

# Nettoyage et manipulation des données avec R

## Introduction au tidyverse

Amal Tawfik

2026-01-15

### Plan

- Introduction au `tidyverse`
- Structure des données et principes *tidy*
- Nettoyer le tableau de données (`janitor`)
- Le *pipe* : enchaîner les opérations
- Manipulation des données avec `dplyr` : `filter()`, `select()`, `arrange()`, `mutate()`, `summarise()`, `group_by()`
- Résumés globaux vs par groupe
- Recoder des variables (`ifelse()`, `case_when()` et interfaces graphiques)

### Objectifs

- Transformer un fichier brut en un `data.frame` (ou `tibble`) propre et prêt pour l'analyse.

Pour y parvenir :

- Identifier et corriger les problèmes courants dans les données brutes
- Utiliser les outils du `tidyverse` pour nettoyer les données
- Structurer vos données selon les principes *tidy*
- Préparer vos données pour l'analyse statistique

### Pourquoi nettoyer les données ?

**80%** du temps est consacré à la préparation et au nettoyage des données, et seulement 20% à l'analyse.

Ce qu'on rencontre dans les données brutes :

- valeurs manquantes (`NA`, cases vides)

- codages incohérents (F, f, 1, 2, etc.)
- erreurs de saisie (99999 comme revenu)
- doublons (individu présent plusieurs fois)
- classes erronées (âge importé comme texte)
- colonnes inutiles ou mal nommées

## Qu'est-ce que le *tidyverse* ?

Le **tidyverse** est un ensemble cohérent de packages R dédiés à la *science des données*, conçu pour être :

- lisible
- expressif
- reproductible
- intuitif

Il repose sur une philosophie commune, une syntaxe uniforme et la structure *tidy* des données.

- Installation en une fois : `install.packages("tidyverse")`
- Charger le tidyverse : `library(tidyverse)`

## Les packages principaux du *tidyverse*

Le **tidyverse** fournit des outils pour chaque étape du *data workflow* :

Étape	Packages & utilités
Importer	<b>readr</b> (CSV), <b>readxl</b> (Excel), <b>haven</b> (SPSS/Stata/SAS)
Manipuler	<b>dplyr</b> : filtrer, trier, sélectionner, créer des variables
Restructurer	<b>tidyr</b> : pivoter (long wide), séparer, combiner
Visualiser	<b>ggplot2</b> : graphiques élégants et cohérents
Gérer les textes	<b>stringr</b> : chaînes de caractères
Gérer les facteurs	<b>forcats</b> : niveaux, réordonnements
Gérer les dates	<b>lubridate</b> : dates, heures, durées
Tibble moderne	<b>tibble</b> : version améliorée d'un <code>data.frame</code>

Tous ces packages partagent une syntaxe cohérente et fonctionnent parfaitement ensemble.

## Structure d'un tableau de données : variables, valeurs et observations

- Une **variable** représente une caractéristique mesurée (quantité, qualité, propriété).
- Une **valeur** est le résultat obtenu pour une variable lors d'une mesure.
- Une **observation** regroupe l'ensemble des valeurs mesurées dans les mêmes conditions (cas ou individu statistique).
- Un **tableau de données rectangulaire** associe chaque valeur à une variable (colonne) et à une observation (ligne).

Dans R, ce type de tableau est représenté par un `data.frame` et sa version modernisée est le `tibble`.

## Données *tidy*

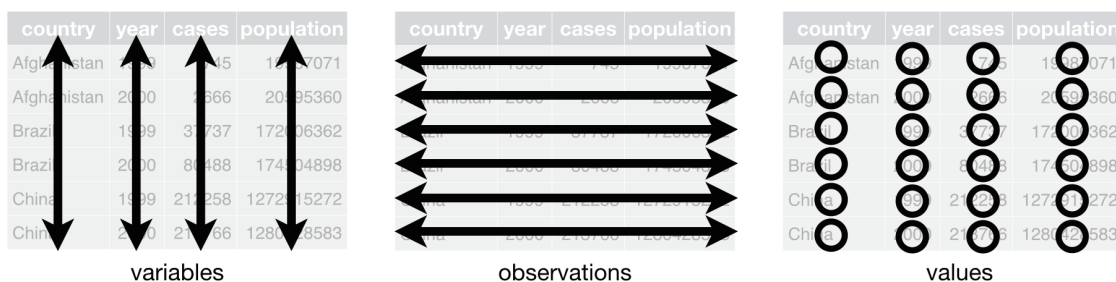
Tous les tableaux de données rectangulaires ne sont pas également faciles à utiliser pour l'analyse.

Plusieurs tableaux peuvent représenter les mêmes données, mais seule une structure *tidy* est pleinement exploitable dans le `tidyverse`.

