong", "Intermediate", "Weak"),
                   Qualified = c(1, 1, 0, 1, 1, 0))
athletes
```

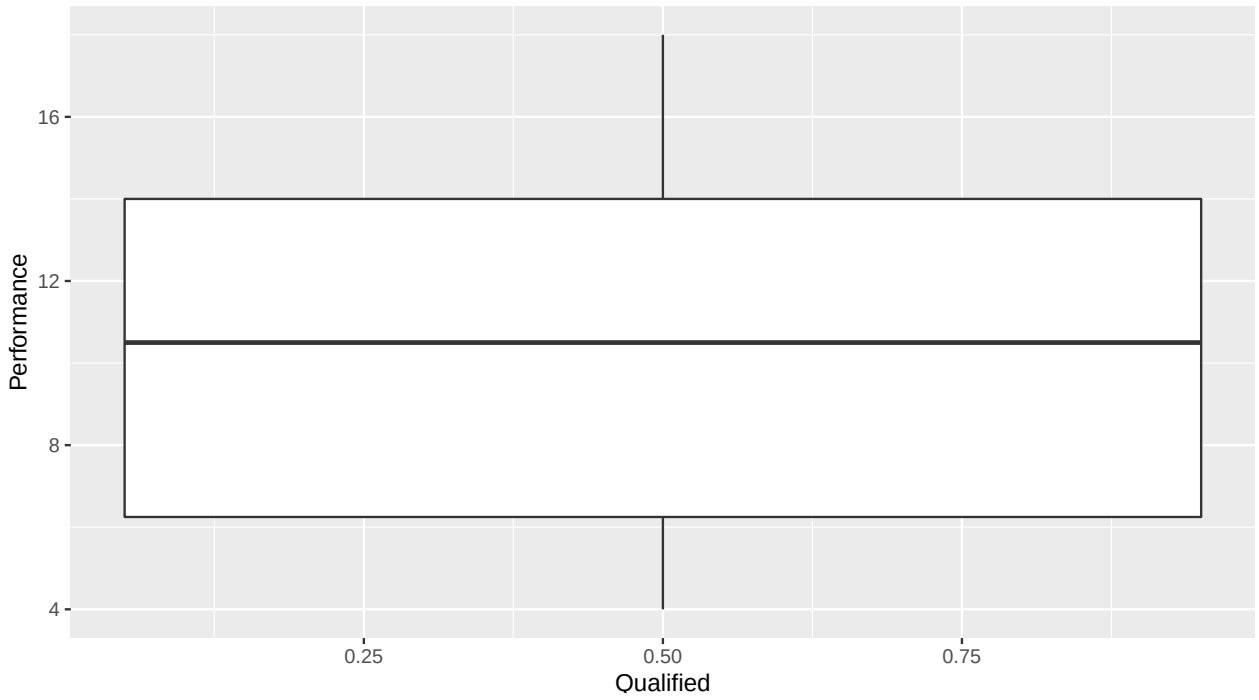
```
# A tibble: 6 x 4
  Name    Performance Aptitude     Qualified
  <chr>         <dbl> <chr>            <dbl>
1 Alice            15 Strong               1
2 Bob              10 Intermediate         1
3 Charles           5 Weak                 0
4 Dan              18 Strong               1
5 Elsa             11 Intermediate         1
6 Fanch             4 Weak                 0
```

The column `Qualified` is here considered a a numeric, whereas it's a qualitative, 1 meaning "yes", 0 meaning "no".

This can pose problem either for graphical representation or, worse, for model fitting in `R`.

