

guide-R

Guide pour l'analyse de données d'enquêtes avec R

Joseph Larmarange

29 septembre 2022

Table des matières

Préface	6
Remerciements	8
Licence	8
I Bases du langage	9
1 Packages	10
1.1 Installation (CRAN)	11
1.2 Chargement	11
1.3 Mise à jour	12
1.4 Installation depuis GitHub	13
1.5 Le tidyverse	14
2 Vecteurs	17
2.1 Types et classes	17
2.2 Création d'un vecteur	18
2.3 Longueur d'un vecteur	21
2.4 Combiner des vecteurs	22
2.5 Vecteurs nommés	22
2.6 Indexation par position	24
2.7 Indexation par nom	25
2.8 Indexation par condition	26
2.9 Assignment par indexation	30
2.10 En résumé	31
2.11 webin-R	32
3 Listes	33
3.1 Propriétés et création	33
3.2 Indexation	36
3.3 En résumé	40
3.4 webin-R	41

4	Tableaux de données	42
4.1	Propriétés et création	42
4.2	Indexation	44
4.3	Afficher les données	48
4.4	En résumé	55
4.5	webin-R	56
5	Tibbles	57
5.1	Le concept de tidy data	57
5.2	tibbles : des tableaux de données améliorés . . .	57
5.3	Données et tableaux imbriqués	62
6	Attributs	65
II	Manipulation de données	68
7	Le pipe	69
7.1	Le pipe natif de R : <code> ></code>	70
7.2	Le pipe du tidyverse : <code>%>%</code>	71
7.3	Vaut-il mieux utiliser <code> ></code> ou <code>%>%</code> ?	72
7.4	Accéder à un élément avec <code>purrr::pluck()</code> et <code>purrr::chuck()</code>	72
8	dplyr	75
8.1	Opérations sur les lignes	76
8.1.1	<code>filter()</code>	76
8.1.2	<code>slice()</code>	81
8.1.3	<code>arrange()</code>	82
8.1.4	<code>slice_sample()</code>	84
8.1.5	<code>distinct()</code>	86
8.2	Opérations sur les colonnes	88
8.2.1	<code>select()</code>	88
8.2.2	<code>relocate()</code>	93
8.2.3	<code>rename()</code>	93
8.2.4	<code>rename_with()</code>	95
8.2.5	<code>pull()</code>	95
8.2.6	<code>mutate()</code>	96
8.3	Opérations groupées	97
8.3.1	<code>group_by()</code>	97
8.3.2	<code>summarise()</code>	101
8.3.3	<code>count()</code>	103

8.3.4	Grouper selon plusieurs variables	104
8.4	Cheatsheet	109
8.5	webin-R	109
9	Facteurs et forcats	110
9.1	Création d'un facteur	110
9.2	Changer l'ordre des modalités	113
9.3	Modifier les modalités	118
9.4	Découper une variable numérique en classes . . .	125
10	Combiner plusieurs variables	129
10.1	if_else()	129
10.2	case_when()	132
10.3	recode_if()	134
11	Étiquettes de variables	139
11.1	Principe	139
11.2	Manipulation sur un vecteur / une colonne . . .	141
11.3	Manipulation sur un tableau de données	143
11.4	Préserver les étiquettes	144
12	Étiquettes de valeurs	146
12.1	La classe <code>haven_labelled</code>	146
12.2	Manipulation sur un vecteur / une colonne . . .	147
12.3	Manipulation sur un tableau de données	151
12.4	Conversion	152
12.4.1	Quand convertir les vecteurs labellisés ? .	152
12.4.2	Convertir un vecteur labellisé en facteur .	154
12.4.3	Convertir un vecteur labellisé en numérique ou en texte	156
12.4.4	Conversion conditionnelle en facteurs . . .	158
13	Valeurs manquantes	162
13.1	Valeurs manquantes étiquetées (<i>tagged NAs</i>) . .	163
13.1.1	Création et test	163
13.1.2	Valeurs uniques, doublons et tris	166
13.1.3	Tagged NAs et étiquettes de valeurs . . .	167
13.1.4	Conversion en user NAs	169
13.2	Valeurs manquantes définies par l'utilisateur (<i>user NAs</i>)	170
13.2.1	Création	171
13.2.2	Tests	173

13.2.3	Conversion	174
14	Import & Export de données	177
14.1	Importer un fichier texte	177
14.1.1	Structure d'un fichier texte	177
14.1.2	Interface graphique avec RStudio	178
14.1.3	Dans un script	179
14.2	Importer un fichier Excel	180
14.3	Importer depuis des logiciels de statistique	181
14.3.1	SPSS	181
14.3.2	SAS	182
14.3.3	Stata	183
14.3.4	dBase	183
14.4	Sauver ses données	183
14.5	Export de tableaux de données	185
III	Analyses	186
15	Graphiques avec ggplot2	187
15.1	Ressources	187
15.2	Les bases de ggplot2	187
15.3	Cheatsheet	192
15.4	Exploration visuelle avec esquisse	192
15.5	webin-R	194
15.6	Combiner plusieurs graphiques	195
IV	Manipulation avancée	199
16	Dates avec lubridate	200
17	Réorganisation avec tidyr	201

Préface

Site en construction

Le présent site est en cours de construction et sera complété dans les prochains mois.

En attendant, nous vous conseillons de consulter le site [analyse-R](https://larmarange.github.io/analyse-R/).

Ce guide porte sur l'analyse de données d'enquêtes avec le logiciel **R**, un logiciel libre de statistiques et de traitement de données. Les exemples présentés ici relèvent principalement du champs des sciences sociales quantitatives et des sciences de santé. Ils peuvent néanmoins s'appliquer à d'autres champs disciplinaires. Cependant, comme tout ouvrage, ce guide ne peut être exhaustif.

Ce guide présente comment réaliser des analyses statistiques et diverses opérations courantes (comme la manipulation de données ou la production de graphiques) avec **R**. Il ne s'agit pas d'un cours de statistiques : les différents chapitres présupposent donc que vous avez déjà une connaissance des différentes techniques présentées. Si vous souhaitez des précisions théoriques / méthodologiques à propos d'un certain type d'analyses, nous vous conseillons d'utiliser votre moteur de recherche préféré. En effet, on trouve sur internet de très nombreux supports de cours (sans compter les nombreux ouvrages spécialisés disponibles en librairie).

De même, il ne s'agit pas d'une introduction ou d'un guide pour les utilisatrices et utilisateurs débutant·es. Si vous découvrez **R**, nous vous conseillons la lecture de l'*Introduction à R et au tidyverse* de Julien Barnier (<https://juba.github.io/tidyverse/>). Vous pouvez également lire les chapitres introductifs d'*analyse-R : Introduction à l'analyse d'enquêtes avec R et RStudio* (<https://larmarange.github.io/analyse-R/>).

Néanmoins, quelques rappels sur les bases du langage sont fournis dans la section *Bases du langage*. Une bonne compréhension de ces dernières, bien qu'un peu ardue de prime abord, permet de comprendre le sens des commandes qu'on utilise et de pleinement exploiter la puissance que **R** offre en matière de manipulation de données.

R disposent de nombreuses extensions ou packages (plus de 16 000) et il existe souvent plusieurs manières de procéder pour arriver au même résultat. En particulier, en matière de manipulation de données, on oppose¹ souvent *base R* qui repose sur les fonctions disponibles en standard dans **R**, la majorité étant fournies dans les packages `{base}`, `{utils}` ou encore `{stats}`, qui sont toujours chargés par défaut, et le `{tidyverse}` qui est une collection de packages comprenant, entre autres, `{dplyr}`, `{tibble}`, `{tidyr}`, `{forcats}` ou encore `{ggplot2}`. Il y a un débat ouvert, parfois passionné, sur le fait de privilégier l'une ou l'autre approche, et les avantages et inconvénients de chacune dépendent de nombreux facteurs, comme la lisibilité du code ou bien les performances en temps de calcul. Dans ce guide, nous avons adopté un point de vue pragmatique et utiliserons, le plus souvent mais pas exclusivement, les fonctions du `{tidyverse}`, de même que nous avons privilégié d'autres packages, comme `{gtsummary}` ou `{questionr}` par exemple pour la statistique descriptive. Cela ne signifie pas, pour chaque point abordé, qu'il s'agit de l'unique manière de procéder. Dans certains cas, il s'agit simplement de préférences personnelles.

¹ Une comparaison des deux syntaxes est illustrée par une [vignette dédiée de dplyr](#).

Bien qu'il en reprenne de nombreux contenus, ce guide ne se substitue pas au site [analyse-R](#). Il s'agit plutôt d'une version complémentaire qui a vocation à être plus structurée et parfois plus sélective dans les contenus présentés.

En complément, on pourra également se référer aux [webin-R](#), une série de vidéos avec partage d'écran, librement accessibles sur Youtube : <https://www.youtube.com/c/webinR>.

Cette version du guide a utilisé *R version 4.2.1 (2022-06-23 ucrt)*. Ce document est généré avec [quarto](#) et le code source est disponible sur [GitHub](#). Pour toute suggestion ou correction, vous pouvez ouvrir un [ticket GitHub](#). Pour d'autres questions, vous pouvez utiliser les forums de discussion disponibles en bas

de chaque page sur la version web du guide. Ce document est régulièrement mis à jour. La dernière version est consultable sur <https://larmarange.github.io/guide-R/>.

Remerciements

Ce document a bénéficié de différents apports provenant notamment de l'*[Introduction à R](#)* et de l'*[Introduction à R et au tidyverse](#)* de Julien Barnier et d'*[analyse-R : introduction à l'analyse d'enquêtes avec R et RStudio](#)*.

Merci donc à Julien Barnier, Julien Biaudet, François Briatte, Milan Bouchet-Valat, Ewen Gallic, Frédérique Giraud, Joël Gombin, Mayeul Kauffmann, Christophe Lalanne & Nicolas Robette.

Licence

Ce document est mis à disposition selon les termes de la [Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International](#).



partie I

Bases du langage

1 Packages

L'installation par défaut du logiciel **R** contient le cœur du programme ainsi qu'un ensemble de fonctions de base fournissant un grand nombre d'outils de traitement de données et d'analyse statistiques.

R étant un logiciel libre, il bénéficie d'une forte communauté d'utilisateurs qui peuvent librement contribuer au développement du logiciel en lui ajoutant des fonctionnalités supplémentaires. Ces contributions prennent la forme d'extensions (packages en anglais) pouvant être installées par l'utilisateur et fournissant alors diverses fonctionnalités supplémentaires.

Il existe un très grand nombre d'extensions (plus de 16 000 à ce jour), qui sont diffusées par un réseau baptisé **CRAN** (*Comprehensive R Archive Network*).

La liste de toutes les extensions disponibles sur **CRAN** est disponible ici : <http://cran.r-project.org/web/packages/>.

Pour faciliter un peu le repérage des extensions, il existe un ensemble de regroupements thématiques (économétrie, finance, génétique, données spatiales...) baptisés Task views : <http://cran.r-project.org/web/views/>.

On y trouve notamment une *Task view* dédiée aux sciences sociales, listant de nombreuses extensions potentiellement utiles pour les analyses statistiques dans ce champ disciplinaire : <http://cran.r-project.org/web/views/SocialSciences.html>.

On peut aussi citer le site *Awesome R* (<https://github.com/qinwf/awesome-R>) qui fournit une liste d'extensions choisies et triées par thématique.

1.1 Installation (CRAN)

L'installation d'une extension se fait par la fonction `install.packages()`, à qui on fournit le nom de l'extension. Par exemple, si on souhaite installer l'extension `{gtsummary}` :

```
install.packages("gtsummary")
```

Sous **RStudio**, on pourra également cliquer sur *Install* dans l'onglet *Packages* du quadrant inférieur droit.

Alternativement, on pourra avoir recours au package `{remotes}` et à sa fonction `remotes::install_cran()` :

```
remotes::install_cran("gtsummary")
```

Note

Le package `{remotes}` n'est pas disponible par défaut sous **R** et devra donc être installé classiquement avec `install.packages("remotes")`. À la différence de `install.packages()`, `remotes::install_cran()` vérifie si le package est déjà installé et, si oui, si la version installée est déjà la dernière version, avant de procéder à une installation complète si et seulement si cela est nécessaire.

1.2 Chargement

Une fois un package installé (c'est-à-dire que ses fichiers ont été téléchargés et copiés sur votre ordinateur), ses fonctions et objets ne sont pas directement accessibles. Pour pouvoir les utiliser, il faut, **à chaque session de travail**, charger le package en mémoire avec la fonction `library()` ou la fonction `require()` :

```
library(gtsummary)
```

À partir de là, on peut utiliser les fonctions de l'extension, consulter leur page d'aide en ligne, accéder aux jeux de données qu'elle contient, etc.

Alternativement, pour accéder à un objet ou une fonction d'un package sans avoir à le charger en mémoire, on pourra avoir recours à l'opérateur `::`. Ainsi, l'écriture `p::f()` signifie la fonction `f()` du package `p`. Cette écriture sera notamment utilisée tout au long de ce guide pour indiquer à quel package appartient telle fonction : `remotes::install_cran()` indique que la fonction `install_cran()` provient du packages `{remotes}`.

! Important

Il est important de bien comprendre la différence entre `install.packages()` et `library()`. La première va chercher un package sur internet et l'installe en local sur le disque dur de l'ordinateur. On n'a besoin d'effectuer cette opération qu'une seule fois. La seconde lit les informations de l'extension sur le disque dur et les met à disposition de **R**. On a besoin de l'exécuter à chaque début de session ou de script.

1.3 Mise à jour

Pour mettre à jour l'ensemble des packages installés, il suffit d'exécuter la fonction `update.packages()` :

```
update.packages()
```

Sous **RStudio**, on pourra alternativement cliquer sur *Update* dans l'onglet *Packages* du quadrant inférieur droit.

Si on souhaite désinstaller une extension précédemment installée, on peut utiliser la fonction `remove.packages()` :

```
remove.packages("gtsummary")
```

💡 Installer / Mettre à jour les packages utilisés par un projet

Après une mise à jour majeure de **R**, il est souvent nécessaire de réinstaller tous les packages utilisés. De même, on peut parfois souhaiter mettre à jour uniquement les packages utilisés par un projet donné sans avoir à mettre à jour tous les autres packages présents sur son PC.

Une astuce consiste à avoir recours à la fonction `renv::dependencies()` qui examine le code du projet courant pour identifier les packages utilisés, puis à passer cette liste de packages à `remotes::install_cran()` qui installera les packages manquants ou pour lesquels une mise à jour est disponible.

Il vous suffit d'exécuter la commande ci-dessous :

```
renv::dependencies() |>
  purrr::pluck("Package") |>
  remotes::install_cran()
```

1.4 Installation depuis GitHub

Certains packages ne sont pas disponibles sur **CRAN** mais seulement sur **GitHub**, une plateforme de développement informatique. Il s'agit le plus souvent de packages qui ne sont pas encore suffisamment matures pour être diffusés sur **CRAN** (sachant que des vérifications strictes sont effectués avant qu'un package ne soit référencés sur **CRAN**).

Dans d'autres cas de figure, la dernière version stable d'un package est disponible sur **CRAN** tandis que la version en cours de développement est, elle, disponible sur **GitHub**. Il faut être vigilant avec les versions de développement. Parfois, elle corrige un bug ou introduit une nouvelle fonctionnalité qui n'est pas encore dans la version stable. Mais les versions de développement peuvent aussi contenir de nouveaux bugs ou des fonctionnalités instables.

⚠ Sous Windows

Pour les utilisatrices et utilisateurs sous **Windows**, il faut être conscient que le code source d'un package doit être compilé afin de pouvoir être utilisé. **CRAN** fournit une version des packages déjà compilée pour **Windows** ce qui facilite l'installation.

Par contre, lorsque l'on installe un package depuis **GitHub**, **R** ne récupère que le code source et il est donc nécessaire de compiler localement le package. Pour cela, il est nécessaire que soit installé sur le PC un outil complémentaire appelé **RTools**. Il est téléchargeable à l'adresse <https://cran.r-project.org/bin/windows/Rtools/>.

Le code source du package `{labelled}` est disponible sur **GitHub** à l'adresse <https://github.com/larmarange/labelled>. Pour installer la version de développement de `{labelled}`, on aura recours à la fonction `remotes::install_github()` à laquelle on passera la partie située à droite de <https://github.com/> dans l'URL du package, à savoir :

```
remotes::install_github("larmarange/labelled")
```

1.5 Le tidyverse

Le terme `{tidyverse}` est une contraction de *tidy* (qu'on pourrait traduire par bien rangé) et de *universe*. Il s'agit en fait d'une collection de packages conçus pour travailler ensemble et basés sur une philosophie commune.

Ils abordent un très grand nombre d'opérations courantes dans **R** (la liste n'est pas exhaustive) :

- visualisation (`{ggplot2}`)
- manipulation des tableaux de données (`{dplyr}`, `{tidyr}`)
- import/export de données (`{readr}`, `{readxl}`, `{haven}`)
- manipulation de variables (`{forcats}`, `{stringr}`, `{lubridate}`)

- programmation (`{purrr}`, `{magrittr}`, `{glue}`)

Un des objectifs de ces extensions est de fournir des fonctions avec une syntaxe cohérente, qui fonctionnent bien ensemble, et qui retournent des résultats prévisibles. Elles sont en grande partie issues du travail d'[Hadley Wickham](#), qui travaille désormais pour [RStudio](#).

`{tidyverse}` est également le nom d'une extension générique qui permet d'installer en une seule commande l'ensemble des packages constituant le *tidyverse* :

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

Lorsque l'on charge le package `{tidyverse}` avec `library()`, cela charge également en mémoire les principaux packages du *tidyverse*².

```
library(tidyverse)
```

² Si on a besoin d'un autre package du *tidyverse* comme `{lubridate}`, il faudra donc le charger individuellement.

```
-- Attaching packages ----- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.4
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.2      v forcats 0.5.2
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```



Figure 1.1: Packages chargés avec `library(tidyverse)`

2 Vecteurs

Les vecteurs sont l'objet de base de **R** et correspondent à une liste de valeurs. Leurs propriétés fondamentales sont :

- les vecteurs sont unidimensionnels (i.e. ce sont des objets à une seule dimension, à la différence d'une matrice par exemple) ;
- toutes les valeurs d'un vecteur sont d'un seul et même type ;
- les vecteurs ont une longueur qui correspond au nombre de valeurs contenues dans le vecteur.

2.1 Types et classes

Dans **R**, il existe plusieurs types fondamentaux de vecteurs et, en particulier, :

- les nombres réels (c'est-à-dire les nombres décimaux³), par exemple 5.23 ;
- les nombres entiers, que l'on saisi en ajoutant le suffixe L⁴, par exemple 4L ;
- les chaînes de caractères (qui correspondent à du texte), que l'on saisit avec des guillemets doubles (") ou simples ('), par exemple "abc" ;
- les valeurs logiques ou valeurs booléennes, à savoir vrai ou faux, que l'on représente avec les mots TRUE et FALSE (en majuscules⁵).

En plus de ces types de base, il existe de nombreux autres types de vecteurs utilisés pour représenter toutes sortes de données, comme les facteurs (voir Chapitre 9) ou les dates (voir Chapitre 16).

³ Pour rappel, **R** étant anglophone, le caractère utilisé pour indiquer les chiffres après la virgule est le point (.).

⁴ **R** utilise 32 bits pour représenter des nombres entiers, ce qui correspond en informatique à des entiers longs ou *long integers* en anglais, d'où la lettre L utilisée pour indiquer un nombre entier.

⁵ On peut également utiliser les raccourcis T et F. Cependant, pour une meilleure lisibilité du code, il est préférable d'utiliser les versions longues TRUE et FALSE.

La fonction `class()` renvoie la nature d'un vecteur tandis que la fonction `typeof()` indique la manière dont un vecteur est stocké de manière interne par **R**.

Table 2.1: Le type et la classe des principaux types de vecteurs

x	class(x)	typeof(x)
3L	integer	integer
5.3	numeric	double
TRUE	logical	logical
"abc"	character	character
factor("a")	factor	integer
as.Date("2020-01-01")	Date	double

Astuce

Pour un vecteur numérique, le type est **"double"** car **R** utilise une double précision pour stocker informatiquement les nombres réels.

En interne, les facteurs sont représentés par un nombre entier auquel est attaché une étiquette, c'est pourquoi `typeof()` renvoie **"integer"**.

Quand aux dates, elles sont stockées en interne sous la forme d'un nombre réel représentant le nombre de jours depuis le 1^{er} janvier 1970, d'où le fait que `typeof()` renvoie **"double"**.

2.2 Création d'un vecteur

Pour créer un vecteur, on utilisera la fonction `c()` en lui passant la liste des valeurs à combiner⁶.

```
taille <- c(1.88, 1.65, 1.92, 1.76, NA, 1.72)
taille
```

```
[1] 1.88 1.65 1.92 1.76    NA 1.72
```

⁶ La lettre **c** est un raccourci du mot anglais *combine*, puisque cette fonction permet de combiner des valeurs individuelles dans un vecteur unique.