Les données sont dites *tidy* (rangées) lorsque trois règles interdépendantes sont respectées :

- chaque variable correspond à une colonne
- chaque observation correspond à une ligne
- chaque valeur occupe une cellule et chaque cellule contient une seule valeur (pas de valeurs empilées du type “foot, basket, tennis”, ni de valeurs composites du type “23 kg”)

## Visualisation de la structure *tidy*



Source : <https://r4ds.hadley.nz/data-tidy.html#fig-tidy-structure>

## En pratique, que change la structure *tidy* ?

Lorsque ces règles sont respectées :

- chaque unité d’observation est clairement identifiée
- des unités d’observation différentes sont stockées dans des tables distinctes

Conséquence : on évite de mélanger individus, événements et mesures dans une même table.

Les données *tidy* respectent une structure standardisée, conçue pour faciliter la manipulation, la visualisation et l’analyse avec les outils du [tidyverse](#).

## Les *tibbles*

Dans le [tidyverse](#), les données sont manipulées sous forme de [tibble](#), une version moderne et améliorée de [data.frame](#) de R.

Les [tibbles](#) sont :

- fournis par le package [tibble](#) (au cœur du [tidyverse](#))
- acceptés par la plupart des fonctions du [tidyverse](#)
- retournés par défaut par [dplyr](#), [tidyr](#), [ggplot2](#), etc.

Un [tibble](#) est un [data.frame](#) plus strict, plus lisible et plus sûr.

## Pourquoi utiliser des *tibbles* ?

Contrairement aux [data.frames](#) classiques, les [tibbles](#) :

- n’ont pas de noms de lignes ([rownames](#))
- autorisent des noms de colonnes non standards (espaces, caractères spéciaux, nombres...) entourés avec des backticks (`)
- s’affichent intelligemment (aperçu des lignes, dimensions, types de variables)
- pas de *partial matching* sur les noms de colonnes (si une table `d` contient une colonne `qualif`, `d$qual` ne retournera pas cette colonne)
- avertissent lorsqu’une colonne n’existe pas

Ils font moins de choses automatiques mais signalent plus tôt les erreurs

## Compatibilité de tibble et data.frame

Un `tibble` reste totalement compatible avec un `data.frame` :

- la plupart des fonctions du tidyverse acceptent un `data.frame` en entrée
- elles retournent généralement un `tibble`

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.6
v forcats    1.0.1      v stringr    1.6.0
v ggplot2    4.0.1      v tibble     3.3.0
v lubridate  1.9.4      v tidyr      1.3.2
v purrr      1.2.0
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
head(mtcars) # Affiche les 6 premières lignes (n = 6 par défaut)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

## Conversion de data.frame en tibble

`as_tibble()` convertit un `data.frame` en `tibble` :

- les variables et leurs types sont conservés
- les `rownames` sont supprimés par défaut

```
as_tibble(mtcars)
```

```
# A tibble: 32 x 11
  mpg   cyl  disp    hp  drat    wt   qsec    vs  am  gear  carb
<dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6  160   110  3.9   2.62  16.5     0    1     4     4
2  21       6  160   110  3.9   2.88  17.0     0    1     4     4
3 22.8      4  108    93  3.85  2.32  18.6     1    1     4     1
4 21.4      6  258   110  3.08  3.22  19.4     1    0     3     1
5 18.7      8  360   175  3.15  3.44  17.0     0    0     3     2
6 18.1      6  225   105  2.76  3.46  20.2     1    0     3     1
7 14.3      8  360   245  3.21  3.57  15.8     0    0     3     4
8 24.4      4  147.    62  3.69  3.19  20       1    0     4     2
9 22.8      4  141.    95  3.92  3.15  22.9     1    0     4     2
10 19.2      6  168.   123  3.92  3.44  18.3     1    0     4     4
# i 22 more rows
```

## Convertir les noms de lignes en variables

Il est possible de transformer les `rownames` en colonne explicite en utilisant l'argument `rownames` :

```
as_tibble(mtcars, rownames = "car_names")
```

```
# A tibble: 32 x 12
  car_names   mpg   cyl  disp    hp  drat    wt   qsec    vs  am  gear  carb
<chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 Mazda RX4    21     6  160   110  3.9   2.62  16.5     0    1     4     4
2 Mazda RX4 ~  21     6  160   110  3.9   2.88  17.0     0    1     4     4
3 Datsun 710   22.8    4  108    93  3.85  2.32  18.6     1    1     4     1
4 Hornet 4 D~  21.4    6  258   110  3.08  3.22  19.4     1    0     3     1
5 Hornet Spo~  18.7    8  360   175  3.15  3.44  17.0     0    0     3     2
6 Valiant     18.1    6  225   105  2.76  3.46  20.2     1    0     3     1
7 Duster 360   14.3    8  360   245  3.21  3.57  15.8     0    0     3     4
8 Merc 240D    24.4    4  147.    62  3.69  3.19  20       1    0     4     2
9 Merc 230     22.8    4  141.    95  3.92  3.15  22.9     1    0     4     2
10 Merc 280    19.2    6  168.   123  3.92  3.44  18.3     1    0     4     4
# i 22 more rows
```

## Nettoyer les noms des variables

Le package `janitor` fournit des outils simples pour nettoyer et standardiser les données.