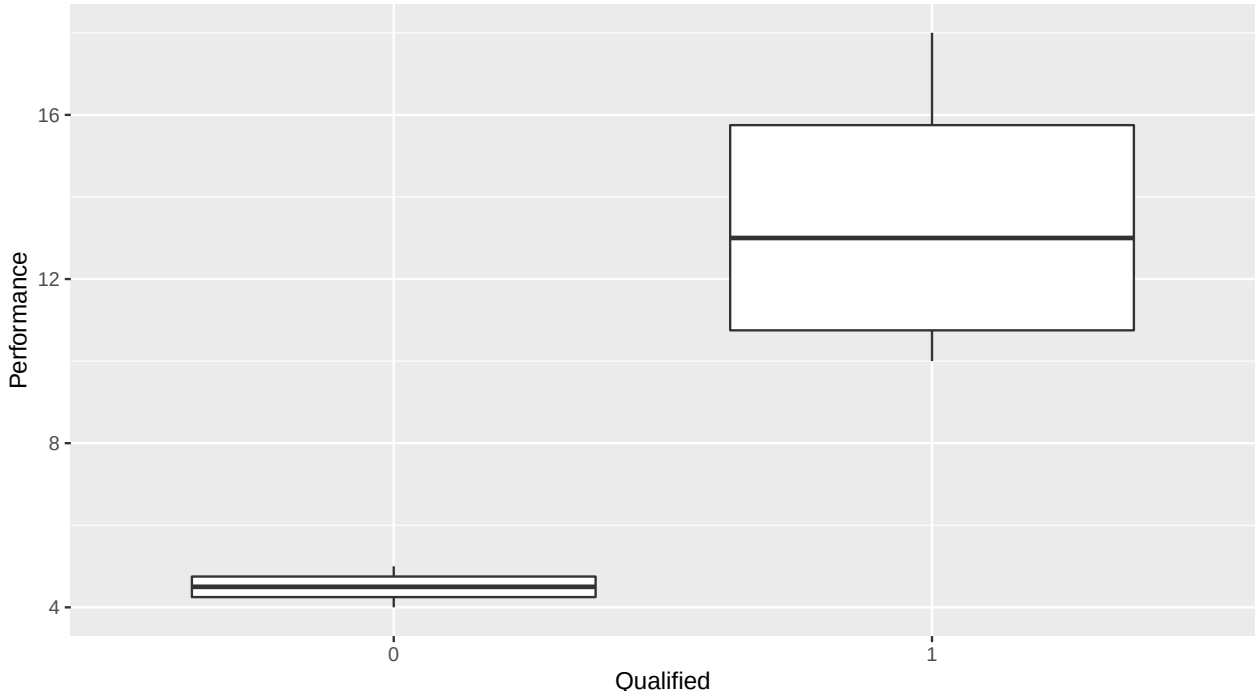
```
# We want a boxplot, with one box per level of Qualified
ggplot(athletes) +
  aes(x = Qualified, y = Performance) +
  geom_boxplot() # Do not work, as Qualified is coded as numeric!
```

```
Warning: Continuous x aesthetic -- did you forget aes(group=...)?
```



To change it, we will change (using `mutate`) the column to factor.

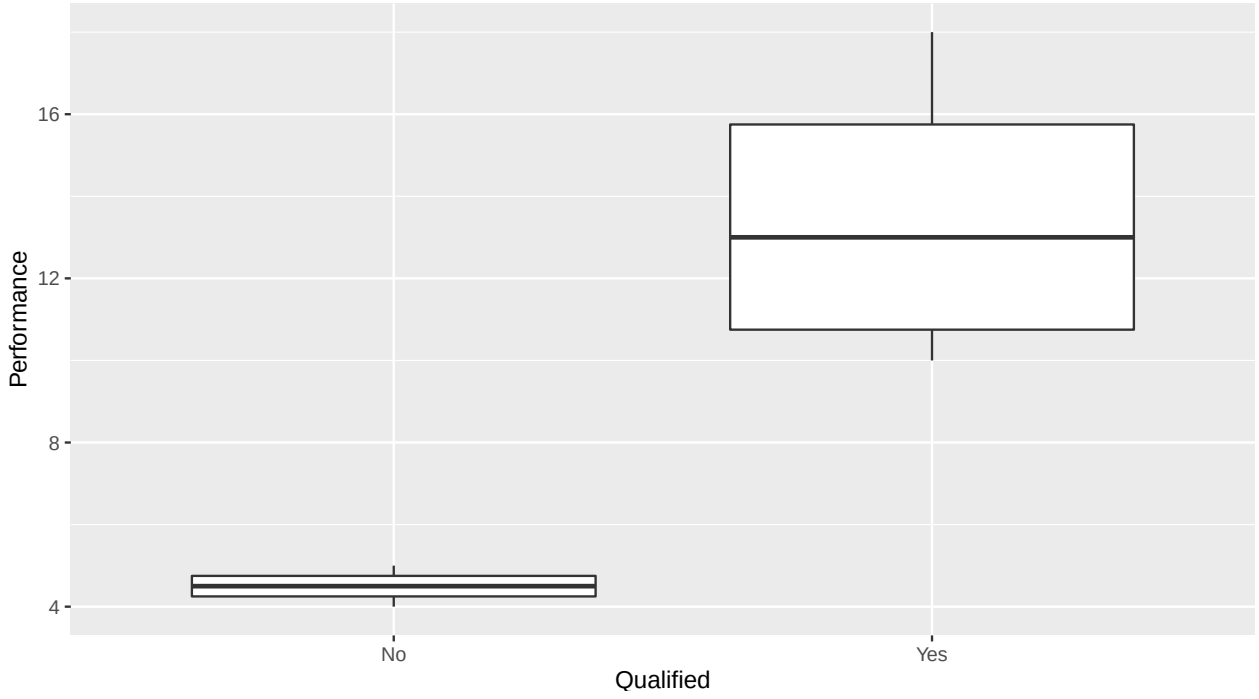
```
athletes <- athletes %>%
  mutate(Qualified = factor(Qualified)) # Changed to qualified
ggplot(athletes) +
  aes(x = Qualified, y = Performance) +
  geom_boxplot() # Now it works
```



Changing factor labelling

The factor *Qualified* now has two **levels**, which are 0 and 1. In the graph above, it would be way better if 0/1 was understandable by