```
sexe <- c("h", "f", "h", "f", "f", "f")
sexe
```

```
[1] "h" "f" "h" "f" "f" "f"
```

```
urbain <- c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)
urbain
```

```
[1] TRUE TRUE FALSE FALSE FALSE TRUE
```

Nous l'avons vu, toutes les valeurs d'un vecteur doivent obligatoirement être du même type. Dès lors, si on essaie de combiner des valeurs de différents types, **R** essaiera de les convertir au mieux. Par exemple :

```
x <- c(2L, 3.14, "a")
x
```

```
[1] "2"      "3.14"   "a"
```

```
class(x)
```

```
[1] "character"
```

Dans le cas présent, toutes les valeurs ont été converties en chaînes de caractères.

Dans certaines situations, on peut avoir besoin de créer un vecteur d'une certaine longueur mais dont toutes les valeurs sont identiques. Cela se réalise facilement avec `rep()` à qui on indiquera la valeur à répéter puis le nombre de répétitions :

```
rep(2, 10)
```

```
[1] 2 2 2 2 2 2 2 2 2 2
```

On peut aussi lui indiquer plusieurs valeurs qui seront alors répétées en boucle :

```
rep(c("a", "b"), 3)
```

```
[1] "a" "b" "a" "b" "a" "b"
```

Dans d'autres situations, on peut avoir besoin de créer un vecteur contenant une suite de valeurs, ce qui se réalise aisément avec `seq()` à qui on précisera les arguments **from** (point de départ), **to** (point d'arrivée) et **by** (pas). Quelques exemples valent mieux qu'un long discours :

```
seq(1, 10)
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
seq(5, 17, by = 2)
```

```
[1] 5 7 9 11 13 15 17
```

```
seq(10, 0)
```

```
[1] 10 9 8 7 6 5 4 3 2 1 0
```

```
seq(100, 10, by = -10)
```

```
[1] 100 90 80 70 60 50 40 30 20 10
```

```
seq(1.23, 5.67, by = 0.33)
```

```
[1] 1.23 1.56 1.89 2.22 2.55 2.88 3.21 3.54 3.87 4.20 4.53 4.86 5.19 5.52
```

L'opérateur `:` est un raccourci de la fonction `seq()` pour créer une suite de nombres entiers. Il s'utilise ainsi :

```
1:5
```

```
[1] 1 2 3 4 5
```

```
24:32
```

```
[1] 24 25 26 27 28 29 30 31 32
```

```
55:43
```

```
[1] 55 54 53 52 51 50 49 48 47 46 45 44 43
```

2.3 Longueur d'un vecteur

La longueur d'un vecteur correspond au nombre de valeurs qui le composent. Elle s'obtient avec `length()` :

```
length(taille)
```

```
[1] 6
```

```
length(c("a", "b"))
```

```
[1] 2
```

La longueur d'un vecteur vide (`NULL`) est zéro.

```
length(NULL)
```

```
[1] 0
```

2.4 Combiner des vecteurs

Pour combiner des vecteurs, rien de plus simple. Il suffit d'utiliser `c()` ! Les valeurs des différents vecteurs seront mises bout à bout pour créer un unique vecteur.

```
x <- c(2, 1, 3, 4)
length(x)
```

```
[1] 4
```

```
y <- c(9, 1, 2, 6, 3, 0)
length(y)
```

```
[1] 6
```

```
z <- c(x, y)
z
```

```
[1] 2 1 3 4 9 1 2 6 3 0
```

```
length(z)
```

```
[1] 10
```

2.5 Vecteurs nommés

Les différentes valeurs d'un vecteur peuvent être nommées. Une première manière de nommer les éléments d'un vecteur est de le faire à sa création :

```
sexe <- c(
  Michel = "h", Anne = "f",
  Dominique = NA, Jean = "h",
```

```
Claude = NA, Marie = "f"
)
```

Lorsqu'on affiche le vecteur, la présentation change quelque peu.

```
sexe
```

Michel	Anne Dominique	Jean	Claude	Marie
"h"	"f" NA	"h"	NA	"f"

La liste des noms s'obtient avec `names()`.

```
names(sexe)
```

```
[1] "Michel" "Anne" "Dominique" "Jean" "Claude" "Marie"
```

Pour ajouter ou modifier les noms d'un vecteur, on doit attribuer un nouveau vecteur de noms :

```
names(sexe) <- c("Michael", "Anna", "Dom", "John", "Alex", "Mary")
sexe
```

Michael	Anna	Dom	John	Alex	Mary
"h"	"f"	NA	"h"	NA	"f"

Pour supprimer tous les noms, il y a la fonction `unname()` :

```
anonyme <- unname(sexe)
anonyme
```

```
[1] "h" "f" NA "h" NA "f"
```

2.6 Indexation par position

L'indexation est l'une des fonctionnalités les plus puissantes mais aussi les plus difficiles à maîtriser de **R**. Il s'agit d'opérations permettant de sélectionner des sous-ensembles de valeurs en fonction de différents critères. Il existe trois types d'indexation : (i) l'indexation par position, (ii) l'indexation par nom et (iii) l'indexation par condition. Le principe est toujours le même : on indique entre crochets⁷ (`[]`) ce qu'on souhaite garder ou non.

Commençons par l'indexation par position encore appelée indexation directe. Ce mode le plus simple d'indexation consiste à indiquer la position des éléments à conserver.

Reprenons notre vecteur `taille` :

```
taille
```

```
[1] 1.88 1.65 1.92 1.76 NA 1.72
```

Si on souhaite le premier élément du vecteur, on peut faire :

```
taille[1]
```

```
[1] 1.88
```

Si on souhaite les trois premiers éléments ou les éléments 2, 5 et 6 :

```
taille[1:3]
```

```
[1] 1.88 1.65 1.92
```

```
taille[c(2, 5, 6)]
```

```
[1] 1.65 NA 1.72
```

Si on veut le dernier élément :

⁷ Pour rappel, les crochets s'obtiennent sur un clavier français de type PC en appuyant sur la touche Alt Gr et la touche (ou).


```
taille[length(taille)]
```

```
[1] 1.72
```

Il est tout à fait possible de sélectionner les valeurs dans le désordre :

```
taille[c(5, 1, 4, 3)]
```

```
[1] NA 1.88 1.76 1.92
```

Dans le cadre de l'indexation par position, il est également possible de spécifier des nombres négatifs, auquel cas cela signifiera toutes les valeurs sauf celles-là. Par exemple :

```
taille[c(-1, -5)]
```

```
[1] 1.65 1.92 1.76 1.72
```

À noter, si on indique une position au-delà de la longueur du vecteur, **R** renverra NA. Par exemple :

```
taille[23:25]
```

```
[1] NA NA NA
```

2.7 Indexation par nom

Lorsqu'un vecteur est nommé, il est dès lors possible d'accéder à ses valeurs à partir de leur nom. Il s'agit de l'indexation par nom.

```
sexe["Anna"]
```

```
Anna  
"f"
```

```
sexe[c("Mary", "Michael", "John")]
```

Mary	Michael	John
"f"	"h"	"h"

Par contre il n'est pas possible d'utiliser l'opérateur `-` comme pour l'indexation directe. Pour exclure un élément en fonction de son nom, on doit utiliser une autre forme d'indexation, l'indexation par condition, expliquée dans la section suivante. On peut ainsi faire...

```
sexe[names(sexe) != "Dom"]
```

... pour sélectionner tous les éléments sauf celui qui s'appelle Dom.

2.8 Indexation par condition

L'indexation par condition consiste à fournir un vecteur logique indiquant si chaque élément doit être inclus (si `TRUE`) ou exclu (si `FALSE`). Par exemple :

```
sexe
```

Michael	Anna	Dom	John	Alex	Mary
"h"	"f"	NA	"h"	NA	"f"

```
sexe[c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)]
```

Michael	John
"h"	"h"

Écrire manuellement une telle condition n'est pas très pratique à l'usage. Mais supposons que nous ayons également à notre disposition les deux vecteurs suivants, également de longueur 6.

```
urbain <- c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)
poids <- c(80, 63, 75, 87, 82, 67)
```

Le vecteur `urbain` est un vecteur logique. On peut directement l'utiliser pour avoir le sexe des enquêtés habitant en milieu urbain :

```
sexe[urbain]
```

```
Michael    Anna    Mary
    "h"      "f"      "f"
```

Supposons qu'on souhaite maintenant avoir la taille des individus pesant 80 kilogrammes ou plus. Nous pouvons effectuer une comparaison à l'aide des opérateurs de comparaison suivants :

Table 2.2: Opérateurs de comparaison

Opérateur de comparaison	Signification
<code>==</code>	égal à
<code>%in%</code>	appartient à
<code>!=</code>	différent de
<code>></code>	strictement supérieur à
<code><</code>	strictement inférieur à
<code>>=</code>	supérieur ou égal à
<code><=</code>	inférieur ou égal à

Voyons tout de suite un exemple :

```
poids >= 80
```

```
[1] TRUE FALSE FALSE TRUE TRUE FALSE
```

Que s'est-il passé ? Nous avons fourni à **R** une condition et il nous a renvoyé un vecteur logique avec autant d'éléments qu'il y a d'observations et dont la valeur est `TRUE` si la condition

est remplie et **FALSE** dans les autres cas. Nous pouvons alors utiliser ce vecteur logique pour obtenir la taille des participants pesant 80 kilogrammes ou plus :

```
taille[poids >= 80]
```

```
[1] 1.88 1.76 NA
```

On peut combiner ou modifier des conditions à l'aide des opérateurs logiques habituels :

Table 2.3: Opérateurs logiques

Opérateur logique	Signification
&	et logique
	ou logique
!	négation logique

Supposons que je veuille identifier les personnes pesant 80 kilogrammes ou plus **et** vivant en milieu urbain :

```
poids >= 80 & urbain
```

```
[1] TRUE FALSE FALSE FALSE FALSE FALSE
```

Les résultats sont différents si je souhaite isoler les personnes pesant 80 kilogrammes ou plus **ou** vivant milieu urbain :

```
poids >= 80 | urbain
```

```
[1] TRUE TRUE FALSE TRUE TRUE TRUE
```

! Comparaison et valeur manquante

Une remarque importante : quand l'un des termes d'une condition comporte une valeur manquante (**NA**), le résultat

de cette condition n'est pas toujours TRUE ou FALSE, il peut aussi être à son tour une valeur manquante.

```
taille
```

```
[1] 1.88 1.65 1.92 1.76    NA 1.72
```

```
taille > 1.8
```

```
[1]  TRUE FALSE  TRUE FALSE    NA FALSE
```

On voit que le test `NA > 1.8` ne renvoie ni vrai ni faux, mais NA.

Une autre conséquence importante de ce comportement est qu'on ne peut pas utiliser l'opérateur l'expression `== NA` pour tester la présence de valeurs manquantes. On utilisera à la place la fonction *ad hoc* `is.na()` :

```
is.na(taille > 1.8)
```

```
[1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

Pour compliquer encore un peu le tout, lorsqu'on utilise une condition pour l'indexation, si la condition renvoie NA, **R** ne sélectionne pas l'élément mais retourne quand même la valeur NA. Ceci a donc des conséquences sur le résultat d'une indexation par comparaison.

Par exemple si je cherche à connaître le poids des personnes mesurant 1,80 mètre ou plus :

```
taille
```

```
[1] 1.88 1.65 1.92 1.76    NA 1.72
```

```
poids
```

```
[1] 80 63 75 87 82 67
```

```
poids[taille > 1.8]
```

```
[1] 80 75 NA
```

Les éléments pour lesquels la taille n'est pas connue ont été transformés en `NA`, ce qui n'influera pas le calcul d'une moyenne. Par contre, lorsqu'on utilisera assignation et indexation ensemble, cela peut créer des problèmes. Il est donc préférable lorsqu'on a des valeurs manquantes de les exclure ainsi :

```
poids[taille > 1.8 & !is.na(taille)]
```

```
[1] 80 75
```

2.9 Assignation par indexation

L'indexation peut être combinée avec l'assignation (opérateur `<-`) pour modifier seulement certaines parties d'un vecteur. Ceci fonctionne pour les différents types d'indexation évoqués précédemment.

```
v <- 1:5  
v
```

```
[1] 1 2 3 4 5
```

```
v[1] <- 3  
v
```

```
[1] 3 2 3 4 5
```

```
sexe["Alex"] <- "non-binaire"  
sexe
```

Michael	Anna	Dom	John	Alex
"h"	"f"	NA	"h"	"non-binaire"
Mary				
"f"				

Enfin on peut modifier plusieurs éléments d'un seul coup soit en fournissant un vecteur, soit en profitant du mécanisme de recyclage. Les deux commandes suivantes sont ainsi rigoureusement équivalentes :

```
sexe[c(1,3,4)] <- c("Homme", "Homme", "Homme")
sexe[c(1,3,4)] <- "Homme"
```

L'assignation par indexation peut aussi être utilisée pour ajouter une ou plusieurs valeurs à un vecteur :

```
length(sexe)
```

```
[1] 6
```

```
sexe[7] <- "f"
sexe
```

Michael	Anna	Dom	John	Alex
"Homme"	"f"	"Homme"	"Homme"	"non-binaire"
Mary				
"f"	"f"			

```
length(sexe)
```

```
[1] 7
```

2.10 En résumé

- Un vecteur est un objet unidimensionnel contenant une liste de valeurs qui sont toutes du même type (entières, numériques, textuelles ou logiques).
- La fonction `class()` permet de connaître le type du vecteur et la fonction `length()` sa longueur, c'est-à-dire son nombre d'éléments.
- La fonction `c()` sert à créer et à combiner des vecteurs.

- Les valeurs manquantes sont représentées avec `NA`.
- Un vecteur peut être nommé, c'est-à-dire qu'un nom textuel a été associé à chaque élément. Cela peut se faire lors de sa création ou avec la fonction `names()`.
- L'indexation consiste à extraire certains éléments d'un vecteur. Pour cela, on indique ce qu'on souhaite extraire entre crochets (`[]`) juste après le nom du vecteur. Le type d'indexation dépend du type d'information transmise.
- S'il s'agit de nombres entiers, c'est l'indexation par position : les nombres représentent la position dans le vecteur des éléments qu'on souhaite extraire. Un nombre négatif s'interprète comme tous les éléments sauf celui-là.
- Si on indique des chaînes de caractères, c'est l'indexation par nom : on indique le nom des éléments qu'on souhaite extraire. Cette forme d'indexation ne fonctionne que si le vecteur est nommé.
- Si on transmet des valeurs logiques, le plus souvent sous la forme d'une condition, c'est l'indexation par condition : `TRUE` indique les éléments à extraire et `FALSE` les éléments à exclure. Il faut être vigilant aux valeurs manquantes (`NA`) dans ce cas précis.
- Enfin, il est possible de ne modifier que certains éléments d'un vecteur en ayant recours à la fois à l'indexation (`[]`) et à l'assignation (`<-`).

2.11 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur [YouTube](https://youtu.be/Eh8piunoqQc).

<https://youtu.be/Eh8piunoqQc>

3 Listes

Par nature, les vecteurs ne peuvent contenir que des valeurs de même type (numérique, textuel ou logique). Or, on peut avoir besoin de représenter des objets plus complexes composés d'éléments disparates. C'est ce que permettent les listes.

3.1 Propriétés et création

Une liste se crée tout simplement avec la fonction `list()` :

```
l1 <- list(1:5, "abc")  
l1
```

```
[[1]]  
[1] 1 2 3 4 5
```

```
[[2]]  
[1] "abc"
```

Une liste est un ensemble d'objets, quels qu'ils soient, chaque élément d'une liste pouvant avoir ses propres dimensions. Dans notre exemple précédent, nous avons créé une liste `l1` composée de deux éléments : un vecteur d'entiers de longueur 5 et un vecteur textuel de longueur 1. La longueur d'une liste correspond aux nombres d'éléments qu'elle contient et s'obtient avec `length()` :

```
length(l1)
```

```
[1] 2
```

Comme les vecteurs, une liste peut être nommée et les noms des éléments d'une liste sont accessibles avec `names()` :

```
l2 <- list(  
  minuscules = letters,  
  majuscules = LETTERS,  
  mois = month.name  
)  
l2
```

`$minuscules`

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
[20] "t" "u" "v" "w" "x" "y" "z"
```

`$majuscules`

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

`$mois`

```
[1] "January"    "February"    "March"       "April"       "May"         "June"  
[7] "July"       "August"      "September"   "October"     "November"    "December"
```

```
length(l2)
```

```
[1] 3
```

```
names(l2)
```

```
[1] "minuscules" "majuscules" "mois"
```

Que se passe-t-il maintenant si on effectue la commande suivante ?

```
l <- list(l1, l2)
```

À votre avis, quelle est la longueur de cette nouvelle liste `l` ?
5 ?

```
length(l)
```

```
[1] 2
```

Eh bien non ! Elle est de longueur 2 car nous avons créé une liste composée de deux éléments qui sont eux-mêmes des listes. Cela est plus lisible si on fait appel à la fonction `str()` qui permet de visualiser la structure d'un objet.

```
str(l)
```

```
List of 2
 $ :List of 2
  ..$ : int [1:5] 1 2 3 4 5
  ..$ : chr "abc"
 $ :List of 3
  ..$ minuscules: chr [1:26] "a" "b" "c" "d" ...
  ..$ majuscules: chr [1:26] "A" "B" "C" "D" ...
  ..$ mois      : chr [1:12] "January" "February" "March" "April" ...
```

Une liste peut contenir tous types d'objets, y compris d'autres listes. Pour combiner les éléments d'une liste, il faut utiliser la fonction `append()` :

```
l <- append(l1, l2)
length(l)
```

```
[1] 5
```

```
str(l)
```

```
List of 5
 $      : int [1:5] 1 2 3 4 5
 $      : chr "abc"
 $ minuscules: chr [1:26] "a" "b" "c" "d" ...
 $ majuscules: chr [1:26] "A" "B" "C" "D" ...
 $ mois      : chr [1:12] "January" "February" "March" "April" ...
```

i Note

On peut noter en passant qu'une liste peut tout à fait n'être que partiellement nommée.

3.2 Indexation

Les crochets simples (`[]`) fonctionnent comme pour les vecteurs. On peut utiliser à la fois l'indexation par position, l'indexation par nom et l'indexation par condition.

```
1
```

```
[[1]]
```

```
[1] 1 2 3 4 5
```

```
[[2]]
```

```
[1] "abc"
```

```
$minuscules
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
```

```
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
$majuscules
```

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
```

```
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
$mois
```

```
[1] "January" "February" "March" "April" "May" "June"
```

```
[7] "July" "August" "September" "October" "November" "December"
```

```
1[c(1,3,4)]
```

```
[[1]]
```

```
[1] 1 2 3 4 5
```

```
$minuscules
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
$majuscules
```

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
l[c("majuscules", "minuscules")]
```

```
$majuscules
```

```
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
$minuscules
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
l[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

```
[[1]]
```

```
[1] 1 2 3 4 5
```

```
[[2]]
```

```
[1] "abc"
```

```
$mois
```

```
[1] "January" "February" "March" "April" "May" "June"
[7] "July" "August" "September" "October" "November" "December"
```

Même si on extrait un seul élément, l'extraction obtenue avec les crochets simples renvoie toujours une liste, ici composée d'un seul élément :

```
str(l[1])
```

```
List of 1
```

```
$ : int [1:5] 1 2 3 4 5
```

Supposons que je souhaite calculer la moyenne des valeurs du premier élément de ma liste. Essayons la commande suivante :

```
mean(l[1])
```

```
Warning in mean.default(l[1]): l'argument n'est ni numérique, ni logique :  
renvoi de NA
```

```
[1] NA
```

Nous obtenons un message d'erreur. En effet, **R** ne sait pas calculer une moyenne à partir d'une liste. Ce qu'il lui faut, c'est un vecteur de valeurs numériques. Autrement dit, ce que nous cherchons à obtenir c'est le contenu même du premier élément de notre liste et non une liste à un seul élément.

C'est ici que les doubles crochets (`[[]]`) vont rentrer en jeu. Pour ces derniers, nous pourrions utiliser l'indexation par position ou l'indexation par nom, mais pas l'indexation par condition. De plus, le critère qu'on indiquera doit indiquer **un et un seul** élément de notre liste. Au lieu de renvoyer une liste à un élément, les doubles crochets vont renvoyer l'élément désigné.

```
str(l[[1]])
```

```
List of 1  
 $ : int [1:5] 1 2 3 4 5
```

```
str(l[[1]])
```

```
int [1:5] 1 2 3 4 5
```

Maintenant, nous pouvons calculer notre moyenne :

```
mean(l[[1]])
```

```
[1] 3
```

Nous pouvons aussi utiliser l'indexation par nom.

```
l[["mois"]]
```

```
[1] "January" "February" "March"    "April"    "May"      "June"
[7] "July"    "August"   "September" "October"  "November" "December"
```

Mais il faut avouer que cette écriture avec doubles crochets et guillemets est un peu lourde. Heureusement, un nouvel acteur entre en scène : le symbole dollar (\$). C'est un raccourci des doubles crochets pour l'indexation par nom qu'on utilise ainsi :

```
l$mois
```

```
[1] "January" "February" "March"    "April"    "May"      "June"
[7] "July"    "August"   "September" "October"  "November" "December"
```

Les écritures `l$mois` et `l[["mois"]]` sont équivalentes. Attention ! Cela ne fonctionne que pour l'indexation par nom.

```
l$1
```

Error: unexpected numeric constant in "l\$1"

L'assignation par indexation fonctionne également avec les doubles crochets ou le signe dollar :

```
l[[2]] <- list(c("un", "vecteur", "textuel"))
l$mois <- c("Janvier", "Février", "Mars")
l
```

```

[[1]]
[1] 1 2 3 4 5

[[2]]
[[2]][[1]]
[1] "un"          "vecteur" "textuel"

$minuscules
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

$majuscules
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

$mois
[1] "Janvier" "Février" "Mars"

```

3.3 En résumé

- Les listes sont des objets unidimensionnels pouvant contenir tout type d'objet, y compris d'autres listes.
- Elles ont une longueur qu'on obtient avec `length()`.
- On crée une liste avec `list()` et on peut fusionner des listes avec `append()`.
- Tout comme les vecteurs, les listes peuvent être nommées et les noms des éléments s'obtiennent avec `base::names()`.
- Les crochets simples (`[]`) permettent de sélectionner les éléments d'une liste, en utilisant l'indexation par position, l'indexation par nom ou l'indexation par condition. Cela renvoie toujours une autre liste.
- Les doubles crochets (`[[]]`) renvoient directement le contenu d'un élément de la liste qu'on aura sélectionné par position ou par nom.
- Le symbole `$` est un raccourci pour facilement sélectionner un élément par son nom, `liste$nom` étant équivalent à `liste[["nom"]]`.

3.4 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur [YouTube](#).

<https://youtu.be/Eh8piunoqQc>

4 Tableaux de données

Les tableaux de données, ou *data frame* en anglais, est un type d'objets essentiel pour les données d'enquêtes.

4.1 Propriétés et création

Dans **R**, les tableaux de données sont tout simplement des listes (voir Chapitre 3) avec quelques propriétés spécifiques :

- les tableaux de données ne peuvent contenir que des vecteurs ;
- tous les vecteurs d'un tableau de données ont la même longueur ;
- tous les éléments d'un tableau de données sont nommés et ont chacun un nom unique.