Souvent, les données importées contiennent des colonnes avec :

- des espaces
- des caractères spéciaux
- des majuscules mélangées
- des noms peu lisibles

La fonction `clean_names()` nettoie les noms de colonnes pour qu'ils soient propres, cohérents et faciles à manipuler.

### Que fait `clean_names()` ?

Par défaut, la fonction `clean_names()` du package `janitor` applique la convention de nommage *snake case* qui est la convention recommandée dans le `tidyverse`.

Elle transforme les noms de colonnes pour :

- être en minuscules
- remplacer les espaces et caractères spéciaux par des *underscores* (`_`)
- éviter les noms invalides ou ambigus
- produire des noms faciles à taper et cohérents

### Exemple

```
library(janitor)
```

Attachement du package : 'janitor'

Les objets suivants sont masqués depuis 'package:stats':

```
chisq.test, fisher.test
```

```
d <- tibble(
  "Nom Col 1" = 1:3,
  "Âge (ans)" = c(24, 30, 29),
  "Score (%)" = c(80, 90, 85)
)
d
```

```
# A tibble: 3 x 3
  `Nom Col 1` `Âge (ans)` `Score (%)`
    <int>      <dbl>      <dbl>
1       1       24       80
2       2       30       90
3       3       29       85
```

```
clean_names(d)
```

```
# A tibble: 3 x 3
  nom_col_1 age_ans score_percent
    <int>      <dbl>      <dbl>
1       1       24       80
2       2       30       90
3       3       29       85
```

## Autres fonctions utiles de nettoyage

Plusieurs autres fonctions du package `janitor` :

- `remove_empty()` supprime les lignes ou colonnes entièrement composées de NA.
- `get_dupes()` identifie les doublons sur une ou plusieurs variables.

Exemple

```
d <- tibble(
  id    = c(1, 2, 2, NA),
  nom   = c("Alice", "Bob", "Bob", NA),
  score = c(15, 10, 15, NA),
  vide  = c(NA, NA, NA, NA))
d
```

```
# A tibble: 4 x 4
  id nom    score vide
  <dbl> <chr> <dbl> <lgl>
1     1 Alice    15 NA
2     2 Bob     10 NA
3     2 Bob     15 NA
4    NA <NA>    NA NA
```



## Examples

```
remove_empty(d)
```

value for "which" not specified, defaulting to c("rows", "cols")

```
# A tibble: 3 x 3
  id nom    score
<dbl> <chr> <dbl>
1     1 Alice    15
2     2 Bob     10
3     2 Bob     15
```

```
remove_empty(d, which = "cols")
```

```
# A tibble: 4 x 3
  id nom    score
<dbl> <chr> <dbl>
1     1 Alice    15
2     2 Bob     10
3     2 Bob     15
4    NA <NA>     NA
```

```
get_dupes(d, score)
```

```
# A tibble: 2 x 5
  score dupe_count id nom  vide
<dbl>    <int> <dbl> <chr> <lgl>
1    15         2     1 Alice NA
2    15         2     2 Bob  NA
```

```
get_dupes(d, id, nom)
```

```
# A tibble: 2 x 5
  id nom  dupe_count score vide
<dbl> <chr>    <int> <dbl> <lgl>
1     2 Bob         2     10 NA
2     2 Bob         2     15 NA
```

## Le *pipe* : enchaîner les opérations

Le *pipe* (`|>` natif R ou `%>%` du package `magrittr`) permet d'enchaîner plusieurs opérations dans l'ordre où on les lit.

**Principe :** Ce qui se trouve à **gauche** du pipe est envoyé comme premier argument à la fonction à **droite**.

```
data |> fonction1() |> fonction2() |> fonction3()
```

```
data |>
  fonction1() |>
  fonction2() |>
  fonction3()
```

“Prends cet objet → puis applique cette transformation → puis cette autre → etc.”