a user. We can change it using the `labels` argument to **relabel the levels**. **Watch out, the labels must be specified in the same order than the levels**

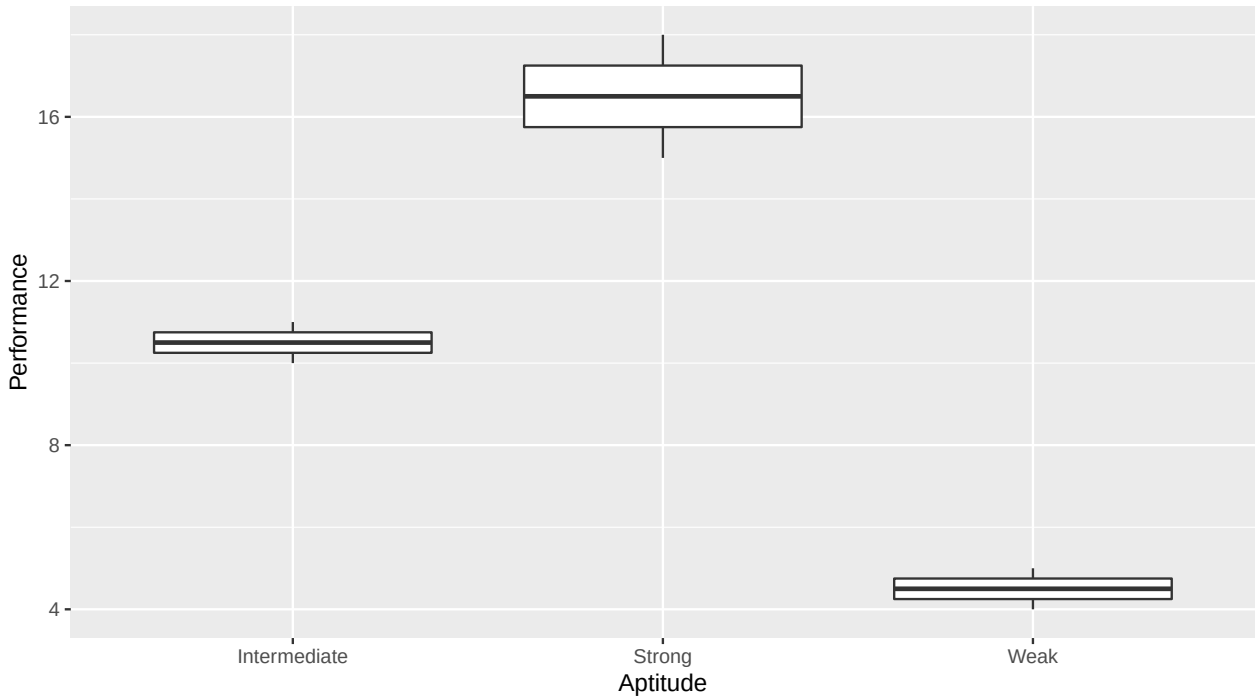
```
athletes %>%
  mutate(Qualified = factor(Qualified,
                            levels = c(0, 1), # Specify the level order
                            labels = c("No", "Yes"))) %>% # And rename in the same order
  ggplot() +
  aes(x = Qualified, y = Performance) +
  geom_boxplot() # Now it works
```



Change characters in factor

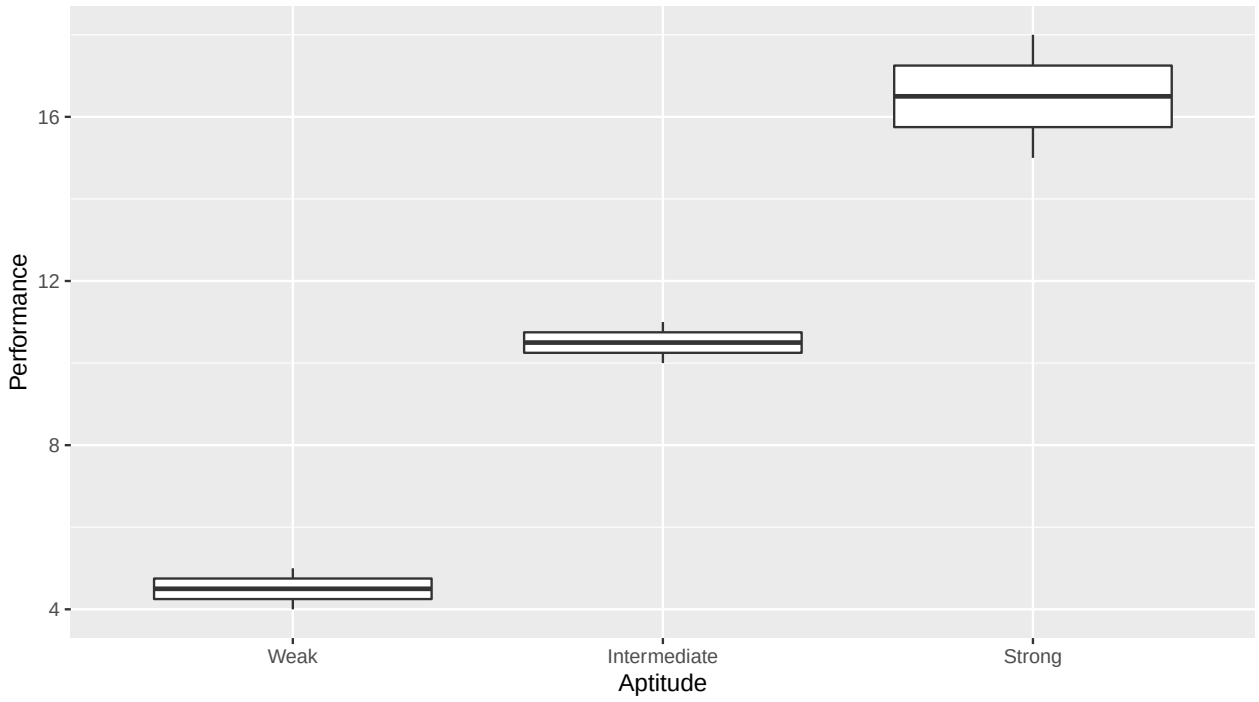
For plots or model, there is no absolute need to transform characters in factor, as `R` understands that it is qualitative attribute.

```
# A boxplot of performance per Aptitude
ggplot(athletes) +
  aes(x = Aptitude, y = Performance) +
  geom_boxplot()
```



Here, the order of the levels is given by alphabetical order, and is not natural. We then can transform Aptitude to factor to specify the right levels order.

```
athletes %>%
  mutate(Aptitude = factor(Aptitude,
                           # Now specify the levels order
                           # Caution, the levels given must appear in the data!
                           # if you want to change labels, use labels = ...
                           levels = c("Weak", "Intermediate", "Strong"))) %>%
# A boxplot of performance per Aptitude
  ggplot() +
  aes(x = Aptitude, y = Performance) +
  geom_boxplot() # The levels are now in natural order
```


Performance
16
12
8
4
Weak
Intermediate
Strong
Aptitude