Dès lors, un tableau de données correspond aux fichiers de données qu'on a l'habitude de manipuler dans d'autres logiciels de statistiques comme **SPSS** ou **Stata**. Les variables sont organisées en colonnes et les observations en lignes.

On peut créer un tableau de données avec la fonction `data.frame()` :

```
df <- data.frame(  
  sexe = c("f", "f", "h", "h"),  
  age = c(52, 31, 29, 35),  
  blond = c(FALSE, TRUE, TRUE, FALSE)  
)  
df
```

```
sexe age blond  
1    f  52 FALSE
```

```
2    f  31  TRUE
3    h  29  TRUE
4    h  35 FALSE
```

```
str(df)
```

```
'data.frame':  4 obs. of  3 variables:
 $ sexe : chr  "f" "f" "h" "h"
 $ age  : num  52 31 29 35
 $ blond: logi  FALSE TRUE TRUE FALSE
```

Un tableau de données étant une liste, la fonction `length()` renverra le nombre d'éléments de la liste, donc dans le cas présent le nombre de variables, et `names()` leurs noms :

```
length(df)
```

```
[1] 3
```

```
names(df)
```

```
[1] "sexe" "age"  "blond"
```

Comme tous les éléments d'un tableau de données ont la même longueur, cet objet peut être vu comme bidimensionnel. Les fonctions `nrow()`, `ncol()` et `dim()` donnent respectivement le nombre de lignes, le nombre de colonnes et les dimensions de notre tableau.

```
nrow(df)
```

```
[1] 4
```

```
ncol(df)
```

```
[1] 3
```

```
dim(df)
```

```
[1] 4 3
```

De plus, tout comme les colonnes ont un nom, il est aussi possible de nommer les lignes avec `row.names()` :

```
row.names(df) <- c("Anna", "Mary-Ann", "Michael", "John")
df
```

	sexe	age	blond
Anna	f	52	FALSE
Mary-Ann	f	31	TRUE
Michael	h	29	TRUE
John	h	35	FALSE

4.2 Indexation

Les tableaux de données étant des listes, nous pouvons donc utiliser les crochets simples (`[]`), les crochets doubles (`[[[]]`) et le symbole dollar (`$`) pour extraire des parties de notre tableau, de la même manière que pour n'importe quelle liste.

```
df[1]
```

	sexe
Anna	f
Mary-Ann	f
Michael	h
John	h

```
df[[1]]
```

```
[1] "f" "f" "h" "h"
```

```
df$sexe
```

```
[1] "f" "f" "h" "h"
```

Cependant, un tableau de données étant un objet bidimensionnel, il est également possible d'extraire des données sur deux dimensions, à savoir un premier critère portant sur les lignes et un second portant sur les colonnes. Pour cela, nous utiliserons les crochets simples (`[]`) en séparant nos deux critères par une virgule (`,`).

Un premier exemple :

```
df
```

	sexe	age	blond
Anna	f	52	FALSE
Mary-Ann	f	31	TRUE
Michael	h	29	TRUE
John	h	35	FALSE

```
df[3, 2]
```

```
[1] 29
```

Cette première commande indique que nous souhaitons la troisième ligne de la seconde colonne, autrement dit l'âge de Michael. Le même résultat peut être obtenu avec l'indexation par nom, l'indexation par condition, ou un mélange de tout ça.

```
df["Michael", "age"]
```

```
[1] 29
```

```
df[c(F, F, T, F), c(F, T, F)]
```

```
[1] 29
```

```
df[3, "age"]
```

```
[1] 29
```

```
df["Michael", 2]
```

```
[1] 29
```

Il est également possible de préciser un seul critère. Par exemple, si je souhaite les deux premières observations, ou les variables *sexe* et *blond* :

```
df[1:2,]
```

	sexe	age	blond
Anna	f	52	FALSE
Mary-Ann	f	31	TRUE

```
df[,c("sexe", "blond")]
```

	sexe	blond
Anna	f	FALSE
Mary-Ann	f	TRUE
Michael	h	TRUE
John	h	FALSE

Il a suffi de laisser un espace vide avant ou après la virgule.

Avertissement

ATTENTION ! Il est cependant impératif de laisser la virgule pour indiquer à **R** qu'on souhaite effectuer une indexation à deux dimensions. Si on oublie la virgule, cela nous ramène au mode de fonctionnement des listes. Et le résultat n'est pas forcément le même :

```
df[2, ]
```

```
      sexe age blond  
Mary-Ann   f  31  TRUE
```

```
df[, 2]
```

```
[1] 52 31 29 35
```

```
df[2]
```

```
      age  
Anna    52  
Mary-Ann 31  
Michael 29  
John    35
```

Note

Au passage, on pourra noter quelques subtilités sur le résultat renvoyé.

```
str(df[2, ])
```

```
'data.frame':  1 obs. of  3 variables:  
 $ sexe : chr "f"  
 $ age  : num 31  
 $ blond: logi TRUE
```

```
str(df[, 2])
```

```
num [1:4] 52 31 29 35
```

```
str(df[2])
```

```
'data.frame':  4 obs. of  1 variable:  
 $ age: num 52 31 29 35
```

```
str(df[[2]])
```

```
num [1:4] 52 31 29 35
```

`df[2,]` signifie qu'on veut toutes les variables pour le second individu. Le résultat est un tableau de données à une ligne et trois colonnes. `df[2]` correspond au mode d'extraction des listes et renvoie donc une liste à un élément, en l'occurrence un tableau de données à quatre observations et une variable. `df[[2]]` quant à lui renvoie le contenu de cette variable, soit un vecteur numérique de longueur quatre. Reste `df[, 2]` qui renvoie toutes les observations pour la seconde colonne. Or l'indexation bidimensionnelle a un fonctionnement un peu particulier : par défaut elle renvoie un tableau de données mais s'il y a une seule variable dans l'extraction, c'est un vecteur qui est renvoyé. Pour plus de détails, on pourra consulter l'entrée d'aide `help("data.frame")`.

4.3 Afficher les données

Prenons un tableau de données un peu plus conséquent, en l'occurrence le jeu de données `?questionr::hdv2003` disponible dans l'extension `{questionr}` et correspondant à un extrait de l'enquête *Histoire de vie* réalisée par l'INSEE en 2003. Il contient 2000 individus et 20 variables.

```
library(questionr)
data(hdv2003)
```

Si on demande d'afficher l'objet `hdv2003` dans la console (résultat non reproduit ici), **R** va afficher l'ensemble du contenu de `hdv2003` à l'écran ce qui, sur un tableau de cette taille, ne sera pas très lisible. Pour une exploration visuelle, le plus simple est souvent d'utiliser la visionneuse intégrée à **RStudio** et qu'on peut appeler avec la fonction `View()`.

View(hdv2003)

	id	age	sexe	nivetud	poids	occup	qualif	freres.sc
1	1	28	Femme	Enseignement superieur y compris technique sup...	2634.3982	Exerce une profession	Employe	
2	2	23	Femme	NA	9738.3958	Etudiant, eleve	NA	
3	3	59	Homme	Derniere annee d'etudes primaires	3994.1025	Exerce une profession	Technicien	
4	4	34	Homme	Enseignement superieur y compris technique sup...	5731.6615	Exerce une profession	Technicien	
5	5	71	Femme	Derniere annee d'etudes primaires	4329.0940	Retraite	Employe	
6	6	35	Femme	Enseignement technique ou professionnel court	8674.6994	Exerce une profession	Employe	
7	7	60	Femme	Derniere annee d'etudes primaires	6165.8035	Au foyer	Ouvrier qualifie	
8	8	47	Homme	Enseignement technique ou professionnel court	12891.6408	Exerce une profession	Ouvrier qualifie	
9	9	20	Femme	NA	7808.8721	Etudiant, eleve	NA	
10	10	28	Homme	Enseignement technique ou professionnel long	2277.1605	Exerce une profession	Autre	
11	11	65	Femme	Enseignement superieur y compris technique sup...	704.3227	Retraite	Employe	
12	12	47	Homme	2eme cycle	6697.8682	Exerce une profession	Ouvrier qualifie	
13	13	63	Femme	Derniere annee d'etudes primaires	7118.4659	Retraite	Employe	
14	14	67	Femme	Enseignement technique ou professionnel court	586.7714	Exerce une profession	NA	
15	15	76	Femme	A arrete ses etudes, avant la derniere annee d'et...	11042.0774	Retraite	NA	
16	16	49	Femme	Enseignement technique ou professionnel court	9958.2287	Exerce une profession	Employe	
17	17	62	Homme	Enseignement superieur y compris technique sup...	4836.1393	Retraite	Cadre	
18	18	20	Femme	NA	1551.4846	Etudiant, eleve	NA	

Showing 1 to 19 of 2,000 entries

Figure 4.1: Interface View() de R RStudio

Les fonctions `head()` et `tail()`, qui marchent également sur les vecteurs, permettent d'afficher seulement les premières (respectivement les dernières) lignes d'un tableau de données :

`head(hdv2003)`

```

id age  sexe                                nivetud    poids
1  1  28  Femme Enseignement superieur y compris technique superieur 2634.398
2  2  23  Femme                                <NA> 9738.396
3  3  59  Homme                                Derniere annee d'etudes primaires 3994.102
4  4  34  Homme Enseignement superieur y compris technique superieur 5731.662
5  5  71  Femme                                Derniere annee d'etudes primaires 4329.094
6  6  35  Femme      Enseignement technique ou professionnel court 8674.699
      occup    qualif freres.soeurs clso
1 Exerce une profession    Employe      8 Oui
2      Etudiant, eleve    <NA>      2 Oui
3 Exerce une profession Technicien      2 Non
4 Exerce une profession Technicien      1 Non
5      Retraite    Employe      0 Oui
6 Exerce une profession    Employe      5 Non
      relig                                trav.imp    trav.satisf
1 Ni croyance ni appartenance      Peu important Insatisfaction

```

2	Ni croyance ni appartenance					<NA>		<NA>
3	Ni croyance ni appartenance	Aussi important que le reste						Equilibre
4	Appartenance sans pratique	Moins important que le reste						Satisfaction
5	Pratiquant regulier					<NA>		<NA>
6	Ni croyance ni appartenance						Le plus important	Equilibre
	hard.rock	lecture.bd	peche.chasse	cuisine	bricol	cinema	sport	heures.tv
1	Non	Non	Non	Oui	Non	Non	Non	0
2	Non	Non	Non	Non	Non	Oui	Oui	1
3	Non	Non	Non	Non	Non	Non	Oui	0
4	Non	Non	Non	Oui	Oui	Oui	Oui	2
5	Non	Non	Non	Non	Non	Non	Non	3
6	Non	Non	Non	Non	Non	Oui	Oui	2

```
tail(hdv2003, 2)
```

	id	age	sexe			nivetud		poids
1999	1999	24	Femme	Enseignement technique ou professionnel	court			13740.810
2000	2000	66	Femme	Enseignement technique ou professionnel	long			7709.513
				occup	qualif	freres.soeurs	clso	
1999	Exerce une profession	Employe		2	Non			
2000		Au foyer	Employe	3	Non			
				relig		trav.imp	trav.satisf	
1999	Appartenance sans pratique	Moins important que le reste						Equilibre
2000	Appartenance sans pratique				<NA>			<NA>
	hard.rock	lecture.bd	peche.chasse	cuisine	bricol	cinema	sport	heures.tv
1999	Non	Non	Non	Non	Non	Oui	Non	0.3
2000	Non	Oui	Non	Oui	Non	Non	Non	0.0

L'extension {dplyr} propose une fonction `dplyr::glimpse()` (ce qui signifie aperçu en anglais) qui permet de visualiser rapidement et de manière condensée le contenu d'un tableau de données.

```
library(dplyr)
glimpse(hdv2003)
```

```
Rows: 2,000
Columns: 20
```

```

$ id          <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
$ age         <int> 28, 23, 59, 34, 71, 35, 60, 47, 20, 28, 65, 47, 63, 67, ~
$ sexe        <fct> Femme, Femme, Homme, Homme, Femme, Femme, Femme, Homme, ~
$ nivetud     <fct> "Enseignement superieur y compris technique superieur", ~
$ poids       <dbl> 2634.3982, 9738.3958, 3994.1025, 5731.6615, 4329.0940, 8~
$ occup       <fct> "Exerce une profession", "Etudiant, eleve", "Exerce une ~
$ qualif      <fct> Employe, NA, Technicien, Technicien, Employe, Employe, 0~
$ freres.soeurs <int> 8, 2, 2, 1, 0, 5, 1, 5, 4, 2, 3, 4, 1, 5, 2, 3, 4, 0, 2,~
$ clso        <fct> Oui, Oui, Non, Non, Oui, Non, Oui, Non, Oui, Non, Oui, 0~
$ relig       <fct> Ni croyance ni appartenance, Ni croyance ni appartenance~
$ trav.imp    <fct> Peu important, NA, Aussi important que le reste, Moins i~
$ trav.satisf <fct> Insatisfaction, NA, Equilibre, Satisfaction, NA, Equilib~
$ hard.rock   <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ lecture.bd  <fct> Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, Non, N~
$ peche.chasse <fct> Non, Non, Non, Non, Non, Non, Non, Oui, Oui, Non, Non, Non, N~
$ cuisine     <fct> Oui, Non, Non, Oui, Non, Non, Oui, Oui, Non, Non, Oui, N~
$ bricol      <fct> Non, Non, Non, Oui, Non, Non, Non, Oui, Non, Non, Oui, 0~
$ cinema      <fct> Non, Oui, Non, Oui, Non, Oui, Non, Non, Oui, Oui, Oui, N~
$ sport       <fct> Non, Oui, Oui, Oui, Non, Oui, Non, Non, Non, Oui, Non, 0~
$ heures.tv   <dbl> 0.0, 1.0, 0.0, 2.0, 3.0, 2.0, 2.9, 1.0, 2.0, 2.0, 1.0, 0~

```

L'extension {labelled} propose une fonction `labelled::look_for()` qui permet de lister les différentes variables d'un fichier de données :

```

library(labelled)
look_for(hdv2003)

```

pos	variable	label	col_type	values
1	id	-	int	
2	age	-	int	
3	sexe	-	fct	Homme Femme
4	nivetud	-	fct	N'a jamais fait d'etudes A arrete ses etudes, avant la derniere ann~ Derniere annee d'etudes primaires 1er cycle 2eme cycle Enseignement technique ou professionnel co~ Enseignement technique ou professionnel lo~

				Enseignement superieur y compris technique~
5	poids	-	dbl	
6	occup	-	fct	Exerce une profession Chomeur Etudiant, eleve Retraite Retire des affaires Au foyer Autre inactif
7	qualif	-	fct	Ouvrier specialise Ouvrier qualifie Technicien Profession intermediaire Cadre Employe Autre
8	freres.soeurs	-	int	
9	clso	-	fct	Oui Non Ne sait pas
10	relig	-	fct	Pratiquant regulier Pratiquant occasionnel Appartenance sans pratique Ni croyance ni appartenance Rejet NSP ou NVPR
11	trav.imp	-	fct	Le plus important Aussi important que le reste Moins important que le reste Peu important
12	trav.satisf	-	fct	Satisfaction Insatisfaction Equilibre
13	hard.rock	-	fct	Non Oui
14	lecture.bd	-	fct	Non Oui
15	peche.chasse	-	fct	Non Oui
16	cuisine	-	fct	Non Oui

17	bricol	-	fct	Non Oui
18	cinema	-	fct	Non Oui
19	sport	-	fct	Non Oui
20	heures.tv	-	dbl	

Lorsqu'on a un gros tableau de données avec de nombreuses variables, il peut être difficile de retrouver la ou les variables d'intérêt. Il est possible d'indiquer à `labelled::look_for()` un mot-clé pour limiter la recherche. Par exemple :

```
look_for(hdv2003, "trav")
```

pos	variable	label	col_type	values
11	trav.imp	-	fct	Le plus important Aussi important que le reste Moins important que le reste Peu important
12	trav.satisf	-	fct	Satisfaction Insatisfaction Equilibre

Il est à noter que si la recherche n'est pas sensible à la casse (i.e. aux majuscules et aux minuscules), elle est sensible aux accents.

La méthode `summary()` qui fonctionne sur tout type d'objet permet d'avoir quelques statistiques de base sur les différentes variables de notre tableau, les statistiques affichées dépendant du type de variable.

```
summary(hdv2003)
```

id		age		sexe	
Min.	: 1.0	Min.	:18.00	Homme:	899
1st Qu.:	500.8	1st Qu.:	35.00	Femme:	1101
Median	:1000.5	Median	:48.00		

Mean :1000.5 Mean :48.16
 3rd Qu.:1500.2 3rd Qu.:60.00
 Max. :2000.0 Max. :97.00

	nivetud	poids
Enseignement technique ou professionnel court	:463	Min. : 78.08
Enseignement superieur y compris technique superieur:	441	1st Qu.: 2221.82
Derniere annee d'etudes primaires	:341	Median : 4631.19
1er cycle	:204	Mean : 5535.61
2eme cycle	:183	3rd Qu.: 7626.53
(Other)	:256	Max. :31092.14
NA's	:112	

	occup	qualif	freres.soeurs
Exerce une profession:	1049	Employe :594	Min. : 0.000
Chomeur : 134		Ouvrier qualifie :292	1st Qu.: 1.000
Etudiant, eleve : 94		Cadre :260	Median : 2.000
Retraite : 392		Ouvrier specialise :203	Mean : 3.283
Retire des affaires : 77		Profession intermediaire:160	3rd Qu.: 5.000
Au foyer : 171		(Other) :144	Max. :22.000
Autre inactif : 83		NA's :347	

	clso	relig
Oui : 936		Pratiquant regulier :266
Non :1037		Pratiquant occasionnel :442
Ne sait pas: 27		Appartenance sans pratique :760
		Ni croyance ni appartenance:399
		Rejet : 93
		NSP ou NVPR : 40

	trav.imp	trav.satisf	hard.rock	lecture.bd
Le plus important : 29		Satisfaction :480	Non:1986	Non:1953
Aussi important que le reste:259		Insatisfaction:117	Oui: 14	Oui: 47
Moins important que le reste:708		Equilibre :451		
Peu important : 52		NA's :952		
NA's :952				

peche.chasse	cuisine	bricol	cinema	sport	heures.tv
Non:1776	Non:1119	Non:1147	Non:1174	Non:1277	Min. : 0.000
Oui: 224	Oui: 881	Oui: 853	Oui: 826	Oui: 723	1st Qu.: 1.000
					Median : 2.000
					Mean : 2.247

```
3rd Qu.: 3.000
Max.    :12.000
NA's    :5
```

On peut également appliquer `summary()` à une variable particulière.

```
summary(hdv2003$sexe)
```

```
Homme Femme
899  1101
```

```
summary(hdv2003$age)
```

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
18.00  35.00   48.00   48.16  60.00   97.00
```

4.4 En résumé

- Les tableaux de données sont des listes avec des propriétés particulières :
 - i. tous les éléments sont des vecteurs ;
 - ii. tous les vecteurs ont la même longueur ;
 - iii. tous les vecteurs ont un nom et ce nom est unique.
- On peut créer un tableau de données avec `data.frame()`.
- Les tableaux de données correspondent aux fichiers de données qu'on utilise usuellement dans d'autres logiciels de statistiques : les variables sont représentées en colonnes et les observations en lignes.
- Ce sont des objets bidimensionnels : `ncol()` renvoie le nombre de colonnes et `nrow()` le nombre de lignes.
- Les doubles crochets (`[[]]`) et le symbole dollar (`$`) fonctionnent comme pour les listes et permettent d'accéder aux variables.
- Il est possible d'utiliser des coordonnées bidimensionnelles avec les crochets simples (`[]`) en indiquant un critère sur les lignes puis un critère sur les colonnes, séparés par une virgule (`,`).

4.5 webin-R

On pourra également se référer au webin-R #02 (*les bases du langage R*) sur [YouTube](#).

<https://youtu.be/Eh8piunoqQc>

5 Tibbles

5.1 Le concept de tidy data

Le `{tidyverse}` est en partie fondé sur le concept de *tidy data*, développé à l'origine par Hadley Wickham dans un [article de 2014](#) du *Journal of Statistical Software*.

Il s'agit d'un modèle d'organisation des données qui vise à faciliter le travail souvent long et fastidieux de nettoyage et de préparation préalable à la mise en oeuvre de méthodes d'analyse.

Les principes d'un jeu de données *tidy* sont les suivants :

1. chaque variable est une colonne
2. chaque observation est une ligne
3. chaque type d'observation est dans une table différente

Un chapitre dédié à `{tidyr}` (voir Chapitre [17](#)) présente comment définir et rendre des données *tidy* avec ce package.

Les extensions du `{tidyverse}`, notamment `{ggplot2}` et `{dplyr}`, sont prévues pour fonctionner avec des données *tidy*.

5.2 tibbles : des tableaux de données améliorés

Une autre particularité du `{tidyverse}` est que ces extensions travaillent avec des tableaux de données au format `tibble::tibble()`, qui est une évolution plus moderne du classique `data.frame` de **R** de base.

Ce format est fourni est géré par l'extension du même nom (`{tibble}`), qui fait partie du coeur du *tidyverse*. La plupart des fonctions des extensions du *tidyverse* acceptent des *data.frames* en entrée, mais retournent un *tibble*.

Contrairement aux *data.frames*, les *tibbles* :

- n'ont pas de noms de lignes (*rownames*)
- autorisent des noms de colonnes invalides pour les *data.frames* (espaces, caractères spéciaux, nombres...) ⁸
- s'affichent plus intelligemment que les *data.frames* : seules les premières lignes sont affichées, ainsi que quelques informations supplémentaires utiles (dimensions, types des colonnes...)
- ne font pas de *partial matching* sur les noms de colonnes ⁹
- affichent un avertissement si on essaie d'accéder à une colonne qui n'existe pas

⁸ Quand on veut utiliser des noms de ce type, on doit les entourer avec des *backticks* (```)

⁹ Dans **R** base, si une table `d` contient une colonne `qualif`, `d$qual` retournera cette colonne.

Pour autant, les *tibbles* restent compatibles avec les *data.frames*.

Il est possible de créer un *tibble* manuellement avec `tibble::tibble()`.