## Pourquoi utiliser le *pipe* ?

Exemple

```
d <- tibble(
  id_etud = c("E001", "E002", "E003", "E004"),
  genre = factor(c("F", "H", "F", "H")),
  age = c(17, 20, 23, 19),
  filiere = factor(c("Soins infirmiers",
                    "Physiothérapie",
                    "Soins infirmiers",
                    "Ergothérapie")),
  abs_just = c(2, 4, 1, 3),
  abs_non_just = c(0, 1, 2, 0),
  revenu = c(6200, NA, NA, 4800)
)
```

# A tibble: 4 x 7

	id_etud	genre	age	filiere	abs_just	abs_non_just	revenu
	<chr>	<fct>	<dbl>	<fct>	<dbl>	<dbl>	<dbl>
1	E001	F	17	Soins infirmiers	2	0	6200
2	E002	H	20	Physiothérapie	4	1	NA
3	E003	F	23	Soins infirmiers	1	2	NA
4	E004	H	19	Ergothérapie	3	0	4800

## Sans *pipe*, deux problèmes courants

1. Fonctions imbriquées (difficile à lire, ordre inversé) :

```
d1 <- select(  
  mutate(  
    filter(d, age >= 18),  
    abs_total = abs_just + abs_non_just  
  ),  
  id_etud, genre, filiere, abs_total  
)  
d1
```

```
# A tibble: 3 x 4  
  id_etud genre filiere      abs_total  
  <chr>   <fct> <fct>         <dbl>  
1 E002    H     Physiothérapie      5  
2 E003    F     Soins infirmiers     3  
3 E004    H     Ergothérapie        3
```

2. Objets intermédiaires (répétitif, risque d'erreur) :

```
d1 <- filter(d, age >= 18)  
d1 <- mutate(d1, abs_total = abs_just + abs_non_just)  
d1 <- select(d1, id_etud, genre, filiere, abs_total)  
d1
```

```
# A tibble: 3 x 4  
  id_etud genre filiere      abs_total  
  <chr>   <fct> <fct>         <dbl>  
1 E002    H     Physiothérapie      5  
2 E003    F     Soins infirmiers     3  
3 E004    H     Ergothérapie        3
```

## Le *pipe* : lire le code comme une phrase

Avec *pipe*, lisible de gauche à droite (et de haut en bas) :

```
d1 <- d |>
  filter(age >= 18) |>
  mutate(abs_total = abs_just + abs_non_just) |>
  select(id_etud, genre, filiere, abs_total)
d1
```

```
# A tibble: 3 x 4
  id_etud genre filiere      abs_total
  <chr>   <fct> <fct>         <dbl>
1 E002    H    Physiothérapie      5
2 E003    F    Soins infirmiers      3
3 E004    H    Ergothérapie         3
```

On lit le code comme une phrase

“Crée d1 en prenant d, en filtrant les personnes de 18 ans ou plus, en ajoutant une variable `abs_total` qui additionne les absences justifiées et non justifiées, puis en sélectionnant uniquement les variables `id_etud`, `genre`, `filiere` et `abs_total`.”

## Les avantages du *pipe*

Enchaîner les opérations avec `|>` (appelé *pipeline*) permet :

- Lisibilité : le code se lit comme une phrase, dans l’ordre naturel
- Modularité : facile d’ajouter ou retirer une étape
- Débogage : on peut tester étape par étape
- Reproductibilité : chaque transformation est explicite

### À retenir :

- On sauvegarde le résultat final dans un objet : `resultat <- donnees |> ...`
- Le *pipe* n’est pas qu’un outil syntaxique, c’est une façon de penser la transformation des données

## Manipuler les données avec dplyr

`dplyr` est le package du `tidyverse` dédié à la manipulation de données. Il propose une syntaxe claire et cohérente, sous forme de **verbes**, pour transformer une ou plusieurs tables.

Les principaux verbes de manipulation :

Action sur les données	Verbe de <code>dplyr</code>
Sélectionner des lignes (observations)	<code>filter()</code>
Sélectionner ou réordonner des colonnes (variables)	<code>select()</code>
Trier les observations	<code>arrange()</code>
Créer ou transformer des variables	<code>mutate()</code>
Résumer les données (par groupe)	<code>summarise()</code> + <code>group_by()</code>

### Sélectionner des lignes (observations) — `filter()`

La fonction `filter()` permet de conserver uniquement les lignes qui satisfont une ou plusieurs conditions.

- Elle agit sur les observations (lignes), pas sur les variables (colonnes).

#### Principe

```
data |> filter(condition)
```

- la condition doit être `TRUE` pour que la ligne soit conservée
- plusieurs conditions peuvent être combinées (`&`, `|`)

Exemple : conserver les personnes âgées de 18 ans ou plus

```
d |> filter(age >= 18)
```

### Écrire des conditions logiques

Pour sélectionner des observations (créer des sous-ensembles) ou recoder des variables, on écrit des **conditions logiques**.

Une condition est une expression logique dont le résultat est :

- `TRUE` = la ligne est conservée
- `FALSE` = la ligne est exclue

Les conditions reposent sur :

- des opérateurs de comparaison
- des opérateurs logiques

`filter()` conserve uniquement les lignes pour lesquelles la condition est vraie.

## Opérateurs logiques et de comparaison

### *Opérateurs de comparaison*

Opérateur	Signification
<code>==</code>	égal à
<code>!=</code>	différent de
<code>&gt;</code>	strictement supérieur à
<code>&lt;</code>	strictement inférieur à
<code>&gt;=</code>	supérieur ou égal à
<code>&lt;=</code>	inférieur ou égal à

### *Opérateurs logiques*

Opérateur	Signification
<code>&amp;</code>	ET logique
<code> </code>	OU logique
<code>!</code>	négation logique

## Exemples de conditions logiques avec `filter()`

```
d |> filter(age < 18) # comparaison
```

```
# A tibble: 1 x 7
  id_etud genre  age filiere      abs_just abs_non_just revenu
<chr>   <fct> <dbl> <fct>         <dbl>         <dbl>   <dbl>
1 E001    F      17 Soins infirmiers      2             0   6200
```

```
d |> filter(genre == "F") # égalité
```

```
# A tibble: 2 x 7
  id_etud genre   age filiere      abs_just abs_non_just revenu
  <chr>   <fct> <dbl> <fct>          <dbl>         <dbl>   <dbl>
1 E001    F      17 Soins infirmiers      2             0    6200
2 E003    F      23 Soins infirmiers      1             2     NA
```

```
d |> filter(abs_just >= 4 & genre == "H") # ET
```

```
# A tibble: 1 x 7
  id_etud genre   age filiere      abs_just abs_non_just revenu
  <chr>   <fct> <dbl> <fct>          <dbl>         <dbl>   <dbl>
1 E002    H      20 Physiothérapie      4             1     NA
```

```
d |> filter(filiere == "Physiothérapie" | filiere == "Ergothérapie") # OU
```

```
# A tibble: 2 x 7
  id_etud genre   age filiere      abs_just abs_non_just revenu
  <chr>   <fct> <dbl> <fct>          <dbl>         <dbl>   <dbl>
1 E002    H      20 Physiothérapie      4             1     NA
2 E004    H      19 Ergothérapie        3             0    4800
```

```
d |> filter(!is.na(revenu)) # valeurs non manquantes
```

```
# A tibble: 2 x 7
  id_etud genre   age filiere      abs_just abs_non_just revenu
  <chr>   <fct> <dbl> <fct>          <dbl>         <dbl>   <dbl>
1 E001    F      17 Soins infirmiers      2             0    6200
2 E004    H      19 Ergothérapie        3             0    4800
```

## Sélectionner des colonnes (variables) — select()

La fonction `select()` permet de choisir, réordonner ou supprimer des variables (colonnes).

— Elle agit sur les variables (colonnes), pas sur les observations (lignes).

### Principe

```
data |> select(col1, col2, ...)
```

— on indique les variables à conserver

— l'ordre d'écriture = ordre final des variables

Exemple : conserver uniquement les variables `id_etud` et `genre`.

```
d |> select(id_etud, genre)
```

## Sélectionner une plage de variables — `select()`

Sélectionner un ensemble de variables consécutives en utilisant `:` :

```
d |> select(genre:filiere)
```

```
# A tibble: 4 x 3
  genre    age filiere
  <fct> <dbl> <fct>
1 F      17 Soins infirmiers
2 H      20 Physiothérapie
3 F      23 Soins infirmiers
4 H      19 Ergothérapie
```

`genre:filiere` sélectionne toutes les variables de `genre` à `filiere` incluses (selon leur ordre dans le tableau).

Quand l'utiliser ?

- Lorsque les variables se suivent dans le tableau
- Pour éviter d'écrire une longue liste de noms de variables

La sélection par plage dépend de l'ordre des variables dans le tableau.

## Réordonner les variables (colonnes) — `select()`

`select()` permet aussi de changer l'ordre des variables dans le tableau de données.

Exemple : placer `revenu` en première colonne, puis conserver toutes les autres avec `everything()`.

```
d |> select(revenu, everything())
```



```
# A tibble: 4 x 7
  revenu id_etud genre   age filiere      abs_just abs_non_just
  <dbl> <chr>   <fct> <dbl> <fct>      <dbl>      <dbl>
1   6200 E001     F    17 Soins infirmiers      2          0
2     NA E002     H    20 Physiothérapie      4          1
3     NA E003     F    23 Soins infirmiers      1          2
4   4800 E004     H    19 Ergothérapie       3          0
```

Utilités :

- Mettre les variables clés au début du tableau de données
- Faciliter la lecture et la compréhension des données avant une analyse ou une exportation

## Sélecteurs de colonnes (*helpers*)

`select()` propose des **sélecteurs** (*helpers*) pour choisir plusieurs variables sans écrire leurs noms un par un. Ces fonctions sont fournies par le package `tidyselect` dont la liste est [ici](#).