```
library(tidyverse)
tibble(
  x = c(1.2345, 12.345, 123.45, 1234.5, 12345),
  y = c("a", "b", "c", "d", "e")
)
```

```
# A tibble: 5 x 2
      x y
  <dbl> <chr>
1   1.23 a
2  12.3  b
3  123.  c
4 1234.  d
5 12345  e
```

On peut ainsi facilement convertir un *data.frame* en *tibble* avec `tibble::as_tibble()` :

```
d <- as_tibble(mtcars)
d
```

```
# 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
# ... with 22 more rows
```

D'ailleurs, quand on regarde la classe d'un tibble, on peut s'apercevoir qu'un tibble hérite de la classe `data.frame` mais possède en plus la classe `tbl_df`. Cela traduit bien le fait que les *tibbles* restent des *data frames*.

```
class(d)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

Si le *data frame* d'origine a des *rownames*, on peut d'abord les convertir en colonnes avec `tibble::rownames_to_columns()` :

```
d <- as_tibble(rownames_to_column(mtcars))
d
```

```
# A tibble: 32 x 12
  rowname      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
# ... with 22 more rows

```

À l'inverse, on peut à tout moment convertir un tibble en *data frame* avec `tibble::as.data.frame()` :

```
as.data.frame(d)
```

	rowname	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
13	Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
14	Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
15	Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
16	Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
17	Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
18	Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
20	Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
21	Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
22	Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
23	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
24	Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4

25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
27	Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
28	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
29	Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
30	Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
31	Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
32	Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Là encore, on peut convertir la colonne *rowname* en “vrais”
rownames avec `tibble::column_to_rownames()` :

```
column_to_rownames(as.data.frame(d))
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2

Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Note

Les deux fonctions `tibble::column_to_rownames()` et `tibble::rownames_to_column()` acceptent un argument supplémentaire `var` qui permet d'indiquer un nom de colonne autre que le nom `rowname` utilisé par défaut pour créer ou identifier la colonne contenant les noms de lignes.

5.3 Données et tableaux imbriqués

Une des particularités des *tibbles* est qu'ils acceptent, à la différence des *data frames*, des colonnes composées de listes et, par extension, d'autres tibbles (qui sont des listes) !

```
d <- tibble(
  g = c(1, 2, 3),
  data = list(
    tibble(x = 1, y = 2),
    tibble(x = 4:5, y = 6:7),
    tibble(x = 10)
  )
)
d
```

```
# A tibble: 3 x 2
      g data
<dbl> <list>
1     1 <tibble [1 x 2]>
2     2 <tibble [2 x 2]>
3     3 <tibble [1 x 1]>
```

```
d$data[[2]]
```

```
# A tibble: 2 x 2
      x     y
<int> <int>
1     4     6
2     5     7
```

Cette fonctionnalité, combinée avec les fonctions de `{tidyr}` et de `{purrr}`, s'avère très puissante pour réaliser des opérations multiples en peu de ligne de code.

Dans l'exemple ci-dessous, nous réalisons des régressions linéaires par sous-groupe et les présentons dans un même tableau. Pour le moment, le code présenté doit vous sembler complexe et un peu obscur. Pas de panique : tout cela sera clarifié dans les différents chapitres de ce guide. Ce qu'il y a à retenir pour le moment, c'est la possibilité de stocker, dans les colonnes d'un *tibble*, différents types de données, y compris des sous-tableaux, des résultats de modèles et même des tableaux mis en forme.

```
reg <-
  iris |>
  group_by(Species) |>
  nest() |>
  mutate(
    model = map(
      data,
      ~ lm(Sepal.Length ~ Petal.Length + Petal.Width, data = .)
    ),
    tbl = map(model, gtsummary::tbl_regression)
  )
reg
```

```
# A tibble: 3 x 4
# Groups:   Species [3]
  Species    data      model  tbl
<fct>    <list>    <list> <list>
```

```
1 setosa      <tibble [50 x 4]> <lm>      <tbl_rgrs>
2 versicolor <tibble [50 x 4]> <lm>      <tbl_rgrs>
3 virginica   <tibble [50 x 4]> <lm>      <tbl_rgrs>
```

```
gtsummary::tbl_merge(
  reg$tbl,
  tab_spanner = paste0("***", reg$Species, "***")
)
```

Table printed with ``knitr::kable()``, not `{gt}`. Learn why at <https://www.danielsjoberg.com/gtsummary/articles/rmarkdown.html>
To suppress this message, include ``message = FALSE`` in code chunk header.

Character	Beta	95% CI	p-value	Beta	95% CI	p-value	Beta	95% CI	p-value
Petal.Length	0.41	-0.20, 1.0	0.2	0.93	0.59, 1.3	<0.001	1.0	0.81, 1.2	<0.001
Petal.Width	0.71	-0.27, 1.7	0.2	-	0.32 1.1, 0.49	0.4	0.01	-0.35, 0.37	>0.9

6 Attributs

Les objets **R** peuvent avoir des attributs qui correspondent en quelque sorte à des métadonnées associées à l'objet en question. Techniquement, un attribut peut être tout type d'objet **R** (un vecteur, une liste, une fonction...).

Parmi les attributs les plus courants, on retrouve notamment :

- `class` : la classe de l'objet
- `length` : sa longueur
- `names` : les noms donnés aux éléments de l'objet
- `levels` : pour les facteurs, les étiquettes des différents niveaux
- `label` : une étiquette de variable

La fonction `attributes()` permet de lister tous les attributs associés à un objet.

```
attributes(iris)
```

```
$names
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
$class
```

```
[1] "data.frame"
```

```
$row.names
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18  
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36  
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54  
[55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72  
[73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90  
[91] 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108  
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
```

```
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150
```

Pour accéder à un attribut spécifique, on aura recours à `attr()` en spécifiant à la fois l'objet considéré et le nom de l'attribut souhaité.

```
iris |> attr("names")
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Pour les attributs les plus courants de **R**, il faut noter qu'il existe le plus souvent des fonctions spécifiques, comme `class()`, `names()` ou `row.names()`.

```
class(iris)
```

```
[1] "data.frame"
```

```
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

La fonction `attr()`, associée à l'opérateur d'assignation (`<-`) permet également de définir ses propres attributs.

```
attr(iris, "perso") <- "Des notes personnelles"
attributes(iris)
```

```
$names
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
$class
```

```
[1] "data.frame"
```

```
$row.names
```

```

[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
[55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
[73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
[91] 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150

```

```
$perso
```

```
[1] "Des notes personnelles"
```

```
attr(iris, "perso")
```

```
[1] "Des notes personnelles"
```

partie II

Manipulation de données

7 Le pipe

Il est fréquent d'enchaîner des opérations en appelant successivement des fonctions sur le résultat de l'appel précédent.

Prenons un exemple. Supposons que nous ayons un vecteur numérique `v` dont nous voulons calculer la moyenne puis l'afficher via un message dans la console. Pour un meilleur rendu, nous allons arrondir la moyenne à une décimale, mettre en forme le résultat à la française, c'est-à-dire avec la virgule comme séparateur des décimales, créer une phrase avec le résultat, puis l'afficher dans la console. Voici le code correspondant, étape par étape.

```
v <- c(1.2, 8.7, 5.6, 11.4)
m <- mean(v)
r <- round(m, digits = 1)
f <- format(r, decimal.mark = ",")
p <- paste0("La moyenne est de ", f, ".")
message(p)
```

La moyenne est de 6,7.

Cette écriture, n'est pas vraiment optimale, car cela entraîne la création d'un grand nombre de variables intermédiaires totalement inutiles. Nous pourrions dès lors imbriquer les différentes fonctions les unes dans les autres :

```
message(paste0("La moyenne est de ", format(round(mean(v), digits = 1), decimal.mark
```

La moyenne est de 6,7.

Nous obtenons bien le même résultat, mais la lecture de cette ligne de code est assez difficile et il n'est pas aisé de bien identifier à quelle fonction est rattaché chaque argument.

Une amélioration possible serait d'effectuer des retours à la ligne avec une indentation adéquate pour rendre cela plus lisible.

```
message(  
  paste0(  
    "La moyenne est de ",  
    format(  
      round(  
        mean(v),  
        digits = 1),  
        decimal.mark = ",",  
      ),  
    ". "  
  )  
)
```

La moyenne est de 6,7.

C'est déjà mieux, mais toujours pas optimal.

7.1 Le pipe natif de R : `|>`

Depuis la version 4.1, **R** a introduit ce que l'on nomme un *pipe* (tuyau en anglais), un nouvel opérateur noté `|>`.

Le principe de cet opérateur est de passer l'élément situé à sa gauche comme premier argument de la fonction située à sa droite. Ainsi, l'écriture `x |> f()` est équivalente à `f(x)` et l'écriture `x |> f(y)` à `f(x, y)`.

Parfois, on souhaite passer l'objet `x` à un autre endroit de la fonction `f()` que le premier argument. Depuis la version 4.2, **R** a introduit l'opérateur `_`, que l'on nomme un *placeholder*, pour indiquer où passer l'objet de gauche. Ainsi, `x |> f(y, a = _)` devient équivalent à `f(y, a = x)`. **ATTENTION** : le

placeholder doit impérativement être transmis à un argument nommé !

Tout cela semble encore un peu abstrait ? Reprenons notre exemple précédent et réécrivons le code avec le *pipe*.

```
v |>
  mean() |>
  round(digits = 1) |>
  format(decimal.mark = ",") |>
  paste0("La moyenne est de ", m = _, ".") |>
  message()
```

La moyenne est de 6,7.

Le code n'est-il pas plus lisible ?

7.2 Le pipe du tidyverse : %>%

Ce n'est qu'à partir de la version 4.1 sortie en 2021 que **R** a proposé de manière native un *pipe*, en l'occurrence l'opérateur `|>`.

En cela, **R** s'est notamment inspiré d'un opérateur similaire introduit dès 2014 dans le *tidyverse*. Le pipe du *tidyverse* fonctionne de manière similaire. Il est implémenté dans le package `{magrittr}` qui doit donc être chargé en mémoire. Le *pipe* est également disponible lorsque l'on effectue `library(tidyverse)`.

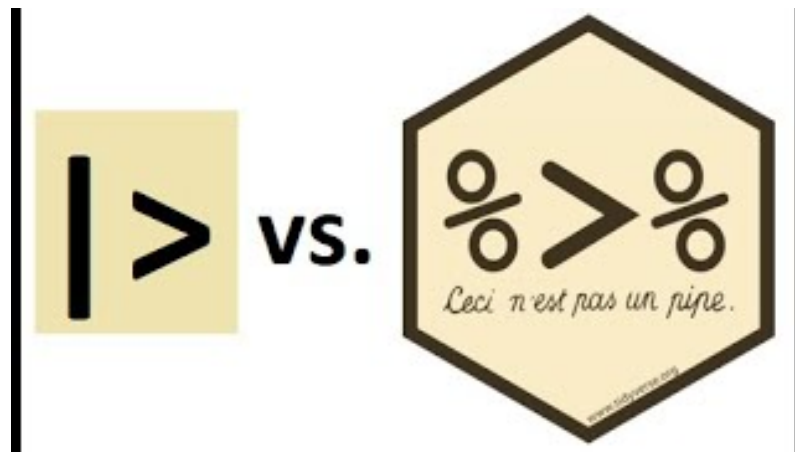
Cet opérateur s'écrit `%>%` et il dispose lui aussi d'un *placeholder* qui est le `..`. La syntaxe du *placeholder* est un peu plus souple puisqu'il peut être passé à tout type d'argument, y compris un argument sans nom. Si l'on reprend notre exemple précédent.

```
library(magrittr)
v %>%
  mean() %>%
  round(digits = 1) %>%
  format(decimal.mark = ",") %>%
```

```
paste0("La moyenne est de ", ., ".") %>%  
message()
```

La moyenne est de 6,7.

7.3 Vaut-il mieux utiliser `|>` ou `%>%` ?



Bonne question. Si vous utilisez une version récente de **R** (4.2), il est préférable d'avoir recours au *pipe* natif de **R** dans la mesure où il est [plus efficient en termes de temps de calcul](#) car il fait partie intégrante du langage. Dans ce guide, nous privilégeons d'ailleurs l'utilisation de `|>`.

Si votre code nécessite de fonctionner avec différentes versions de **R**, par exemple dans le cadre d'un package, il est alors préférable, pour le moment, d'utiliser celui fourni par `{magrittr}` (`%>%`).

7.4 Accéder à un élément avec `purrr::pluck()` et `purrr::chuck()`

Il est fréquent d'avoir besoin d'accéder à un élément précis d'une liste, d'un tableau ou d'un vecteur, ce que l'on fait d'ordinaire avec la syntaxe `[[]]` ou `$` pour les listes ou `[]` pour

les vecteurs. Cependant, cette syntaxe se combine souvent mal avec un enchaînement d'opérations utilisant le *pipe*.

Le package `{purrr}`, chargé par défaut avec `library(tidyverse)`, fournit une fonction `purrr::pluck()` qui, est l'équivalent de `[[]]`, et qui permet de récupérer un élément par son nom ou sa position. Ainsi, si l'on considère le tableau de données `iris`, `pluck(iris, "Petal.Width")` est équivalent à `iris$Petal.Width`. Voyons un exemple d'utilisation dans le cadre d'un enchaînement d'opérations.

```
iris |>
  purrr::pluck("Petal.Width") |>
  mean()
```

```
[1] 1.199333
```

Cette écriture est équivalente à :

```
mean(iris$Petal.Width)
```

```
[1] 1.199333
```

`purrr::pluck()` fonctionne également sur des vecteurs (et dans ce cas opère comme `[[]]`).

```
v <- c("a", "b", "c", "d")
v |> purrr::pluck(2)
```

```
[1] "b"
```

```
v[2]
```

```
[1] "b"
```

On peut également, dans un même appel à `purrr::pluck()`, enchaîner plusieurs niveaux. Les trois syntaxes ci-après sont ainsi équivalents :

```
iris |>
  purrr::pluck("Sepal.Width", 3)
```

```
[1] 3.2
```

```
iris |>
  purrr::pluck("Sepal.Width") |>
  purrr::pluck(3)
```

```
[1] 3.2
```

```
iris[["Sepal.Width"]][3]
```

```
[1] 3.2
```

Si l'on demande un élément qui n'existe pas, `purrr::pluck()` renverra l'élément vide (NULL). Si l'on souhaite plutôt que cela génère une erreur, on aura alors recours à `purrr::chuck()`.

```
iris |> purrr::pluck("inconnu")
```

```
NULL
```

```
iris |> purrr::chuck("inconnu")
```

```
Error: Can't find name `inconnu` in vector
```

```
v |> purrr::pluck(10)
```

```
NULL
```

```
v |> purrr::chuck(10)
```

```
Error: Index 1 exceeds the length of plucked object (10 > 4)
```

8 dplyr

`{dplyr}` est l'un des packages les plus connus du *tidyverse*. Il facilite le traitement et la manipulation des tableaux de données (qu'il s'agisse de *data frame* ou de *tibble*). Il propose une syntaxe claire et cohérente, sous formes de verbes correspondant à des fonctions.

`{dplyr}` part du principe que les données sont *tidy* (chaque variable est une colonne, chaque observation est une ligne, voir Chapitre 5). Les verbes de `{dplyr}` prennent en entrée un tableau de données¹⁰ (*data frame* ou *tibble*) et renvoient systématiquement un *tibble*.

```
library(dplyr)
```

Dans ce qui suit on va utiliser le jeu de données `{nycflights13}`, contenu dans l'extension du même nom (qu'il faut donc avoir installée). Celui-ci correspond aux données de tous les vols au départ d'un des trois aéroports de New-York en 2013. Il a la particularité d'être réparti en trois tables :

- `nycflights13::flights` contient des informations sur les vols : date, départ, destination, horaires, retard...
- `nycflights13::airports` contient des informations sur les aéroports
- `nycflights13::airlines` contient des données sur les compagnies aériennes

On va charger les trois tables du jeu de données :

```
library(nycflights13)
## Chargement des trois tables du jeu de données
data(flights)
data(airports)
```

¹⁰ Le package `{dbplyr}` permet d'étendre les verbes de `{dplyr}` à des tables de bases de données **SQL**, `{dtplyr}` à des tableaux de données du type `{data.table}` et `{srvyr}` à des données pondérées du type `{survey}`.

```
data(airlines)
```

Normalement trois objets correspondant aux trois tables ont dû apparaître dans votre environnement.

8.1 Opérations sur les lignes

8.1.1 filter()

`dplyr::filter()` sélectionne des lignes d'un tableau de données selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoie `TRUE` (vrai) sont conservées¹¹.

Par exemple, si on veut sélectionner les vols du mois de janvier, on peut filtrer sur la variable *month* de la manière suivante :

```
filter(flights, month == 1)
```

¹¹ Si le test renvoie faux (`FALSE`) ou une valeur manquante (`NA`), les lignes correspondantes ne seront donc pas sélectionnées.

```
# A tibble: 27,004 x 19
  year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>      <int>    <dbl>   <int>   <int>    <dbl> <chr>
1  2013     1     1     517         515         2     830     819        11 UA
2  2013     1     1     533         529         4     850     830        20 UA
3  2013     1     1     542         540         2     923     850        33 AA
4  2013     1     1     544         545        -1    1004    1022       -18 B6
5  2013     1     1     554         600        -6     812     837       -25 DL
6  2013     1     1     554         558        -4     740     728        12 UA
7  2013     1     1     555         600        -5     913     854        19 B6
8  2013     1     1     557         600        -3     709     723       -14 EV
9  2013     1     1     557         600        -3     838     846        -8 B6
10 2013     1     1     558         600        -2     753     745         8 AA
# ... with 26,994 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

Cela peut s'écrire plus simplement avec un pipe :

```
flights |> filter(month == 1)
```

```
# A tibble: 27,004 x 19
```

```
  year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>       <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     1     517         515     2     830     819     11  UA
2  2013     1     1     533         529     4     850     830     20  UA
3  2013     1     1     542         540     2     923     850     33  AA
4  2013     1     1     544         545    -1    1004    1022    -18  B6
5  2013     1     1     554         600    -6     812     837    -25  DL
6  2013     1     1     554         558    -4     740     728     12  UA
7  2013     1     1     555         600    -5     913     854     19  B6
8  2013     1     1     557         600    -3     709     723    -14  EV
9  2013     1     1     557         600    -3     838     846     -8  B6
10 2013     1     1     558         600    -2     753     745      8  AA
# ... with 26,994 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

Si l'on veut uniquement les vols avec un retard au départ
(variable *dep_delay*) compris entre 10 et 15 minutes :

```
flights |>
  filter(dep_delay >= 10 & dep_delay <= 15)
```

```
# A tibble: 14,919 x 19
```

```
  year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>       <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     1     611         600    11     945     931     14  UA
2  2013     1     1     623         610    13     920     915      5  AA
3  2013     1     1     743         730    13    1107    1100      7  AA
4  2013     1     1     743         730    13    1059    1056      3  DL
5  2013     1     1     851         840    11    1215    1206      9  UA
6  2013     1     1     912         900    12    1241    1220     21  AA
7  2013     1     1     914         900    14    1058    1043     15  UA
```

```

 8 2013      1      1      920      905      15      1039      1025      14 B6
 9 2013      1      1     1011     1001      10      1133      1128       5 EV
10 2013      1      1     1112     1100      12      1440      1438       2 UA
# ... with 14,909 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay

```

Si l'on passe plusieurs arguments à `dplyr::filter()`, celui-ci rajoute automatiquement une condition **ET**. La ligne ci-dessus peut donc également être écrite de la manière suivante, avec le même résultat :

```

flights |>
  filter(dep_delay >= 10, dep_delay <= 15)

```

Enfin, on peut également placer des fonctions dans les tests, qui nous permettent par exemple de sélectionner les vols avec la plus grande distance :

```

flights |>
  filter(distance == max(distance))

```

```

# A tibble: 342 x 19
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     1     857        900     -3    1516    1530    -14 HA
2  2013     1     2     909        900      9    1525    1530     -5 HA
3  2013     1     3     914        900     14    1504    1530    -26 HA
4  2013     1     4     900        900      0    1516    1530    -14 HA
5  2013     1     5     858        900     -2    1519    1530    -11 HA
6  2013     1     6    1019        900     79    1558    1530     28 HA
7  2013     1     7    1042        900    102    1620    1530     50 HA
8  2013     1     8     901        900      1    1504    1530    -26 HA
9  2013     1     9     641        900   1301    1242    1530   1272 HA
10 2013     1    10     859        900     -1    1449    1530    -41 HA
# ... with 332 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names

```

```
# 1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
# 5: arr_delay
```

💡 Évaluation contextuelle

Il est important de noter que `{dplyr}` procède à une évaluation contextuelle des expressions qui lui sont passées. Ainsi, on peut indiquer directement le nom d'une variable et `{dplyr}` l'interprétera dans le contexte du tableau de données, c'est-à-dire regardera s'il existe une colonne portant ce nom dans le tableau.

Dans l'expression `flights |> filter(month == 1)`, `month` est interprété comme la colonne `month` du tableau `flights`, à savoir `flights$month`.

Il est également possible d'indiquer des objets extérieurs au tableau :

```
m <- 2
flights |>
  filter(month == m)
```

```
# A tibble: 24,951 x 19
```

	year	month	day	dep_time	sched_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	2	1	456	500	-4	652	648	4	US
2	2013	2	1	520	525	-5	816	820	-4	UA
3	2013	2	1	527	530	-3	837	829	8	UA
4	2013	2	1	532	540	-8	1007	1017	-10	B6
5	2013	2	1	540	540	0	859	850	9	AA
6	2013	2	1	552	600	-8	714	715	-1	EV
7	2013	2	1	552	600	-8	919	910	9	AA
8	2013	2	1	552	600	-8	655	709	-14	B6
9	2013	2	1	553	600	-7	833	815	18	FL
10	2013	2	1	553	600	-7	821	825	-4	MQ

```
# ... with 24,941 more rows, 9 more variables: flight <int>, tailnum <chr>,
# origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
# minute <dbl>, time_hour <dtm>, and abbreviated variable names
# 1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
# 5: arr_delay
```

Cela fonctionne car il n'y a pas de colonne `m` dans

`flights`. Dès lors, `{dplyr}` regarde s'il existe un objet `m` dans l'environnement de travail.

Par contre, si une colonne existe dans le tableau, elle aura priorité sur les objets du même nom dans l'environnement. Dans l'exemple ci-dessous, le résultat obtenu n'est pas celui voulu. Il est interprété comme sélectionner toutes les lignes où la colonne *mois* est égale à elle-même et donc cela sélectionne toutes les lignes du tableau.