Exemples de sélecteurs : `starts_with()`, `ends_with()`, `contains()`

```
d |> select(starts_with("abs")) # commence par "abs"
```

```
# A tibble: 4 x 2
  abs_just abs_non_just
  <dbl>      <dbl>
1      2          0
2      4          1
3      1          2
4      3          0
```

```
d |> select(ends_with("_just")) # se termine par "_just"
```

```
# A tibble: 4 x 2
  abs_just abs_non_just
  <dbl>      <dbl>
1      2          0
2      4          1
3      1          2
4      3          0
```

```
d |> select(contains("age"))      # contient "age"
```

```
# A tibble: 4 x 1
```

```
  age
<dbl>
1   17
2   20
3   23
4   19
```

## Supprimer ou renommer des variables

Supprimer une variable avec - :

```
d |> select(-abs_non_just)      # supprimer "abs_non_just"
```

```
# A tibble: 4 x 6
```

```
  id_etud genre  age filiere      abs_just revenu
  <chr>   <fct> <dbl> <fct>      <dbl>  <dbl>
1 E001    F      17 Soins infirmiers      2    6200
2 E002    H      20 Physiothérapie      4      NA
3 E003    F      23 Soins infirmiers      1      NA
4 E004    H      19 Ergothérapie       3    4800
```

Renommer des variables :

```
d |>
  select(identifiant = id_etud,
         sexe = genre,
         revenu)
```

```
# A tibble: 4 x 3
```

```
  identifiant sexe  revenu
  <chr>        <fct> <dbl>
1 E001        F      6200
2 E002        H      NA
3 E003        F      NA
4 E004        H    4800
```

## Trier les observations — `arrange()`

La fonction `arrange()` permet de trier les lignes (observations) d'un tableau selon une ou plusieurs variables.

Elle change uniquement l'ordre des observations, sans modifier les valeurs.

Principe

```
data |> arrange(variable)
```

- le tri est croissant par défaut
- les valeurs manquantes (NA) sont placées à la fin

Exemple : trier les observations par âge croissant

```
d |> arrange(age)
```

## Trier selon plusieurs variables — `arrange()`

Il est possible de trier selon plusieurs variables, dans l'ordre indiqué.

Exemple : trier d'abord par `filiere`, puis par `age` à l'intérieur de chaque `filiere`.

```
d |> arrange(filiere, age)
```

```
# A tibble: 4 x 7
  id_etud genre  age filiere      abs_just abs_non_just revenu
<chr>   <fct> <dbl> <fct>         <dbl>         <dbl>   <dbl>
1 E004    H      19 Ergothérapie      3             0    4800
2 E002    H      20 Physiothérapie    4             1     NA
3 E001    F      17 Soins infirmiers  2             0    6200
4 E003    F      23 Soins infirmiers  1             2     NA
```

Pour trier en ordre décroissant, on utilise `desc()` :

```
d |> arrange(desc(revenu))
```

```
# A tibble: 4 x 7
  id_etud genre  age filiere      abs_just abs_non_just revenu
<chr>   <fct> <dbl> <fct>         <dbl>         <dbl>   <dbl>
1 E001    F      17 Soins infirmiers  2             0    6200
2 E004    H      19 Ergothérapie    3             0    4800
3 E002    H      20 Physiothérapie    4             1     NA
4 E003    F      23 Soins infirmiers  1             2     NA
```

## Créer ou transformer des variables — mutate()

La fonction `mutate()` permet de :

- créer de nouvelles variables
- transformer/recoder des variables existantes

Elle ne modifie pas le nombre d'observations.

Principe

```
data |> mutate(nouvelle_variable = expression)
```

- la nouvelle variable est ajoutée au tableau

Exemple : créer une variable `abs_total` (total des absences)

```
d |> mutate(abs_total = abs_just + abs_non_just)
```

## Exemples avec mutate()

```
d <- d |>
mutate(
  age2 = age^2,                # transformation numérique (au carré)
  majeur = age >= 18,          # indicateur logique TRUE / FALSE
  age_plus_1 = age + 1,        # création d'une nouvelle variable
  abs_total = abs_just + abs_non_just, # somme de deux variables
  abs_dicho = as.integer(abs_total > 3), # indicateur binaire (1 si > 3 absences, 0 sinon)
  id_etud = as.factor(id_etud) # transformation en facteur
)
```

```
d
```

```
# A tibble: 4 x 12
```

	id_etud	genre	age	filier	abs_just	abs_non_just	revenu	age2	majeur	age_plus_1	
	<fct>	<fct>	<dbl>	<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<lgl>	<dbl>	
1	E001	F	17	Soins infirmiers	2	0	6200	289	FALSE	18	
2	E002	H	20	Physiothérapie	4	1	NA	400	TRUE	21	
3	E003	F	23	Soins infirmiers	1	2	NA	529	TRUE	24	
4	E004	H	19	Ergothérapie	3	0	4800	361	TRUE	20	

Dans `mutate()`, les variables sont créées de haut en bas : une variable nouvellement créée peut être réutilisée immédiatement.

## Résumer les données — `summarise()`

La fonction `summarise()` permet de calculer des statistiques sur une ou plusieurs variables d'un tableau de données.

- Elle retourne un nouveau tableau contenant uniquement les résumés spécifiés.

Principe

```
data |>
  summarise(
    nom_stat1 = expression1,
    nom_stat2 = expression2
  )
```

- Chaque `expression` calcule une statistique
- Le résultat est un *tibble* réduit contenant les nouvelles variables

## Exemple d'utilisation de `summarise()`

`n()` retourne le nombre d'observations (lignes)

Exemple : un *tibble* avec 3 variables résumé.

```
d |>
  summarise(
    n = n(), # nombre total d'observations
    age_moyen = mean(age, na.rm = TRUE), # moyenne (ignore les NA)
    revenu_max = max(revenu, na.rm = TRUE) # maximum (ignore les NA)
  )
```

```
# A tibble: 1 x 3
      n age_moyen revenu_max
<int>   <dbl>     <dbl>
1     4    19.8      6200
```

`summarise()` agrège les données : elle produit un ou des résumés statistiques.

## Résumer les données par groupe

`group_by()` définit des groupes d'observations à partir d'une ou plusieurs variables et `summarise()` calcule des statistiques pour chaque groupe.

Principe

```
data |>
  group_by(variable_groupe) |>      # crée des groupes
  summarise(nom_stat = expression) # calcule une statistique par groupe
```

Exemple : un *tibble* avec 1 ligne par filière contenant les résumés.

```
d |>
  group_by(filiere) |>              # crée un groupe par filière
  summarise(
    n = n(),                        # nombre d'observations par filière
    age_moyen = mean(age, na.rm = TRUE) # moyenne d'âge par filière (ignore les NA)
  )
```

```
# A tibble: 3 x 3
  filiere      n age_moyen
  <fct>      <int>   <dbl>
1 Ergothérapie    1     19
2 Physiothérapie 1     20
3 Soins infirmiers 2     20
```

### `summarise()` : global vs par groupe

- Sans `group_by()`, `summarise()` calcule une statistique globale : le résultat contient une seule ligne et une variable par statistique demandée.
- Avec `group_by()`, `summarise()` calcule une statistique par groupe : une ligne est produite pour chaque groupe et une variable par statistique demandée.
- Les fonctions comme `mean()`, `sum()`, `min()`, `max()`, `sd()`, `median()`, `n()` sont fréquemment utilisées dans `summarise()`.

### Recoder une variable — `ifelse()`

La fonction `ifelse()` permet de créer ou recoder une variable en appliquant une condition simple.

Elle fonctionne comme un “si ... alors ... sinon ...”.

Principe

```
ifelse(condition, valeur_si_vrai, valeur_si_faux)
```

Exemples : créer une variable `majeur` (oui/non) et dichotomiser le revenu

```
d |>
  mutate(
    majeur = ifelse(age >= 18, "oui", "non"),
    rev_haut = ifelse(revenu > 5000, 1, 0)
  )
```

# A tibble: 4 x 13

	id_etud	genre	age	filiere	abs_just	abs_non_just	revenu	age2	majeur	age_plus_1
	<fct>	<fct>	<dbl>	<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<dbl>
1	E001	F	17	Soins infirmiers	2	0	6200	289	non	18
2	E002	H	20	Physiothérapie	4	1	NA	400	oui	21
3	E003	F	23	Soins infirmiers	1	2	NA	529	oui	24
4	E004	H	19	Ergothérapie	3	0	4800	361	oui	20

## Recoder avec `ifelse()` — bonnes pratiques

- `ifelse()` est adapté aux recodages binaires (2 catégories).
- La condition de `ifelse()` peut utiliser plusieurs variables (ex. `ifelse(age >= 18 & genre == "F", ...)`).
- Il est possible d’emboîter plusieurs `ifelse()`, mais cela devient vite illisible → utiliser `case_when()` pour plusieurs conditions.

## Recodages multiples — `case_when()`

La fonction `case_when()` est une version lisible de plusieurs `ifelse` en cascade et permet de recoder ou de créer une variable selon plusieurs conditions.