```
month <- 3
flights |>
  filter(month == month)
```

A tibble: 336,776 x 19

	year	month	day	dep_time	sched_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	1	1	517	515	2	830	819	11	UA
2	2013	1	1	533	529	4	850	830	20	UA
3	2013	1	1	542	540	2	923	850	33	AA
4	2013	1	1	544	545	-1	1004	1022	-18	B6
5	2013	1	1	554	600	-6	812	837	-25	DL
6	2013	1	1	554	558	-4	740	728	12	UA
7	2013	1	1	555	600	-5	913	854	19	B6
8	2013	1	1	557	600	-3	709	723	-14	EV
9	2013	1	1	557	600	-3	838	846	-8	B6
10	2013	1	1	558	600	-2	753	745	8	AA

... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
minute <dbl>, time_hour <dtm>, and abbreviated variable names
1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
5: arr_delay

Afin de distinguer ce qui correspond à une colonne du tableau et à un objet de l'environnement, on pourra avoir recours à `.data` et `.env` (voir `help(".env", package = "rlang")`).

```
month <- 3
flights |>
  filter(.data$month == .env$month)
```



```
# A tibble: 28,834 x 19
  year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     3     1       4      2159    125    318     56    142 B6
2  2013     3     1      50      2358     52    526    438     48 B6
3  2013     3     1     117      2245    152    223   2354    149 B6
4  2013     3     1    454       500     -6    633    648    -15 US
5  2013     3     1    505       515    -10    746    810    -24 UA
6  2013     3     1    521       530     -9    813    827    -14 UA
7  2013     3     1    537       540     -3    856    850     6 AA
8  2013     3     1    541       545     -4   1014   1023    -9 B6
9  2013     3     1    549       600    -11    639    703   -24 US
10 2013     3     1    550       600    -10    747    801   -14 EV
# ... with 28,824 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

8.1.2 slice()

Le verbe `dplyr::slice()` sélectionne des lignes du tableau selon leur position. On lui passe un chiffre ou un vecteur de chiffres.

Si l'on souhaite sélectionner la 345^e ligne du tableau `airports` :

```
airports |>
  slice(345)
```

```
# A tibble: 1 x 8
  faa   name          lat   lon   alt   tz dst  tzone
  <chr> <chr>          <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 CYF   Chefnak Airport  60.1 -164.   40   -9 A   America/Anchorage
```

Si l'on veut sélectionner les 5 premières lignes :

```
airports |>
  slice(1:5)
```

```
# A tibble: 5 x 8
```

	faa	name	lat	lon	alt	tz	dst	tzone
	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
1	04G	Lansdowne Airport	41.1	-80.6	1044	-5	A	America/New~
2	06A	Moton Field Municipal Airport	32.5	-85.7	264	-6	A	America/Chi~
3	06C	Schaumburg Regional	42.0	-88.1	801	-6	A	America/Chi~
4	06N	Randall Airport	41.4	-74.4	523	-5	A	America/New~
5	09J	Jekyll Island Airport	31.1	-81.4	11	-5	A	America/New~

8.1.3 arrange()

`dplyr::arrange()` réordonne les lignes d'un tableau selon une ou plusieurs colonnes.

Ainsi, si l'on veut trier le tableau `flights` selon le retard au départ, dans l'ordre croissant :

```
flights |>
  arrange(dep_delay)
```

```
# A tibble: 336,776 x 19
```

	year	month	day	dep_time	sched_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	12	7	2040	2123	-43	40	2352	48	B6
2	2013	2	3	2022	2055	-33	2240	2338	-58	DL
3	2013	11	10	1408	1440	-32	1549	1559	-10	EV
4	2013	1	11	1900	1930	-30	2233	2243	-10	DL
5	2013	1	29	1703	1730	-27	1947	1957	-10	F9
6	2013	8	9	729	755	-26	1002	955	7	MQ
7	2013	10	23	1907	1932	-25	2143	2143	0	EV
8	2013	3	30	2030	2055	-25	2213	2250	-37	MQ
9	2013	3	2	1431	1455	-24	1601	1631	-30	9E
10	2013	5	5	934	958	-24	1225	1309	-44	B6

```
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
```

```
# 1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
# 5: arr_delay
```

On peut trier selon plusieurs colonnes. Par exemple selon le mois, puis selon le retard au départ :

```
flights |>
  arrange(month, dep_delay)
```

```
# A tibble: 336,776 x 19
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1    11    1900      1930    -30    2233    2243    -10 DL
2  2013     1    29    1703      1730    -27    1947    1957    -10 F9
3  2013     1    12    1354      1416    -22    1606    1650    -44 FL
4  2013     1    21    2137      2159    -22    2232    2316    -44 DL
5  2013     1    20     704       725    -21    1025    1035    -10 AS
6  2013     1    12    2050      2110    -20    2310    2355    -45 B6
7  2013     1    12    2134      2154    -20         4      50    -46 B6
8  2013     1    14    2050      2110    -20    2329    2355    -26 B6
9  2013     1     4    2140      2159    -19    2241    2316    -35 DL
10 2013     1    11    1947      2005    -18    2209    2230    -21 9E
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

Si l'on veut trier selon une colonne par ordre décroissant, on lui applique la fonction `dplyr::desc()` :

```
flights |>
  arrange(desc(dep_delay))
```

```
# A tibble: 336,776 x 19
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     9     641       900   1301    1242    1530    1272 HA
2  2013     6    15    1432      1935   1137    1607    2120    1127 MQ
```

```

3 2013      1    10    1121      1635    1126    1239    1810    1109 MQ
4 2013      9    20    1139      1845    1014    1457    2210    1007 AA
5 2013      7    22     845      1600    1005    1044    1815     989 MQ
6 2013      4    10    1100      1900     960    1342    2211     931 DL
7 2013      3    17    2321       810     911     135    1020     915 DL
8 2013      6    27     959      1900     899    1236    2226     850 DL
9 2013      7    22    2257       759     898     121    1026     895 DL
10 2013     12     5     756      1700     896    1058    2020     878 AA
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay

```

Combiné avec `dplyr::slice()`, `dplyr::arrange()` permet par exemple de sélectionner les trois vols ayant eu le plus de retard :

```

flights |>
  arrange(desc(dep_delay)) |>
  slice(1:3)

# A tibble: 3 x 19
   year month   day dep_time sched_dep~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>       <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     9     641         900    1301    1242    1530    1272 HA
2  2013     6    15    1432        1935    1137    1607    2120    1127 MQ
3  2013     1    10    1121        1635    1126    1239    1810    1109 MQ
# ... with 9 more variables: flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dtm>, and abbreviated variable names 1: sched_dep_time,
#   2: dep_delay, 3: arr_time, 4: sched_arr_time, 5: arr_delay

```

8.1.4 slice_sample()

`dplyr::slice_sample()` permet de sélectionner aléatoirement un nombre de lignes ou une fraction des lignes d'un tableau. Ainsi si l'on veut choisir 5 lignes au hasard dans le tableau `airports` :

```
airports |>
  slice_sample(n = 5)
```

```
# A tibble: 5 x 8
  faa   name                lat   lon   alt   tz dst  tzone
  <chr> <chr>                <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 CDB   Cold Bay              55.2 -163.    96   -9 A    America/Anchor~
2 FYV   Drake Fld            36.0 -94.2  1251  -6 A    America/Chicago
3 4I7   Putnam County Airport 39.6 -86.8   842  -5 U    America/New_Yo~
4 VCT   Victoria Regional Airport 28.9 -96.9   115  -6 A    America/Chicago
5 CCO   Coweta County Airport 33.3 -84.8   970  -5 A    America/New_Yo~
```

Si l'on veut tirer au hasard 10% des lignes de flights :

```
flights |>
  slice_sample(prop = .1)
```

```
# A tibble: 33,677 x 19
  year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>   <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013    11     6    1435    1440     -5    1642    1706    -24 B6
2  2013     1    11    1446    1450     -4    1622    1640    -18 MQ
3  2013     5     5     819     822     -3    1102    1149    -47 UA
4  2013     7     7    1539    1545     -6    1849    1900    -11 AA
5  2013     7     7     948     955     -7    1058    1115    -17 MQ
6  2013     3    12     644     649     -5     855     920    -25 UA
7  2013     6    27    1754    1800     -6    2004    1913     51 US
8  2013    11    30    1154     936    138    1254    1052    122 B6
9  2013     2    14     633     635     -2     933     950    -17 UA
10 2013     2    11     750     752     -2    1112    1058     14 UA
# ... with 33,667 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

Ces fonctions sont utiles notamment pour faire de l’“échantillonnage” en tirant au hasard un certain nombre d’observations du tableau.

8.1.5 distinct()

`dplyr::distinct()` filtre les lignes du tableau pour ne conserver que les lignes distinctes, en supprimant toutes les lignes en double.

```
flights |>
  select(day, month) |>
  distinct()
```

```
# A tibble: 365 x 2
   day month
  <int> <int>
1     1     1
2     2     1
3     3     1
4     4     1
5     5     1
6     6     1
7     7     1
8     8     1
9     9     1
10    10     1
# ... with 355 more rows
```

On peut lui spécifier une liste de variables : dans ce cas, pour toutes les observations ayant des valeurs identiques pour les variables en question, `dplyr::distinct()` ne conservera que la première d'entre elles.

```
flights |>
  distinct(month, day)
```

```
# A tibble: 365 x 2
  month   day
  <int> <int>
1     1     1
2     1     2
3     1     3
```

```

4      1      4
5      1      5
6      1      6
7      1      7
8      1      8
9      1      9
10     1     10
# ... with 355 more rows

```

L'option `.keep_all` permet, dans l'opération précédente, de conserver l'ensemble des colonnes du tableau :

```

flights |>
  distinct(month, day, .keep_all = TRUE)

# A tibble: 365 x 19
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     1     517        515     2     830     819     11 UA
2  2013     1     2      42       2359    43     518     442     36 B6
3  2013     1     3      32       2359    33     504     442     22 B6
4  2013     1     4      25       2359    26     505     442     23 B6
5  2013     1     5      14       2359    15     503     445     18 B6
6  2013     1     6      16       2359    17     451     442      9 B6
7  2013     1     7      49       2359    50     531     444     47 B6
8  2013     1     8     454        500    -6     625     648    -23 US
9  2013     1     9       2       2359     3     432     444    -12 B6
10 2013     1    10       3       2359     4     426     437    -11 B6
# ... with 355 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay

```

8.2 Opérations sur les colonnes

8.2.1 select()

`dplyr::select()` permet de sélectionner des colonnes d'un tableau de données. Ainsi, si l'on veut extraire les colonnes `lat` et `lon` du tableau `airports` :

```
airports |>
  select(lat, lon)
```

```
# A tibble: 1,458 x 2
   lat   lon
<dbl> <dbl>
1  41.1 -80.6
2  32.5 -85.7
3  42.0 -88.1
4  41.4 -74.4
5  31.1 -81.4
6  36.4 -82.2
7  41.5 -84.5
8  42.9 -76.8
9  39.8 -76.6
10 48.1 -123.
# ... with 1,448 more rows
```

Si on fait précéder le nom d'un -, la colonne est éliminée plutôt que sélectionnée :

```
airports |>
  select(-lat, -lon)
```

```
# A tibble: 1,458 x 6
   faa   name                alt    tz dst  tzone
<chr> <chr>                <dbl> <dbl> <chr> <chr>
1 04G   Lansdowne Airport      1044   -5 A   America/New_York
2 06A   Moton Field Municipal  264    -6 A   America/Chicago
3 06C   Schaumburg Regional    801    -6 A   America/Chicago
4 06N   Randall Airport        523    -5 A   America/New_York
```



```

5 09J   Jekyll Island Airport           11   -5 A   America/New_York
6 0A9   Elizabethton Municipal Airport 1593   -5 A   America/New_York
7 0G6   Williams County Airport         730   -5 A   America/New_York
8 0G7   Finger Lakes Regional Airport   492   -5 A   America/New_York
9 0P2   Shoestring Aviation Airfield    1000   -5 U   America/New_York
10 0S9   Jefferson County Intl           108   -8 A   America/Los_Angeles
# ... with 1,448 more rows

```

`dplyr::select()` comprend toute une série de fonctions facilitant la sélection de multiples colonnes. Par exemple, `dplyr::starts_with()`, `dplyr::ends_with()`, `dplyr::contains()` ou `dplyr::matches()` permettent d'exprimer des conditions sur les noms de variables :

```

flights |>
  select(starts_with("dep_"))

```

```

# A tibble: 336,776 x 2
  dep_time dep_delay
  <int>     <dbl>
1     517         2
2     533         4
3     542         2
4     544        -1
5     554        -6
6     554        -4
7     555        -5
8     557        -3
9     557        -3
10    558        -2
# ... with 336,766 more rows

```

La syntaxe `colonne1:colonne2` permet de sélectionner toutes les colonnes situées entre *colonne1* et *colonne2* incluses¹² :

```

flights |>
  select(year:day)

```

```

# A tibble: 336,776 x 3

```

¹² À noter que cette opération est un peu plus “fragile” que les autres, car si l'ordre des colonnes change elle peut renvoyer un résultat différent.

```

      year month   day
   <int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
5  2013     1     1
6  2013     1     1
7  2013     1     1
8  2013     1     1
9  2013     1     1
10 2013     1     1
# ... with 336,766 more rows

```

`dplyr::all_of()` et `dplyr::any_of()` permettent de fournir une liste de variables à extraire sous forme de vecteur textuel. Alors que `dplyr::all_of()` renverra une erreur si une variable n'est pas trouvée dans le tableau de départ, `dplyr::any_of()` sera moins stricte.

```

flights |>
  select(all_of(c("year", "month", "day")))

```

```

# A tibble: 336,776 x 3
      year month   day
   <int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
5  2013     1     1
6  2013     1     1
7  2013     1     1
8  2013     1     1
9  2013     1     1
10 2013     1     1
# ... with 336,766 more rows

```

```
flights |>
  select(all_of(c("century", "year", "month", "day")))
```

```
Error in `select()`:
! Can't subset columns that don't exist.
x Column `century` doesn't exist.
```

```
Erreur : Can't subset columns that don't exist.
x Column `century` doesn't exist.
```

```
flights |>
  select(any_of(c("century", "year", "month", "day")))
```

```
# A tibble: 336,776 x 3
```

```
   year month   day
<int> <int> <int>
1  2013     1     1
2  2013     1     1
3  2013     1     1
4  2013     1     1
5  2013     1     1
6  2013     1     1
7  2013     1     1
8  2013     1     1
9  2013     1     1
10 2013     1     1
# ... with 336,766 more rows
```

`dplyr::where()` permet de sélectionner des variables à partir d'une fonction qui renvoie une valeur logique. Par exemple, pour sélectionner seulement les variables textuelles :

```
flights |>
  select(where(is.character))
```

```
# A tibble: 336,776 x 4
```

```
  carrier tailnum origin dest
```

```

      <chr>    <chr>    <chr>    <chr>
1 UA        N14228    EWR      IAH
2 UA        N24211    LGA      IAH
3 AA        N619AA    JFK      MIA
4 B6        N804JB    JFK      BQN
5 DL        N668DN    LGA      ATL
6 UA        N39463    EWR      ORD
7 B6        N516JB    EWR      FLL
8 EV        N829AS    LGA      IAD
9 B6        N593JB    JFK      MCO
10 AA       N3ALAA    LGA      ORD
# ... with 336,766 more rows

```

`dplyr::select()` peut être utilisée pour réordonner les colonnes d'une table en utilisant la fonction `dplyr::everything()`, qui sélectionne l'ensemble des colonnes non encore sélectionnées. Ainsi, si l'on souhaite faire passer la colonne *name* en première position de la table `airports`, on peut faire :

```

airports |>
  select(name, everything())

```

```

# A tibble: 1,458 x 8
   name                                faa    lat    lon    alt    tz dst  tzone
   <chr>                             <chr> <dbl> <dbl> <dbl> <dbl> <chr> <chr>
1 Lansdowne Airport                 04G   41.1 -80.6  1044   -5 A    America/~
2 Moton Field Municipal Airport     06A   32.5 -85.7   264   -6 A    America/~
3 Schaumburg Regional               06C   42.0 -88.1   801   -6 A    America/~
4 Randall Airport                   06N   41.4 -74.4   523   -5 A    America/~
5 Jekyll Island Airport              09J   31.1 -81.4    11   -5 A    America/~
6 Elizabethton Municipal Airport     0A9   36.4 -82.2  1593   -5 A    America/~
7 Williams County Airport            0G6   41.5 -84.5   730   -5 A    America/~
8 Finger Lakes Regional Airport      0G7   42.9 -76.8   492   -5 A    America/~
9 Shoestring Aviation Airfield       0P2   39.8 -76.6  1000   -5 U    America/~
10 Jefferson County Intl             0S9   48.1 -123.   108   -8 A    America/~
# ... with 1,448 more rows

```

8.2.2 relocate()

Pour réordonner des colonnes, on pourra aussi avoir recours à `dplyr::relocate()` en indiquant les premières variables. Il n'est pas nécessaire d'ajouter `everything()` car avec `dplyr::relocate()` toutes les variables sont conservées.

```
airports |>
  relocate(lon, lat, name)
```

```
# A tibble: 1,458 x 8
   lon lat name      faa alt tz dst tzone
<dbl> <dbl> <chr> <chr> <dbl> <dbl> <chr> <chr>
1 -80.6 41.1 Lansdowne Airport 04G 1044 -5 A America/~
2 -85.7 32.5 Moton Field Municipal Airport 06A 264 -6 A America/~
3 -88.1 42.0 Schaumburg Regional 06C 801 -6 A America/~
4 -74.4 41.4 Randall Airport 06N 523 -5 A America/~
5 -81.4 31.1 Jekyll Island Airport 09J 11 -5 A America/~
6 -82.2 36.4 Elizabethton Municipal Airport 0A9 1593 -5 A America/~
7 -84.5 41.5 Williams County Airport 0G6 730 -5 A America/~
8 -76.8 42.9 Finger Lakes Regional Airport 0G7 492 -5 A America/~
9 -76.6 39.8 Shoestring Aviation Airfield 0P2 1000 -5 U America/~
10 -123. 48.1 Jefferson County Intl 0S9 108 -8 A America/~
# ... with 1,448 more rows
```

8.2.3 rename()

Une variante de `dplyr::select()` est `dplyr::rename()`¹³, qui permet de renommer facilement des colonnes. On l'utilise en lui passant des paramètres de la forme `nouveau_nom = ancien_nom`. Ainsi, si on veut renommer les colonnes *lon* et *lat* de *airports* en *longitude* et *latitude* :

```
airports |>
  rename(longitude = lon, latitude = lat)
```

```
# A tibble: 1,458 x 8
   faa name latitude longi~1 alt tz dst tzone
```

¹³ Il est également possible de renommer des colonnes directement avec `select()`, avec la même syntaxe que pour `rename()`.

```

      <chr> <chr>                                <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
1 04G    Lansdowne Airport                      41.1   -80.6  1044   -5 A    Amer~
2 06A    Moton Field Municipal Airport          32.5   -85.7   264   -6 A    Amer~
3 06C    Schaumburg Regional                    42.0   -88.1   801   -6 A    Amer~
4 06N    Randall Airport                       41.4   -74.4   523   -5 A    Amer~
5 09J    Jekyll Island Airport                 31.1   -81.4    11   -5 A    Amer~
6 0A9    Elizabethton Municipal Airport         36.4   -82.2  1593   -5 A    Amer~
7 0G6    Williams County Airport               41.5   -84.5   730   -5 A    Amer~
8 0G7    Finger Lakes Regional Airport          42.9   -76.8   492   -5 A    Amer~
9 0P2    Shoestring Aviation Airfield           39.8   -76.6  1000   -5 U    Amer~
10 OS9   Jefferson County Intl                  48.1  -123.   108   -8 A    Amer~
# ... with 1,448 more rows, and abbreviated variable name 1: longitude

```

Si les noms de colonnes comportent des espaces ou des caractères spéciaux, on peut les entourer de guillemets (") ou de quotes inverses (`) :

```

flights |>
  rename(
    "retard départ" = dep_delay,
    "retard arrivée" = arr_delay
  ) |>
  select(`retard départ`, `retard arrivée`)

```

```

# A tibble: 336,776 x 2
  `retard départ` `retard arrivée`
      <dbl>         <dbl>
1         2         11
2         4         20
3         2         33
4        -1        -18
5        -6        -25
6        -4         12
7        -5         19
8        -3        -14
9        -3         -8
10       -2          8
# ... with 336,766 more rows

```

8.2.4 rename_with()

La fonction `dplyr::rename_with()` permet de renommer plusieurs colonnes d'un coup en transmettant une fonction, par exemple `toupper()` qui passe tous les caractères en majuscule.

```
airports |>
  rename_with(toupper)
```

A tibble: 1,458 x 8

	FAA	NAME	LAT	LON	ALT	TZ	DST	TZONE
	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
1	04G	Lansdowne Airport	41.1	-80.6	1044	-5	A	America/~
2	06A	Moton Field Municipal Airport	32.5	-85.7	264	-6	A	America/~
3	06C	Schaumburg Regional	42.0	-88.1	801	-6	A	America/~
4	06N	Randall Airport	41.4	-74.4	523	-5	A	America/~
5	09J	Jekyll Island Airport	31.1	-81.4	11	-5	A	America/~
6	0A9	Elizabethton Municipal Airport	36.4	-82.2	1593	-5	A	America/~
7	0G6	Williams County Airport	41.5	-84.5	730	-5	A	America/~
8	0G7	Finger Lakes Regional Airport	42.9	-76.8	492	-5	A	America/~
9	0P2	Shoestring Aviation Airfield	39.8	-76.6	1000	-5	U	America/~
10	0S9	Jefferson County Intl	48.1	-123.	108	-8	A	America/~

... with 1,448 more rows

On pourra notamment utiliser les fonctions du package `snakecase` et, en particulier, `snakecase::to_snake_case()` que je recommande pour nommer de manière consistante les variables¹⁴.