Principe

```
case_when(
  condition1 ~ valeur1,
  condition2 ~ valeur2,
  .default   = valeur_par_defaut # valeur utilisée si aucune condition n'est vraie
)
```

- Dans `case_when()`, les conditions sont testées une par une, de haut en bas.
- La **première condition vraie** détermine la valeur renvoyée.
- L'argument `.default` = permet d'indiquer une valeur à utiliser si aucune condition n'est satisfaite (sinon `.default` = NULL).
- Sans `.default`, toutes les valeurs qui ne correspondent à aucune condition deviennent NA.
- L'ordre des conditions est essentiel : toujours aller du plus **spécifique** → au plus **général**.

## Recoder une variable — `case_when()`

Recoder les âges en classes

```
d |>
  mutate(age_cat = case_when(
    age < 18 ~ "<18",
    age >= 18 & age < 30 ~ "18-29",
    age >= 30 ~ "30+",
    .default = NA # NA renvoyé si aucune condition n'est vraie
  ))
```

# Variante : gestion explicite des valeurs manquantes avec `is.na()`

```
d |>
  mutate(age_cat = case_when(
    is.na(age) ~ NA, # attribue NA si l'âge est manquant
    age < 18 ~ "<18",
    age >= 18 & age < 30 ~ "18-29",
    .default = "30+" # toutes les autres situations (30 ans et +)
  ))
```

# A tibble: 4 x 13

	id_etud	genre	age	filiere	abs_just	abs_non_just	revenu	age2	majeur	age_plus_1
	<fct>	<fct>	<dbl>	<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<lgl>	<dbl>
1	E001	F	17	Soins infirmiers	2	0	6200	289	FALSE	18



2	E002	H	20	Physiothérapie	4	1	NA	400	TRUE	21
29										
3	E003	F	23	Soins infirmiers	1	2	NA	529	TRUE	24
29										
4	E004	H	19	Ergothérapie	3	0	4800	361	TRUE	20
29										

- `is.na(x)` teste si une valeur est manquante.
- Elle renvoie TRUE si x est NA, et FALSE sinon.
- Très utile pour recoder explicitement les valeurs manquantes dans `case_when()`.

## Recoder avec plusieurs variables — `case_when()`

Recoder selon l'âge et le genre

```
d |>
mutate(age_genre = case_when(
  age >= 18 & genre == "H" ~ "Homme majeur",
  age < 18 & genre == "H" ~ "Homme mineur",
  age >= 18 & genre == "F" ~ "Femme majeure",
  age < 18 & genre == "F" ~ "Femme mineure",
  .default = NA # NA renvoyé si aucune condition n'est vraie
))
```

```
# A tibble: 4 x 13
  id_etud genre   age filiere      abs_just abs_non_just revenu age2 majeur age_plus_1
  <fct>   <fct> <dbl> <fct>         <dbl>         <dbl>   <dbl> <dbl> <lgl>      <dbl>
1 E001    F     17 Soins infirmiers      2             0   6200   289 FALSE      18
2 E002    H     20 Physiothérapie      4             1    NA     400 TRUE       21
3 E003    F     23 Soins infirmiers      1             2    NA     529 TRUE       24
4 E004    H     19 Ergothérapie        3             0  4800     361 TRUE       20
```

- `case_when()` permet de combiner plusieurs variables dans les conditions (ex. âge et genre).

## Interfaces graphiques de recodage

Le package `questionr` propose des [interfaces graphiques](#) (*Addins*) dans RStudio qui permettent de recoder les variables sans écrire de code.

- Chaque *Addin* génère automatiquement le code R correspondant que vous pouvez ensuite exécuter dans votre script.

Dans RStudio : **Addins** → **QUESTIONR** →

- **Levels recoding** : recoder et regrouper des modalités d'une variable catégorielle, changer la classe d'une variable
- **Levels ordering** : réordonner les niveaux d'un facteur
- **Numeric range dividing** : découper une variable numérique en classes

## Les *Addins* interactifs du package *questionr*

<i>Addin</i> (RStudio)	Fonction principale	Fonctions R équivalentes
<b>Levels recoding</b>	Regrouper / renommer les modalités d'un facteur ou d'un caractère	<code>fct_recode()</code> , <code>case_when()</code>
<b>Levels ordering</b>	Réordonner manuellement les niveaux d'un facteur	<code>fct_relevel()</code> , <code>factor(..., levels = ...)</code>
<b>Numeric range dividing</b>	Découper une variable numérique en classes (catégories)	<code>cut()</code> , <code>case_when()</code>

- Chaque *Addin* génère automatiquement le code R reproduisant exactement la transformation choisie.
- Vous devez ensuite exécuter ce code pour appliquer le recodage.

## Ressources

- [Site du tidyverse](#)
- [Introduction à R et au tidyverse](#)