8.2.5 pull()

La fonction `dplyr::pull()` permet d'accéder au contenu d'une variable. C'est un équivalent aux opérateurs `$` ou `[[]]`. On peut lui passer un nom de variable ou bien sa position.

```
airports |>
  pull(alt) |>
  mean()
```

¹⁴ Le *snake case* est une convention typographique en informatique consistant à écrire des ensembles de mots, généralement, en minuscules en les séparant par des tirets bas.

```
[1] 1001.416
```

i Note

`dplyr::pull()` ressemble à la fonction `purrr::chuck()` que nous avons déjà abordée (cf. Section 7.4). Cependant, `dplyr::pull()` ne fonctionne que sur des tableaux de données tandis que `purrr::chuck()` est plus générique et peut s'appliquer à tous types de listes.

8.2.6 `mutate()`

`dplyr::mutate()` permet de créer de nouvelles colonnes dans le tableau de données, en général à partir de variables existantes.

Par exemple, la table `airports` contient l'altitude de l'aéroport en pieds. Si l'on veut créer une nouvelle variable `alt_m` avec l'altitude en mètres, on peut faire :

```
airports <-  
  airports |>  
  mutate(alt_m = alt / 3.2808)
```

On peut créer plusieurs nouvelles colonnes en une seule fois, et les expressions successives peuvent prendre en compte les résultats des calculs précédents. L'exemple suivant convertit d'abord la distance en kilomètres dans une variable `distance_km`, puis utilise cette nouvelle colonne pour calculer la vitesse en km/h.

```
flights <-  
  flights |>  
  mutate(  
    distance_km = distance / 0.62137,  
    vitesse = distance_km / air_time * 60  
  )
```


8.3 Opérations groupées

8.3.1 group_by()

Un élément très important de `{dplyr}` est la fonction `dplyr::group_by()`. Elle permet de définir des groupes de lignes à partir des valeurs d'une ou plusieurs colonnes. Par exemple, on peut grouper les vols selon leur mois :

```
flights |>
  group_by(month)
```

A tibble: 336,776 x 21

Groups: month [12]

	year	month	day	dep_time	sched_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	1	1	517	515	2	830	819	11	UA
2	2013	1	1	533	529	4	850	830	20	UA
3	2013	1	1	542	540	2	923	850	33	AA
4	2013	1	1	544	545	-1	1004	1022	-18	B6
5	2013	1	1	554	600	-6	812	837	-25	DL
6	2013	1	1	554	558	-4	740	728	12	UA
7	2013	1	1	555	600	-5	913	854	19	B6
8	2013	1	1	557	600	-3	709	723	-14	EV
9	2013	1	1	557	600	-3	838	846	-8	B6
10	2013	1	1	558	600	-2	753	745	8	AA

... with 336,766 more rows, 11 more variables: flight <int>, tailnum <chr>,
origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
minute <dbl>, time_hour <dtm>, distance_km <dbl>, vitesse <dbl>, and
abbreviated variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
4: sched_arr_time, 5: arr_delay

Par défaut ceci ne fait rien de visible, à part l'apparition d'une mention *Groups* dans l'affichage du résultat. Mais à partir du moment où des groupes ont été définis, les verbes comme `dplyr::slice()` ou `dplyr::mutate()` vont en tenir compte lors de leurs opérations.

Par exemple, si on applique `dplyr::slice()` à un tableau préalablement groupé, il va sélectionner les lignes aux positions

indiquées *pour chaque groupe*. Ainsi la commande suivante affiche le premier vol de chaque mois, selon leur ordre d'apparition dans le tableau :

```
flights |>
  group_by(month) |>
  slice(1)
```

```
# A tibble: 12 x 21
```

```
# Groups:   month [12]
```

	year	month	day	dep_time	sched_de~1	dep_d~2	arr_t~3	sched~4	arr_d~5	carrier
	<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>
1	2013	1	1	517	515	2	830	819	11	UA
2	2013	2	1	456	500	-4	652	648	4	US
3	2013	3	1	4	2159	125	318	56	142	B6
4	2013	4	1	454	500	-6	636	640	-4	US
5	2013	5	1	9	1655	434	308	2020	408	VX
6	2013	6	1	2	2359	3	341	350	-9	B6
7	2013	7	1	1	2029	212	236	2359	157	B6
8	2013	8	1	12	2130	162	257	14	163	B6
9	2013	9	1	9	2359	10	343	340	3	B6
10	2013	10	1	447	500	-13	614	648	-34	US
11	2013	11	1	5	2359	6	352	345	7	B6
12	2013	12	1	13	2359	14	446	445	1	B6

```
# ... with 11 more variables: flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dtm>, distance_km <dbl>, vitesse <dbl>, and abbreviated
#   variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
#   4: sched_arr_time, 5: arr_delay
```

Idem pour `dplyr::mutate()` : les opérations appliquées lors du calcul des valeurs des nouvelles colonnes sont appliquée groupe de lignes par groupe de lignes. Dans l'exemple suivant, on ajoute une nouvelle colonne qui contient le retard moyen *du mois correspondant* :

```
flights |>
  group_by(month) |>
  mutate(mean_delay_month = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 336,776 x 22
# Groups:   month [12]
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     1     517        515     2     830     819     11 UA
2  2013     1     1     533        529     4     850     830     20 UA
3  2013     1     1     542        540     2     923     850     33 AA
4  2013     1     1     544        545    -1    1004    1022    -18 B6
5  2013     1     1     554        600    -6     812     837    -25 DL
6  2013     1     1     554        558    -4     740     728     12 UA
7  2013     1     1     555        600    -5     913     854     19 B6
8  2013     1     1     557        600    -3     709     723    -14 EV
9  2013     1     1     557        600    -3     838     846     -8 B6
10 2013     1     1     558        600    -2     753     745      8 AA
# ... with 336,766 more rows, 12 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, distance_km <dbl>, vitesse <dbl>,
#   mean_delay_month <dbl>, and abbreviated variable names 1: sched_dep_time,
#   2: dep_delay, 3: arr_time, 4: sched_arr_time, 5: arr_delay
```

Ceci peut permettre, par exemple, de déterminer si un retard donné est supérieur ou inférieur au retard moyen du mois en cours.

`dplyr::group_by()` peut aussi être utile avec `dplyr::filter()`, par exemple pour sélectionner les vols avec le retard au départ le plus important *pour chaque mois* :

```
flights |>
  group_by(month) |>
  filter(dep_delay == max(dep_delay, na.rm = TRUE))
```

```
# A tibble: 12 x 21
# Groups:   month [12]
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
  <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     9     641        900    1301    1242    1530    1272 HA
2  2013    10    14    2042        900     702    2255    1127     688 DL
3  2013    11     3     603       1645     798     829    1913     796 DL
4  2013    12     5     756       1700     896    1058    2020     878 AA
```

```

5 2013 2 10 2243 830 853 100 1106 834 F9
6 2013 3 17 2321 810 911 135 1020 915 DL
7 2013 4 10 1100 1900 960 1342 2211 931 DL
8 2013 5 3 1133 2055 878 1250 2215 875 MQ
9 2013 6 15 1432 1935 1137 1607 2120 1127 MQ
10 2013 7 22 845 1600 1005 1044 1815 989 MQ
11 2013 8 8 2334 1454 520 120 1710 490 EV
12 2013 9 20 1139 1845 1014 1457 2210 1007 AA
# ... with 11 more variables: flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dtm>, distance_km <dbl>, vitesse <dbl>, and abbreviated
#   variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
#   4: sched_arr_time, 5: arr_delay

```

Attention : la clause `dplyr::roup_by()` marche pour les verbes déjà vus précédemment, *sauf* pour `dplyr::arrange()`, qui par défaut trie la table sans tenir compte des groupes. Pour obtenir un tri par groupe, il faut lui ajouter l'argument `.by_group = TRUE`.

On peut voir la différence en comparant les deux résultats suivants :

```

flights |>
  group_by(month) |>
  arrange(desc(dep_delay))

```

```

# A tibble: 336,776 x 21
# Groups:   month [12]
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     9     641        900   1301   1242   1530   1272 HA
2  2013     6    15    1432       1935   1137   1607   2120   1127 MQ
3  2013     1    10    1121       1635   1126   1239   1810   1109 MQ
4  2013     9    20    1139       1845   1014   1457   2210   1007 AA
5  2013     7    22     845       1600   1005   1044   1815    989 MQ
6  2013     4    10    1100       1900    960   1342   2211    931 DL
7  2013     3    17    2321        810    911    135   1020    915 DL
8  2013     6    27     959       1900    899   1236   2226    850 DL
9  2013     7    22    2257        759    898    121   1026    895 DL

```

```

10 2013    12     5      756      1700      896      1058      2020      878 AA
# ... with 336,766 more rows, 11 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, distance_km <dbl>, vitesse <dbl>, and
#   abbreviated variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
#   4: sched_arr_time, 5: arr_delay

```

```

flights |>
  group_by(month) |>
  arrange(desc(dep_delay), .by_group = TRUE)

```

```

# A tibble: 336,776 x 21
# Groups:   month [12]
   year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>   <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
1  2013     1     9     641       900   1301   1242   1530   1272 HA
2  2013     1    10    1121      1635   1126   1239   1810   1109 MQ
3  2013     1     1     848      1835    853   1001   1950    851 MQ
4  2013     1    13    1809       810    599   2054   1042    612 DL
5  2013     1    16    1622       800    502   1911   1054    497 B6
6  2013     1    23    1551       753    478   1812   1006    486 DL
7  2013     1    10    1525       900    385   1713   1039    394 UA
8  2013     1     1    2343      1724    379    314   1938    456 EV
9  2013     1     2    2131      1512    379   2340   1741    359 UA
10 2013     1     7    2021      1415    366   2332   1724    368 B6
# ... with 336,766 more rows, 11 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dtm>, distance_km <dbl>, vitesse <dbl>, and
#   abbreviated variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
#   4: sched_arr_time, 5: arr_delay

```

8.3.2 summarise()

`dplyr::summarise()` permet d'agréger les lignes du tableau en effectuant une opération résumée sur une ou plusieurs colonnes. Il s'agit de toutes les fonctions qui prennent en entrée un ensemble de valeurs et renvoie une valeur unique, comme la moyenne (`mean()`). Par exemple, si l'on souhaite

connaître les retards moyens au départ et à l'arrivée pour l'ensemble des vols du tableau `flights` :

```
flights |>
  summarise(
    retard_dep = mean(dep_delay, na.rm=TRUE),
    retard_arr = mean(arr_delay, na.rm=TRUE)
  )
```

```
# A tibble: 1 x 2
  retard_dep retard_arr
    <dbl>      <dbl>
1    12.6      6.90
```

Cette fonction est en général utilisée avec `dplyr::group_by()`, puisqu'elle permet de découper d'agréger et de résumer les lignes du tableau groupe par groupe. Si l'on souhaite calculer le délai maximum, le délai minimum et le délai moyen au départ pour chaque mois, on pourra faire :

```
flights |>
  group_by(month) |>
  summarise(
    max_delay = max(dep_delay, na.rm=TRUE),
    min_delay = min(dep_delay, na.rm=TRUE),
    mean_delay = mean(dep_delay, na.rm=TRUE)
  )
```

```
# A tibble: 12 x 4
  month max_delay min_delay mean_delay
  <int>    <dbl>    <dbl>    <dbl>
1     1    1301     -30     10.0
2     2     853     -33     10.8
3     3     911     -25     13.2
4     4     960     -21     13.9
5     5     878     -24     13.0
6     6    1137     -21     20.8
7     7    1005     -22     21.7
8     8     520     -26     12.6
```

9	9	1014	-24	6.72
10	10	702	-25	6.24
11	11	798	-32	5.44
12	12	896	-43	16.6

`dplyr::summarise()` dispose d'une fonction spéciale `dplyr::n()`, qui retourne le nombre de lignes du groupe. Ainsi si l'on veut le nombre de vols par destination, on peut utiliser :

```
flights |>
  group_by(dest) |>
  summarise(n = n())
```

```
# A tibble: 105 x 2
  dest      n
  <chr> <int>
1 ABQ    254
2 ACK    265
3 ALB    439
4 ANC      8
5 ATL  17215
6 AUS   2439
7 AVL    275
8 BDL    443
9 BGR    375
10 BHM    297
# ... with 95 more rows
```

`dplyr::n()` peut aussi être utilisée avec `dplyr::filter()` et `dplyr::mutate()`.

8.3.3 count()

À noter que quand l'on veut compter le nombre de lignes par groupe, on peut utiliser directement la fonction `dplyr::count()`. Ainsi le code suivant est identique au précédent :

```
flights |>
  count(dest)
```

```
# A tibble: 105 x 2
  dest      n
  <chr> <int>
1 ABQ    254
2 ACK    265
3 ALB    439
4 ANC      8
5 ATL  17215
6 AUS   2439
7 AVL    275
8 BDL    443
9 BGR    375
10 BHM    297
# ... with 95 more rows
```

8.3.4 Grouper selon plusieurs variables

On peut grouper selon plusieurs variables à la fois, il suffit de les indiquer dans la clause du `dplyr::group_by()` :

```
flights |>
  group_by(month, dest) |>
  summarise(nb = n()) |>
  arrange(desc(nb))
```

``summarise()`` has grouped output by 'month'. You can override using the ``groups`` argument.

```
# A tibble: 1,113 x 3
# Groups:   month [12]
  month dest      nb
  <int> <chr> <int>
1      8 ORD    1604
2     10 ORD    1604
3      5 ORD    1582
```



```

4      9 ORD      1582
5      7 ORD      1573
6      6 ORD      1547
7      7 ATL      1511
8      8 ATL      1507
9      8 LAX      1505
10     7 LAX      1500
# ... with 1,103 more rows

```

On peut également compter selon plusieurs variables :

```

flights |>
  count(origin, dest) |>
  arrange(desc(n))

# A tibble: 224 x 3
   origin dest      n
   <chr>  <chr> <int>
1 JFK    LAX    11262
2 LGA    ATL    10263
3 LGA    ORD     8857
4 JFK    SFO     8204
5 LGA    CLT     6168
6 EWR    ORD     6100
7 JFK    BOS     5898
8 LGA    MIA     5781
9 JFK    MCO     5464
10 EWR    BOS     5327
# ... with 214 more rows

```

On peut utiliser plusieurs opérations de groupage dans le même *pipeline*. Ainsi, si l'on souhaite déterminer le couple origine/destination ayant le plus grand nombre de vols selon le mois de l'année, on devra procéder en deux étapes :

- d'abord grouper selon mois, origine et destination pour calculer le nombre de vols
- puis grouper uniquement selon le mois pour sélectionner la ligne avec la valeur maximale.

Au final, on obtient le code suivant :

```
flights |>
  group_by(month, origin, dest) |>
  summarise(nb = n()) |>
  group_by(month) |>
  filter(nb == max(nb))
```

`summarise()` has grouped output by 'month', 'origin'. You can override using the `.groups` argument.

```
# A tibble: 12 x 4
# Groups:   month [12]
  month origin dest    nb
  <int> <chr>   <chr> <int>
1     1  JFK    LAX    937
2     2  JFK    LAX    834
3     3  JFK    LAX    960
4     4  JFK    LAX    935
5     5  JFK    LAX    960
6     6  JFK    LAX    928
7     7  JFK    LAX    985
8     8  JFK    LAX    979
9     9  JFK    LAX    925
10    10  JFK    LAX    965
11    11  JFK    LAX    907
12    12  JFK    LAX    947
```

Lorsqu'on effectue un `dplyr::group_by()` suivi d'un `dplyr::summarise()`, le tableau résultat est automatiquement dégroupé *de la dernière variable de regroupement*. Ainsi le tableau généré par le code suivant est groupé par *month* et *origin*¹⁵ :

```
flights |>
  group_by(month, origin, dest) |>
  summarise(nb = n())
```

¹⁵ Comme expliqué dans le message affiché dans la console, cela peut être contrôlé avec l'argument `.groups` de `dplyr::summarise()`, dont les options sont décrites dans l'aide de la fonction.

`summarise()` has grouped output by 'month', 'origin'. You can override using the `.groups` argument.

```
# A tibble: 2,313 x 4
# Groups:   month, origin [36]
  month origin dest      nb
  <int> <chr>  <chr> <int>
1     1   EWR   ALB      64
2     1   EWR   ATL    362
3     1   EWR   AUS     51
4     1   EWR   AVL      2
5     1   EWR   BDL     37
6     1   EWR   BNA    111
7     1   EWR   BOS    430
8     1   EWR   BQN     31
9     1   EWR   BTV    100
10    1   EWR   BUF    119
# ... with 2,303 more rows
```

Cela peut permettre d'enchaîner les opérations groupées. Dans l'exemple suivant, on calcule le pourcentage des trajets pour chaque destination par rapport à tous les trajets du mois :

```
flights |>
  group_by(month, dest) |>
  summarise(nb = n()) |>
  mutate(pourcentage = nb / sum(nb) * 100)
```

`summarise()` has grouped output by 'month'. You can override using the `.groups` argument.

```
# A tibble: 1,113 x 4
# Groups:   month [12]
  month dest      nb pourcentage
  <int> <chr> <int>      <dbl>
1     1  ALB      64      0.237
2     1  ATL    1396      5.17
3     1  AUS     169      0.626
4     1  AVL      2      0.00741
5     1  BDL     37      0.137
6     1  BHM     25      0.0926
7     1  BNA    399      1.48
```

```

8      1 BOS      1245      4.61
9      1 BQN        93      0.344
10     1 BTV       223      0.826
# ... with 1,103 more rows

```

On peut à tout moment dégrouper un tableau à l'aide de `dplyr::ungroup()`. Ce serait par exemple nécessaire, dans l'exemple précédent, si on voulait calculer le pourcentage sur le nombre total de vols plutôt que sur le nombre de vols par mois :

```

flights |>
  group_by(month, dest) |>
  summarise(nb = n()) |>
  ungroup() |>
  mutate(pourcentage = nb / sum(nb) * 100)

```

``summarise()`` has grouped output by 'month'. You can override using the ``.groups`` argument.

```

# A tibble: 1,113 x 4
  month dest      nb pourcentage
  <int> <chr> <int>      <dbl>
1     1 ALB      64     0.0190
2     1 ATL    1396     0.415
3     1 AUS     169     0.0502
4     1 AVL       2     0.000594
5     1 BDL      37     0.0110
6     1 BHM      25     0.00742
7     1 BNA     399     0.118
8     1 BOS    1245     0.370
9     1 BQN      93     0.0276
10    1 BTV     223     0.0662
# ... with 1,103 more rows

```

À noter que `dplyr::count()`, par contre, renvoie un tableau non groupé :

```
flights |>
  count(month, dest)
```

```
# A tibble: 1,113 x 3
  month dest      n
  <int> <chr> <int>
1     1 ALB      64
2     2 ATL    1396
3     3 AUS     169
4     4 AVL       2
5     5 BDL     37
6     6 BHM     25
7     7 BNA    399
8     8 BOS   1245
9     9 BQN     93
10    10 BTV    223
# ... with 1,103 more rows
```

8.4 Cheatsheet



8.5 webin-R

On pourra également se référer au webin-R #04 (*manipuler les données avec dplyr*) sur [YouTube](#).

<https://youtu.be/aFvBhgmawcs>

9 Facteurs et forcats

Dans **R**, les facteurs sont utilisés pour représenter des variables catégorielles, c'est-à-dire des variables qui ont un nombre fixé et limité de valeurs possibles (par exemple une variable *sexe* ou une variable *niveau d'éducation*).

De telles variables sont parfois représentées sous forme textuelle (vecteurs de type `character`). Cependant, cela ne permet pas d'indiquer un ordre spécifique aux modalités, à la différence des facteurs.

i Note

Lorsque l'on importe des données d'enquêtes, il est fréquent que les variables catégorielles sont codées sous la forme d'un code numérique (par exemple 1 pour *femme* et 2 pour *homme*) auquel est associé une *étiquette de valeur*. C'est notamment le fonctionnement usuel de logiciels tels que **SPSS**, **Stata** ou **SAS**. Les étiquettes de valeurs seront abordées dans un prochain chapitre (voir Chapitre 12).

Au moment de l'analyse (tableaux statistiques, graphiques, modèles de régression...), il sera nécessaire de transformer ces vecteurs avec étiquettes en facteurs.

9.1 Création d'un facteur

Le plus simple pour créer un facteur est de partir d'un vecteur textuel et d'utiliser la fonction `factor()`.

```
x <- c("nord", "sud", "sud", "est", "est", "est")
x |>
```

```
factor()
```

```
[1] nord sud  sud  est  est  est  
Levels: est nord sud
```

Par défaut, les niveaux du facteur obtenu correspondent aux valeurs uniques du facteur textuel, triés par ordre alphabétique. Si l'on veut contrôler l'ordre des niveaux, et éventuellement indiquer un niveau absent des données, on utilisera l'argument `levels` de `factor()`.

```
x |>  
  factor(levels = c("nord", "est", "sud", "ouest"))
```

```
[1] nord sud  sud  est  est  est  
Levels: nord est sud ouest
```

Si une valeur observée dans les données n'est pas indiqué dans `levels`, elle sera silencieusement convertie en valeur manquante (NA).

```
x |>  
  factor(levels = c("nord", "sud"))
```

```
[1] nord sud  sud  <NA> <NA> <NA>  
Levels: nord sud
```

Si l'on veut être averti par un warning dans ce genre de situation, on pourra avoir plutôt recours à la fonction `readr::parse_factor()` du package `{readr}`, qui, le cas échéant, renverra un tableau avec les problèmes rencontrés.

```
x |>  
  readr::parse_factor(levels = c("nord", "sud"))
```

```
Warning: 3 parsing failures.
row col      expected actual
  4 -- value in level set    est
  5 -- value in level set    est
  6 -- value in level set    est

[1] nord sud  sud  <NA> <NA> <NA>
attr(,"problems")
# A tibble: 3 x 4
  row   col expected      actual
  <int> <int> <chr>      <chr>
1     4     NA value in level set est
2     5     NA value in level set est
3     6     NA value in level set est
Levels: nord sud
```

Une fois un facteur créé, on peut accéder à la liste de ses étiquettes avec `levels()`.

```
f <- factor(x)
levels(f)
```

```
[1] "est" "nord" "sud"
```

Dans certaines situations (par exemple pour la réalisation d'une régression logistique ordinaire), on peut avoir besoin d'indiquer que les modalités du facteur sont ordonnées hiérarchiquement. Dans ce cas là, on aura simplement recours à `ordered()` pour créer/convertir notre facteur.

```
c("supérieur", "primaire", "secondaire", "primaire", "supérieur") |>
  ordered(levels = c("primaire", "secondaire", "supérieur"))
```

```
[1] supérieur primaire secondaire primaire supérieur
Levels: primaire < secondaire < supérieur
```

Techniquement, les valeurs d'un facteur sont stockés de manière interne à l'aide de nombres entiers, dont la valeur représente la

position de l'étiquette correspondante dans l'attribut `levels`. Ainsi, un facteur à `n` modalités sera toujours codé avec les nombres entiers allant de 1 à `n`.

```
class(f)
```

```
[1] "factor"
```

```
typeof(f)
```

```
[1] "integer"
```

```
as.integer(f)
```

```
[1] 2 3 3 1 1 1
```

```
as.character(f)
```

```
[1] "nord" "sud"  "sud"  "est"  "est"  "est"
```

9.2 Changer l'ordre des modalités

Le package `{forcats}`, chargé par défaut lorsque l'on exécute la commande `library(tidyverse)`, fournit plusieurs fonctions pour manipuler des facteurs. Pour donner des exemples d'utilisation de ces différentes fonctions, nous allons utiliser le jeu de données `hdv2003` du package `{questionr}`.

```
library(tidyverse)
data("hdv2003", package = "questionr")
```

Considérons la variable *qualif* qui indique le niveau de qualification des enquêtés. On peut voir la liste des niveaux de ce facteur, et leur ordre, avec `levels()`, ou en effectuant un tri à plat avec la fonction `questionr::freq()`.

```
hdv2003$qualif |>
  levels()
```

```
[1] "Ouvrier specialise"      "Ouvrier qualifie"
[3] "Technicien"             "Profession intermediaire"
[5] "Cadre"                  "Employe"
[7] "Autre"
```

```
hdv2003$qualif |>
  questionr::freq()
```

	n	%	val%
Ouvrier specialise	203	10.2	12.3
Ouvrier qualifie	292	14.6	17.7
Technicien	86	4.3	5.2
Profession intermediaire	160	8.0	9.7
Cadre	260	13.0	15.7
Employe	594	29.7	35.9
Autre	58	2.9	3.5
NA	347	17.3	NA

Parfois, on a simplement besoin d'inverser l'ordre des facteurs, ce qui peut se faire facilement avec la fonction `forcats::fct_rev()`. Elle renvoie le facteur fourni en entrée en ayant inversé l'ordre des modalités (mais sans modifier l'ordre des valeurs dans le vecteur).

```
hdv2003$qualif |>
  fct_rev() |>
  questionr::freq()
```

	n	%	val%
Autre	58	2.9	3.5
Employe	594	29.7	35.9
Cadre	260	13.0	15.7
Profession intermediaire	160	8.0	9.7
Technicien	86	4.3	5.2

Ouvrier qualifie	292	14.6	17.7
Ouvrier specialise	203	10.2	12.3
NA	347	17.3	NA

Pour plus de contrôle, on utilisera `forcats::fct_relevel()` où l'on indique l'ordre souhaité des modalités. On peut également seulement indiquer les premières modalités, les autres seront ajoutées à la fin sans changer leur ordre.

```
hdv2003$qualif |>
  fct_relevel("Cadre", "Autre", "Technicien", "Employe") |>
  questionr::freq()
```

	n	%	val%
Cadre	260	13.0	15.7
Autre	58	2.9	3.5
Technicien	86	4.3	5.2
Employe	594	29.7	35.9
Ouvrier specialise	203	10.2	12.3
Ouvrier qualifie	292	14.6	17.7
Profession intermediaire	160	8.0	9.7
NA	347	17.3	NA

La fonction `forcats::fct_infreq()` ordonne les modalités de celle la plus fréquente à celle la moins fréquente (nombre d'observations) :

```
hdv2003$qualif |>
  fct_infreq() |>
  questionr::freq()
```

	n	%	val%
Employe	594	29.7	35.9
Ouvrier qualifie	292	14.6	17.7
Cadre	260	13.0	15.7
Ouvrier specialise	203	10.2	12.3
Profession intermediaire	160	8.0	9.7
Technicien	86	4.3	5.2
Autre	58	2.9	3.5
NA	347	17.3	NA

Pour inverser l'ordre, on combinera `forcats::fct_infreq()` avec `forcats::fct_rev()`.

```
hdv2003$qualif |>
  fct_infreq() |>
  fct_rev() |>
  questionr::freq()
```

	n	%	val%
Autre	58	2.9	3.5
Technicien	86	4.3	5.2
Profession intermediaire	160	8.0	9.7
Ouvrier specialise	203	10.2	12.3
Cadre	260	13.0	15.7
Ouvrier qualifie	292	14.6	17.7
Employe	594	29.7	35.9
NA	347	17.3	NA

Dans certains cas, on souhaite créer un facteur dont les modalités sont triées selon leur ordre d'apparation dans le jeu de données. Pour cela, on aura recours à `forcats::fct_inorder()`.

```
v <- c("c", "a", "d", "b", "a", "c")
factor(v)
```

```
[1] c a d b a c
Levels: a b c d
```

```
fct_inorder(v)
```

```
[1] c a d b a c
Levels: c a d b
```

La fonction `forcats::fct_reorder()` permet de trier les modalités en fonction d'une autre variable. Par exemple, si je souhaite trier les modalités de la variable *qualif* en fonction de l'âge moyen (dans chaque modalité) :

```

hdv2003$qualif_tri_age <-
  hdv2003$qualif |>
  fct_reorder(hdv2003$age, .fun = mean)
hdv2003 |>
  dplyr::group_by(qualif_tri_age) |>
  dplyr::summarise(age_moyen = mean(age))

```

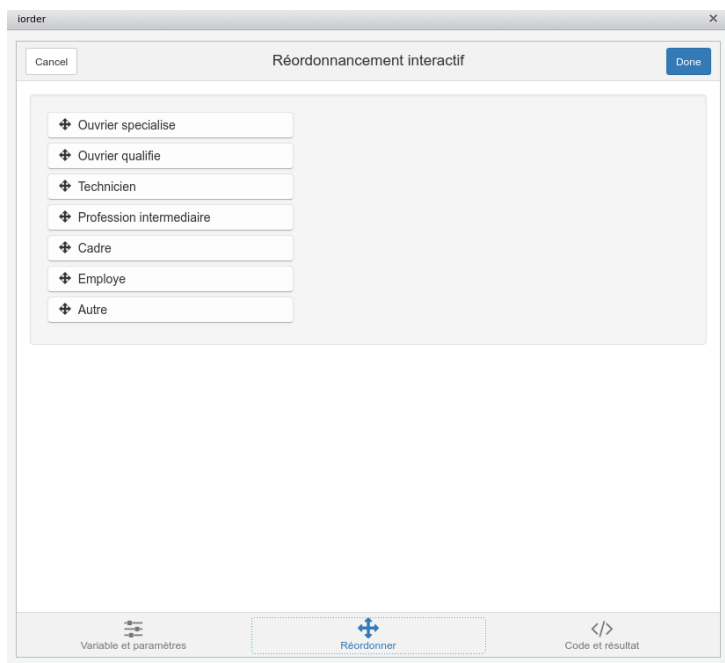
```

# A tibble: 8 x 2
  qualif_tri_age      age_moyen
  <fct>             <dbl>
1 Technicien        45.9
2 Employe           46.7
3 Autre             47.0
4 Ouvrier specialise 48.9
5 Profession intermediaire 49.1
6 Cadre             49.7
7 Ouvrier qualifie  50.0
8 <NA>              47.9

```

Astuce

`{questionr}` propose une interface graphique afin de faciliter les opérations de réordonnancement manuel. Pour la lancer, sélectionner le menu *Addins* puis *Levels ordering*, ou exécuter la fonction `questionr::iorder()` en lui passant comme paramètre le facteur à réordonner.



Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](<https://youtu.be/CokvTbtWdwc?t=3934>).
<https://youtu.be/CokvTbtWdwc>

9.3 Modifier les modalités

Pour modifier le nom des modalités, on pourra avoir recours à `forcats::fct_recode()` avec une syntaxe de la forme "nouveau nom" = "ancien nom".

```
hdv2003$sexe |>
  questionr::freq()
```

	n	%	val%
Homme	899	45	45
Femme	1101	55	55

```

hdv2003$sexe <-
  hdv2003$sexe |>
  fct_recode(f = "Femme", m = "Homme")
hdv2003$sexe |>
  questionr::freq()

```

	n	%	val%
m	899	45	45
f	1101	55	55

On peut également fusionner des modalités ensemble en leur attribuant le même nom.

```

hdv2003$nivetud |>
  questionr::freq()

```

	n	%	val%
N'a jamais fait d'etudes	39	2.0	2.1
A arrete ses etudes, avant la derniere annee d'etudes primaires	86	4.3	4.6
Derniere annee d'etudes primaires	341	17.0	18.1
1er cycle	204	10.2	10.8
2eme cycle	183	9.2	9.7
Enseignement technique ou professionnel court	463	23.2	24.5
Enseignement technique ou professionnel long	131	6.6	6.9
Enseignement superieur y compris technique superieur	441	22.0	23.4
NA	112	5.6	NA

```

hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_recode(
    "primaire" = "N'a jamais fait d'etudes",
    "primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
    "primaire" = "Derniere annee d'etudes primaires",
    "secondaire" = "1er cycle",
    "secondaire" = "2eme cycle",
    "technique/professionnel" = "Enseignement technique ou professionnel court",
    "technique/professionnel" = "Enseignement technique ou professionnel long",
    "superieur" = "Enseignement superieur y compris technique superieur"
  )

```

```
)
hdv2003$instruction |>
  questionr::freq()
```

	n	%	val%
primaire	466	23.3	24.7
secondaire	387	19.4	20.5
technique/professionnel	594	29.7	31.5
supérieur	441	22.0	23.4
NA	112	5.6	NA

Interface graphique

Le package `{questionr}` propose une interface graphique facilitant le recodage des modalités d'une variable qualitative. L'objectif est de permettre à la personne qui l'utilise de saisir les nouvelles valeurs dans un formulaire, et de générer ensuite le code R correspondant au recodage indiqué.

Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Levels recoding*. Sinon, vous pouvez lancer dans la console la fonction `questionr::irec()` en lui passant comme paramètre la variable à recoder.

Cancel Recodage interactif Done

Ouvrier specialise → Ouvrier specialise

Ouvrier qualifie → Ouvrier qualifie

Technicien → Technicien

Profession intermediaire → Profession intermediaire

Cadre → Cadre

Employe → Employe

Autre → Autre

NA → NA

Variable et paramètres Recodage Code et résultat

Astuce

Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](<https://youtu.be/CokvTbtWdwc?t=3387>).
<https://youtu.be/CokvTbtWdwc>

La fonction `forcats::fct_collapse()` est une variante de `forcats::fct_recode()` pour indiquer les fusions de modalités. La même recodification s'écrirait alors :

```
hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_collapse(
    "primaire" = c(
      "N'a jamais fait d'etudes",
      "A arrete ses etudes, avant la derniere annee d'etudes primaires",
      "Derniere annee d'etudes primaires"
```

```

),
"secondaire" = c(
  "1er cycle",
  "2eme cycle"
),
"technique/professionnel" = c(
  "Enseignement technique ou professionnel court",
  "Enseignement technique ou professionnel long"
),
"supérieur" = "Enseignement superieur y compris technique superieur"
)

```

Pour transformer les valeurs manquantes (NA) en une modalité explicite, on pourra avoir recours à `forcats::fct_explicit_na()`.

```

hdv2003$instruction <-
  hdv2003$instruction |>
  fct_explicit_na(na_level = "(manquant)")
hdv2003$instruction |>
  questionr::freq()

```

	n	%	val%
primaire	466	23.3	23.3
secondaire	387	19.4	19.4
technique/professionnel	594	29.7	29.7
supérieur	441	22.0	22.0
(manquant)	112	5.6	5.6

Plusieurs fonctions permettent de regrouper plusieurs modalités dans une modalité *autres*.

Par exemple, avec `forcats::fct_other()`, on pourra indiquer les modalités à garder.

```

hdv2003$qualif |>
  questionr::freq()

```

	n	%	val%
Ouvrier specialise	203	10.2	12.3

Ouvrier qualifie	292	14.6	17.7
Technicien	86	4.3	5.2
Profession intermediaire	160	8.0	9.7
Cadre	260	13.0	15.7
Employe	594	29.7	35.9
Autre	58	2.9	3.5
NA	347	17.3	NA

```
hdv2003$qualif |>
  fct_other(keep = c("Technicien", "Cadre", "Employe")) |>
  questionr::freq()
```

	n	%	val%
Technicien	86	4.3	5.2
Cadre	260	13.0	15.7
Employe	594	29.7	35.9
Other	713	35.6	43.1
NA	347	17.3	NA

La fonction `forcats::fct_lump_n()` permet de ne conserver que les modalités les plus fréquentes et de regrouper les autres dans une modalité *autres*.

```
hdv2003$qualif |>
  fct_lump_n(n = 4, other_level = "Autres") |>
  questionr::freq()
```

	n	%	val%
Ouvrier specialise	203	10.2	12.3
Ouvrier qualifie	292	14.6	17.7
Cadre	260	13.0	15.7
Employe	594	29.7	35.9
Autres	304	15.2	18.4
NA	347	17.3	NA

Et `forcats::fct_lump_min()` celles qui ont un minimum d'observations.

```
hdv2003$qualif |>
  fct_lump_min(min = 200, other_level = "Autres") |>
  questionr::freq()
```

	n	%	val%
Ouvrier specialise	203	10.2	12.3
Ouvrier qualifie	292	14.6	17.7
Cadre	260	13.0	15.7
Employe	594	29.7	35.9
Autres	304	15.2	18.4
NA	347	17.3	NA

Il peut arriver qu'une des modalités d'un facteur ne soit pas représentée dans les données.

```
v <- factor(
  c("a", "a", "b", "a"),
  levels = c("a", "b", "c")
)
questionr::freq(v)
```

	n	%	val%
a	3	75	75
b	1	25	25
c	0	0	0

Pour calculer certains tests statistiques ou faire tourner un modèle, ces modalités sans observation peuvent être problématiques. `forcats::fct_drop()` permet de supprimer les modalités qui n'apparaissent pas dans les données.

```
v
```

```
[1] a a b a
Levels: a b c
```

```
v |> fct_drop()
```

```
[1] a a b a
Levels: a b
```

À l'inverse, `forcats::fct_expand()` permet d'ajouter une ou plusieurs modalités à un facteur.

```
v
```

```
[1] a a b a
Levels: a b c
```

```
v |> fct_expand("d", "e")
```

```
[1] a a b a
Levels: a b c d e
```

9.4 Découper une variable numérique en classes

Il est fréquent d'avoir besoin de découper une variable numérique en une variable catégorielles (un facteur) à plusieurs modalités, par exemple pour créer des groupes d'âges à partir d'une variable *age*.

On utilise pour cela la fonction `cut()` qui prend, outre la variable à découper, un certain nombre d'arguments :

- **breaks** indique soit le nombre de classes souhaité, soit, si on lui fournit un vecteur, les limites des classes ;
- **labels** permet de modifier les noms de modalités attribués aux classes ;
- **include.lowest** et **right** influent sur la manière dont les valeurs situées à la frontière des classes seront incluses ou exclues ;
- **dig.lab** indique le nombre de chiffres après la virgule à conserver dans les noms de modalités.

Prenons tout de suite un exemple et tentons de découper la variable *age* en cinq classes :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(age, 5))
hdv2003$groupe_ages |> questionr::freq()
```

	n	%	val%
(17.9,33.8]	454	22.7	22.7
(33.8,49.6]	628	31.4	31.4
(49.6,65.4]	556	27.8	27.8
(65.4,81.2]	319	16.0	16.0
(81.2,97.1]	43	2.1	2.1

Par défaut **R** nous a bien créé cinq classes d'amplitudes égales. La première classe va de 17,9 à 33,8 ans (en fait de 17 à 32), etc.

Les frontières de classe seraient plus présentables si elles utilisaient des nombres ronds. On va donc spécifier manuellement le découpage souhaité, par tranches de 20 ans :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(age, c(18, 20, 40, 60, 80, 97)))
hdv2003$groupe_ages |> questionr::freq()
```

	n	%	val%
(18,20]	55	2.8	2.8
(20,40]	660	33.0	33.3
(40,60]	780	39.0	39.3
(60,80]	436	21.8	22.0
(80,97]	52	2.6	2.6
NA	17	0.9	NA

Les symboles dans les noms attribués aux classes ont leur importance : (signifie que la frontière de la classe est exclue, tandis que [signifie qu'elle est incluse. Ainsi, (20,40] signifie « strictement supérieur à 20 et inférieur ou égal à 40 ».

On remarque que du coup, dans notre exemple précédent, la valeur minimale, 18, est exclue de notre première classe, et qu'une observation est donc absente de ce découpage. Pour résoudre ce problème on peut soit faire commencer la première classe à 17, soit utiliser l'option `include.lowest=TRUE` :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(
    age,
    c(18, 20, 40, 60, 80, 97),
    include.lowest = TRUE
  ))
hdv2003$groupe_ages |> questionr::freq()
```

	n	%	val%
[18,20]	72	3.6	3.6
(20,40]	660	33.0	33.0
(40,60]	780	39.0	39.0
(60,80]	436	21.8	21.8
(80,97]	52	2.6	2.6

On peut également modifier le sens des intervalles avec l'option `right=FALSE` :

```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(
    age,
    c(18, 20, 40, 60, 80, 97),
    include.lowest = TRUE,
    right = FALSE
  ))
hdv2003$groupe_ages |> questionr::freq()
```

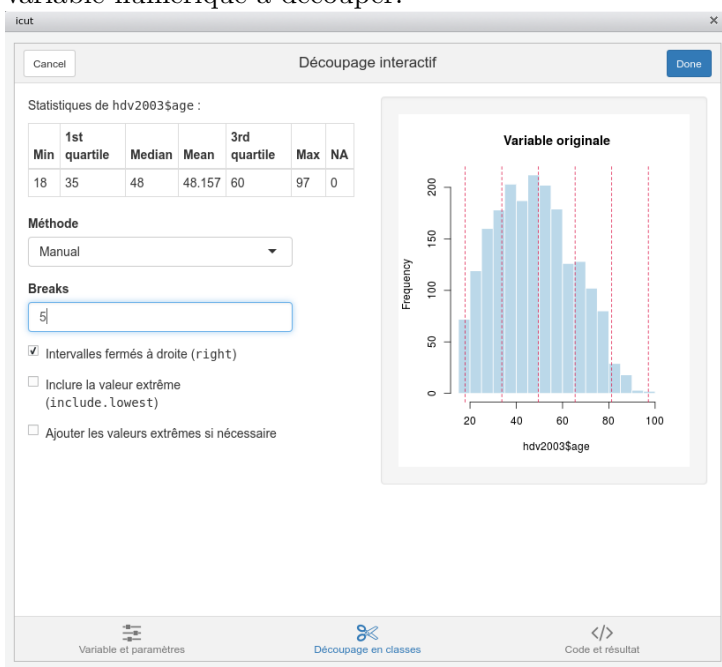
	n	%	val%
[18,20)	48	2.4	2.4
[20,40)	643	32.1	32.1
[40,60)	793	39.6	39.6

```
[60,80) 454 22.7 22.7
[80,97] 62 3.1 3.1
```

💡 Interface graphique

Il n'est pas nécessaire de connaître toutes les options de `cut()`. Le package `{questionr}` propose là encore une interface graphique permettant de visualiser l'effet des différents paramètres et de générer le code **R** correspondant.

Pour utiliser cette interface, sous **RStudio** vous pouvez aller dans le menu *Addins* (présent dans la barre d'outils principale) puis choisir *Numeric range dividing*. Sinon, vous pouvez lancer dans la console la fonction `questionr::icut()` en lui passant comme paramètre la variable numérique à découper.



Une démonstration en vidéo de cet *add-in* est disponible dans le webin-R #05 (*recoder des variables*) sur [YouTube](<https://youtu.be/CokvTbtWdwc?t=2795>).
<https://youtu.be/CokvTbtWdwc>

10 Combiner plusieurs variables

Parfois, on a besoin de créer une nouvelle variable en partant des valeurs d'une ou plusieurs autres variables. Dans ce cas on peut utiliser les fonctions `dplyr::if_else()` pour les cas les plus simples, ou `dplyr::case_when()` pour les cas plus complexes.

Une fois encore, nous utiliser le jeu de données `hdv2003` pour illustrer ces différentes fonctions.

```
library(tidyverse)
data("hdv2003", package = "questionr")
```

10.1 `if_else()`

`dplyr::if_else()` prend trois arguments : un test, les valeurs à renvoyer si le test est vrai, et les valeurs à renvoyer si le test est faux.

Voici un exemple simple :

```
v <- c(12, 14, 8, 16)
if_else(v > 10, "Supérieur à 10", "Inférieur à 10")
```

```
[1] "Supérieur à 10" "Supérieur à 10" "Inférieur à 10" "Supérieur à 10"
```

La fonction devient plus intéressante avec des tests combinant plusieurs variables. Par exemple, imaginons qu'on souhaite créer une nouvelle variable indiquant les hommes de plus de 60 ans :

```

hdv2003 <-
  hdv2003 |>
  mutate(
    statut = if_else(
      sexe == "Homme" & age > 60,
      "Homme de plus de 60 ans",
      "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()

```

	n	% val%
Autre	1778	88.9 88.9
Homme de plus de 60 ans	222	11.1 11.1

Il est possible d'utiliser des variables ou des combinaisons de variables au sein du `dplyr::if_else()`. Supposons une petite enquête menée auprès de femmes et d'hommes. Le questionnaire comportait une question de préférence posée différemment aux femmes et aux hommes et dont les réponses ont ainsi été collectées dans deux variables différentes, *pref_f* et *pref_h*, que l'on souhaite combiner en une seule variable. De même, une certaine mesure quantitative a été réalisée, mais une correction est nécessaire pour normaliser ce score (retirer 0.4 aux scores des hommes et 0.6 aux scores des femmes). Cela peut être réalisé avec le code ci-dessous.

```

df <- tibble(
  sexe = c("f", "f", "h", "h"),
  pref_f = c("a", "b", NA, NA),
  pref_h = c(NA, NA, "c", "d"),
  mesure = c(1.2, 4.1, 3.8, 2.7)
)
df

```

```

# A tibble: 4 x 4
  sexe pref_f pref_h mesure

```

	<chr>	<chr>	<chr>	<dbl>
1	f	a	<NA>	1.2
2	f	b	<NA>	4.1
3	h	<NA>	c	3.8
4	h	<NA>	d	2.7

```
df <-
  df |>
  mutate(
    pref = if_else(sexe == "f", pref_f, pref_h),
    indicateur = if_else(sexe == "h", mesure - 0.4, mesure - 0.6)
  )
df
```

```
# A tibble: 4 x 6
  sexe pref_f pref_h mesure pref indicateur
<chr> <chr> <chr>   <dbl> <chr>   <dbl>
1 f     a     <NA>   1.2 a       0.6
2 f     b     <NA>   4.1 b       3.5
3 h     <NA>   c       3.8 c       3.4
4 h     <NA>   d       2.7 d       2.3
```

! if_else() et ifelse()

La fonction `dplyr::if_else()` ressemble à la fonction `ifelse()` en base **R**. Il y a néanmoins quelques petites différences :

- `dplyr::if_else()` vérifie que les valeurs fournies pour `true` et celles pour `false` sont du même type et de la même classe et renvoie une erreur dans le cas contraire, là où `ifelse()` sera plus permissif ;
- si un vecteur a des attributs (cf. Chapitre 6), ils seront préservés par `dplyr::if_else()` (et pris dans le vecteur `true`), ce que ne fera pas `if_else()` ;
- `dplyr::if_else()` propose un argument optionnel supplémentaire `missing` pour indiquer les valeurs à retourner lorsque le test renvoie `NA`.

10.2 case_when()

`dplyr::case_when()` est une généralisation de `dplyr::if_else()` qui permet d'indiquer plusieurs tests et leurs valeurs associées.

Imaginons que l'on souhaite créer une nouvelle variable permettant d'identifier les hommes de plus de 60 ans, les femmes de plus de 60 ans, et les autres. On peut utiliser la syntaxe suivante :

```
hdv2003 <-  
  hdv2003 |>  
  mutate(  
    statut = case_when(  
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",  
      age >= 60 & sexe == "Femme" ~ "Femme, 60 et plus",  
      TRUE ~ "Autre"  
    )  
  )  
hdv2003 |>  
  pull(statut) |>  
  questionr::freq()
```

	n	% val%
Autre	1484	74.2 74.2
Femme, 60 et plus	278	13.9 13.9
Homme, 60 et plus	238	11.9 11.9

`dplyr::case_when()` prend en arguments une série d'instructions sous la forme `condition ~ valeur`. Il les exécute une par une, et dès qu'une `condition` est vraie, il renvoie la `valeur` associée.

La clause `TRUE ~ "Autre"` permet d'assigner une valeur à toutes les lignes pour lesquelles aucune des conditions précédentes n'est vraie.

! Important

Attention : comme les conditions sont testées l'une après l'autre et que la valeur renvoyée est celle correspondant à la première condition vraie, l'ordre de ces conditions est très important. Il faut absolument aller du plus spécifique au plus général.

Par exemple le recodage suivant ne fonctionne pas :

```
hdv2003 <-  
  hdv2003 |>  
  mutate(  
    statut = case_when(  
      sexe == "Homme" ~ "Homme",  
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",  
      TRUE ~ "Autre"  
    )  
  )  
hdv2003 |>  
  pull(statut) |>  
  questionr::freq()
```

	n	%	val%
Autre	1101	55	55
Homme	899	45	45

Comme la condition `sexe == "Homme"` est plus générale que `sexe == "Homme" & age > 60`, cette deuxième condition n'est jamais testée ! On n'obtiendra jamais la valeur correspondante.

Pour que ce recodage fonctionne il faut donc changer l'ordre des conditions pour aller du plus spécifique au plus général :

```

hdv2003 <-
  hdv2003 |>
  mutate(
    statut = case_when(
      age >= 60 & sexe == "Homme" ~ "Homme, 60 et plus",
      sexe == "Homme" ~ "Homme",
      TRUE ~ "Autre"
    )
  )
hdv2003 |>
  pull(statut) |>
  questionr::freq()

```

	n	% val%
Autre	1101	55.0 55.0
Homme	661	33.1 33.1
Homme, 60 et plus	238	11.9 11.9

C'est pour cela que l'on peut utiliser, en toute dernière condition, la valeur TRUE pour indiquer dans tous les autres cas.

10.3 recode_if()

Parfois, on n'a besoin de ne modifier une variable que pour certaines observations. Prenons un petit exemple :

```

df <- tibble(
  pref = factor(c("bleu", "rouge", "autre", "rouge", "autre")),
  autre_details = c(NA, NA, "bleu ciel", NA, "jaune")
)
df

```

```

# A tibble: 5 x 2
  pref  autre_details
<fct> <chr>
1 bleu  <NA>

```

```
2 rouge <NA>
3 autre bleu ciel
4 rouge <NA>
5 autre jaune
```

Nous avons demandé aux enquêtés d'indiquer leur couleur préférée. Ils pouvaient répondre bleu ou rouge et avait également la possibilité de choisir autre et d'indiquer la valeur de leur choix dans un champs textuel libre.

Une des personnes enquêtées a choisi autre et a indiqué dans le champs texte la valeur bleu ciel. Pour les besoins de l'analyse, on peut considérer que cette valeur bleu ciel pour être tout simplement recodée en bleu.

En syntaxe **R** classique, on pourra simplement faire :

```
df$pref[df$autre_details == "bleu ciel"] <- "bleu"
```

Avec `dplyr::if_else()`, on serait tenté d'écrire :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", "bleu", pref))
```

```
Error in `mutate()` :
! Problem while computing `pref = if_else(autre_details == "bleu ciel",
"bleu", pref)`.
Caused by error in `if_else()` :
! `false` must be a character vector, not a `factor` object.
```

On obtient une erreur, car `dplyr::if_else()` exige les valeurs fournies pour `true` et `false` soient de même type. Essayons alors :

```
df |>
  mutate(pref = if_else(autre_details == "bleu ciel", factor("bleu"), pref))
```

```
Warning in `[<-.factor`(`*tmp*`, i, value = structure(1L, levels = c("autre", :
niveau de facteur incorrect, NAs générés
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 <NA> <NA>
2 <NA> <NA>
3 bleu bleu ciel
4 <NA> <NA>
5 <NA> jaune
```

Ici nous avons un autre problème, signalé par un message d'avertissement (*warning*) : `dplyr::if_else()` ne préserve que les attributs du vecteur passé en `true` et non ceux passés à `false`. Or l'ensemble des modalités (niveaux du facteur) de la variable `pref` n'ont pas été définis dans `factor("bleu")` et sont ainsi perdus, générant une perte de données (valeurs manquantes NA).

Pour obtenir le bon résultat, il faudrait inverser la condition :

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel",
    pref,
    factor("bleu")
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
  <fct> <chr>
1 <NA> <NA>
2 <NA> <NA>
3 bleu bleu ciel
4 <NA> <NA>
5 autre jaune
```

Mais ce n'est toujours pas suffisant. En effet, la variable `autre_details` a des valeurs manquantes pour lesquelles le test `autre_details != "bleu ciel"` renvoie NA ce qui une fois encore génère des valeurs manquantes non souhaitées.

Dès lors, il nous faut soit définir l'argument `missing` de `dplyr::if_else()`, soit être plus précis dans notre test.

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel",
    pref,
    factor("bleu"),
    missing = pref
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
<fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

```
df |>
  mutate(pref = if_else(
    autre_details != "bleu ciel" | is.na(autre_details),
    pref,
    factor("bleu")
  ))
```

```
# A tibble: 5 x 2
  pref  autre_details
<fct> <chr>
1 bleu  <NA>
2 rouge <NA>
3 bleu  bleu ciel
4 rouge <NA>
5 autre jaune
```

Bref, on peut s'en sortir avec `dplyr::if_else()` mais ce n'est pas forcément le plus pratique dans le cas présent. La syntaxe

en base **R** fonctionne très bien, mais ne peut pas être intégrée à un enchainement d'opérations utilisant le *pipe*.

Dans ce genre de situation, on pourra être intéressé par la fonction `labelled::recode_if()` disponible dans le package `{labelled}`. Elle permet de ne modifier que certaines observations d'un vecteur en fonction d'une condition. Si la condition vaut `FALSE` ou `NA`, les observations concernées restent inchangées. Voyons comment cela s'écrit :

```
df <-  
  df |>  
  mutate(  
    pref = pref |>  
      labelled::recode_if(autre_details == "bleu ciel", "bleu")  
  )  
df
```

```
# A tibble: 5 x 2  
  pref autre_details  
  <fct> <chr>  
1 bleu <NA>  
2 rouge <NA>  
3 bleu bleu ciel  
4 rouge <NA>  
5 autre jaune
```

C'est tout de suite plus intuitif !

11 Étiquettes de variables

11.1 Principe

Les étiquettes de variable permettent de donner un nom long, plus explicite, aux différentes colonnes d'un tableau de données (ou encore directement à un vecteur autonome). Dans le champs des grandes enquêtes, il est fréquent de nommer les variables *q101*, *q102*, etc. pour refléter le numéro de la question et d'indiquer ce qu'elle représente (groupe d'âges, milieu de résidence...) avec une étiquette.

Un usage, introduit par le package `{haven}`, et repris depuis par de nombreux autres packages dont `{gtsummary}` que nous aborderons dans de prochains chapitres, consiste à stocker les étiquettes de variables sous la forme d'un attribut¹⁶ `"label"` attaché au vecteur / à la colonne du tableau.

¹⁶ Pour plus d'information sur les attributs, voir Chapitre 6.

Le package `{labelled}` permet de manipuler aisément ces étiquettes de variables.

La visionneuse de données de **RStudio** sait reconnaître et afficher ces étiquettes de variable lorsqu'elles existent. Prenons pour exemple le jeu de données `gtsummary::trial` dont les colonnes ont des étiquettes de variable. La commande `View(gtsummary::trial)` permet d'ouvrir la visionneuse de données de **RStudio**. Comme on peut le constater, une étiquette de variable est bien présente sous le nom des différentes colonnes.

La fonction `labelled::look_for()` du package `{labelled}` permet de lister l'ensemble des variables d'un tableau de données et affiche notamment les étiquettes de variable associées.

	trt	age	marker	stage	grade	response	death	ttdeath	
	Chemotherapy Treatment	Age	Marker Level (ng/mL)	T Stage	Grade	Tumor Response	Patient Died	Months to Death/Censor	
1	Drug A	23	0.160	T1	II		0	0	24.00
2	Drug B	9	1.107	T2	I		1	0	24.00
3	Drug A	31	0.277	T1	II		0	0	24.00
4	Drug A	NA	2.067	T3	III		1	1	17.64
5	Drug A	51	2.767	T4	III		1	1	16.43
6	Drug B	39	0.613	T4	I		0	1	15.64
7	Drug A	37	0.554	T1	II		0	0	24.00
8	Drug A	32	1.739	T1	I		0	1	16.43
9	Drug A	31	0.144	T1	II		0	0	24.00
10	Drug B	34	0.285	T3	I		0	1	10.53
11	Drug B	42	0.513	T1	III		0	0	24.00
12	Drug B	63	0.660	T3	I		1	0	24.00
13	Drug B	54	0.831	T4	III		0	1	14.34
14	Drug B	21	0.258	T4	I		0	1	12.89
15	Drug B	48	0.128	T1	I		0	1	22.68
16	Drug B	71	0.445	T4	III		0	1	8.71
17	Drug A	38	2.083	T4	III		1	0	24.00
18	Drug B	49	0.157	T2	II		0	1	15.21
19	Drug A	57	0.066	T1	III		0	0	24.00
20	Drug A	46	0.325	T1	II		0	0	24.00

Figure 11.1: Présentation du tableau `gtsummary::trial` dans la visionneuse de **RStudio**

```
library(labelled)
gtsummary::trial |>
  look_for()
```

pos	variable	label	col_type	values
1	trt	Chemotherapy Treatment	chr	
2	age	Age	dbl	
3	marker	Marker Level (ng/mL)	dbl	
4	stage	T Stage	fct	T1 T2 T3 T4
5	grade	Grade	fct	I II III
6	response	Tumor Response	int	
7	death	Patient Died	int	
8	ttdeath	Months to Death/Censor	dbl	

La fonction `labelled::look_for()` permet également de rechercher des variables en tenant compte à la fois de leur nom et de leur étiquette.

```
gtsummary::trial |>
  look_for("months")
```

```
pos variable label          col_type values
8   ttdeath Months to Death/Censor dbl
```

Astuce

Comme on le voit, la fonction `labelled::look_for()` est tout à fait adaptée pour générer un dictionnaire de codification. Ses différentes options sont détaillées dans une [vignette dédiée](#). Les résultats renvoyés par `labelled::look_for()` sont récupérables dans un tableau de données que l'on pourra ainsi manipuler à sa guise.

```
gtsummary::trial |>
  look_for() |>
  dplyr::as_tibble()
```

```
# A tibble: 8 x 6
   pos variable label          col_type levels      value_labels
  <int> <chr>      <chr>          <chr>   <named list> <named list>
1     1 trt      Chemotherapy Treatment chr     <NULL>      <NULL>
2     2 age      Age              dbl     <NULL>      <NULL>
3     3 marker    Marker Level (ng/mL) dbl     <NULL>      <NULL>
4     4 stage     T Stage          fct     <chr [4]>    <NULL>
5     5 grade     Grade            fct     <chr [3]>    <NULL>
6     6 response  Tumor Response   int     <NULL>      <NULL>
7     7 death     Patient Died     int     <NULL>      <NULL>
8     8 ttdeath   Months to Death/Censor dbl     <NULL>      <NULL>
```

11.2 Manipulation sur un vecteur / une colonne

La fonction `labelled::var_label()` permet de voir l'étiquette de variable attachée à un vecteur (renvoie `NULL` s'il n'y en a pas) mais également d'ajouter/modifier une étiquette.

Le fait d'ajouter une étiquette de variable à un vecteur ne modifie en rien son type ni sa classe. On peut associer une

étiquette de variable à n'importe quel type de variable, qu'elle soit numérique, textuelle, un facteur ou encore des dates.

```
v <- c(1, 5, 2, 4, 1)
v |> var_label()
```

NULL

```
var_label(v) <- "Mon étiquette"
var_label(v)
```

```
[1] "Mon étiquette"
```

```
str(v)
```

```
num [1:5] 1 5 2 4 1
- attr(*, "label")= chr "Mon étiquette"
```

```
var_label(v) <- "Une autre étiquette"
var_label(v)
```

```
[1] "Une autre étiquette"
```

```
str(v)
```

```
num [1:5] 1 5 2 4 1
- attr(*, "label")= chr "Une autre étiquette"
```

Pour supprimer une étiquette, il suffit d'attribuer la valeur NULL.

```
var_label(v) <- NULL
str(v)
```

```
num [1:5] 1 5 2 4 1
```

On peut appliquer `labelled::var_label()` directement sur une colonne de tableau.

```
var_label(iris$Petal.Length) <- "Longueur du pétale"
var_label(iris$Petal.Width) <- "Largeur du pétale"
var_label(iris$Species) <- "Espèce"
iris |>
  look_for()
```

pos	variable	label	col_type	values
1	Sepal.Length	-	dbl	
2	Sepal.Width	-	dbl	
3	Petal.Length	Longueur du pétale	dbl	
4	Petal.Width	Largeur du pétale	dbl	
5	Species	Espèce	fct	setosa versicolor virginica

11.3 Manipulation sur un tableau de données

La fonction `labelled::set_variable_labels()` permet de manipuler les étiquettes de variable d'un tableau de données avec une syntaxe du type `{dplyr}`.

```
iris <-
  iris |>
  set_variable_labels(
    Species = NULL,
    Sepal.Length = "Longueur du sépale"
  )
iris |>
  look_for()
```

pos	variable	label	col_type	values
1	Sepal.Length	Longueur du sépale	dbl	
2	Sepal.Width	-	dbl	

```

3  Petal.Length Longueur du pétale dbl
4  Petal.Width  Largeur du pétale  dbl
5  Species      -                  fct      setosa
                                         versicolor
                                         virginica

```

11.4 Préserver les étiquettes

Certaines fonctions de **R** ne préservent pas les attributs et risquent donc d'effacer les étiquettes de variables que l'on a défini. Un exemple est la fonction générique **subset()** qui permet de sélectionner certaines lignes remplissant une certaines conditions.

```

iris |>
  look_for()

```

```

pos variable      label      col_type values
1  Sepal.Length  Longueur du sépale  dbl
2  Sepal.Width   -            dbl
3  Petal.Length  Longueur du pétale  dbl
4  Petal.Width   Largeur du pétale  dbl
5  Species       -            fct      setosa
                                         versicolor
                                         virginica

```

```

iris |>
  subset(Species == "setosa") |>
  look_for()

```

```

pos variable      label col_type values
1  Sepal.Length -      dbl
2  Sepal.Width   -      dbl
3  Petal.Length -      dbl
4  Petal.Width   -      dbl
5  Species       -      fct      setosa
                                         versicolor
                                         virginica

```


On pourra, dans ce cas précis, préférer la fonction `dplyr::filter()` qui préserve les attributs et donc les étiquettes de variables.

```
iris |>
  dplyr::filter(Species == "setosa") |>
  look_for()
```

pos	variable	label	col_type	values
1	Sepal.Length	Longueur du sépale	dbl	
2	Sepal.Width	-	dbl	
3	Petal.Length	Longueur du pétale	dbl	
4	Petal.Width	Largeur du pétale	dbl	
5	Species	-	fct	setosa versicolor virginica

On pourra également tirer parti de la fonction `labelled::copy_labels_from()` qui permet de copier les étiquettes d'un tableau à un autre.

```
iris |>
  subset(Species == "setosa") |>
  copy_labels_from(iris) |>
  look_for()
```

pos	variable	label	col_type	values
1	Sepal.Length	Longueur du sépale	dbl	
2	Sepal.Width	-	dbl	
3	Petal.Length	Longueur du pétale	dbl	
4	Petal.Width	Largeur du pétale	dbl	
5	Species	-	fct	setosa versicolor virginica

12 Étiquettes de valeurs

Dans le domaine des grandes enquêtes, il est fréquent de coder les variables catégorielles avec des codes numériques auxquels on associe une certaine valeur. Par exemple, une variable *milieu de résidence* pourrait être codée 1 pour urbain, 2 pour semi-urbain, 3 pour rural et 9 pour indiquer une donnée manquante. Une variable binaire pourrait quant à elle être codée 0 pour non et 1 pour oui. Souvent, chaque enquête définit ses propres conventions.

Les logiciels statistiques propriétaires **SPSS**, **Stata** et **SAS** ont tous les trois un système d'étiquettes de valeurs pour représenter ce type de variables catégorielles.

R n'a pas, de manière native, de système d'étiquettes de valeurs. Le format utilisé en interne pour représenter les variables catégorielles est celui des facteurs (cf. Chapitre 9). Cependant, ce dernier ne permet de contrôler comment sont associées une étiquette avec une valeur numérique précise.

12.1 La classe `haven_labelled`

Afin d'assurer une importation complète des données depuis **SPSS**, **Stata** et **SAS**, le package `{haven}` a introduit un nouveau type de vecteurs, la classe `haven_labelled`, qui permet justement de rendre compte de ces vecteurs labellisés (i.e. avec des étiquettes de valeurs). Le package `{labelled}` fournit un jeu de fonctions pour faciliter la manipulation des vecteurs labellisés.

! Important

Les vecteurs labellisés sont un format intermédiaire qui permet d'importer les données telles qu'elles ont été définies dans le fichier source. Il n'est pas destiné à être utilisé pour l'analyse statistique.

Pour la réalisation de tableaux, graphiques, modèles, **R** attend que les variables catégorielles soit codées sous formes de facteurs, et que les variables continues soient numériques. On aura donc besoin, à un moment ou à un autre, de convertir les vecteurs labellisés en facteurs ou en variables numériques classiques.

12.2 Manipulation sur un vecteur / une colonne

Pour définir des étiquettes, la fonction de base est `labelled::val_labels()`. Il est possible de définir des étiquettes de valeurs pour des vecteurs numériques, d'entiers et textuels. On indiquera les étiquettes sous la forme `étiquette = valeur`. Cette fonction s'utilise de la même manière que `labelled::var_label()` abordée au chapitre précédent (cf. Chapitre 11). Un appel simple renvoie les étiquettes de valeur associées au vecteur, `NULL` s'il n'y en n'a pas. Combiner avec l'opérateur d'assignation (`<-`), on peut ajouter/modifier les étiquettes de valeurs associées au vecteur.

```
library(labelled)
v <- c(1, 2, 1, 9)
v
```

```
[1] 1 2 1 9
```

```
class(v)
```

```
[1] "numeric"
```

```
val_labels(v)
```

NULL

```
val_labels(v) <- c(non = 1, oui = 2)
val_labels(v)
```

```
non oui
  1   2
```

```
v
```

```
<labelled<double>[4]>
[1] 1 2 1 9
```

```
Labels:
  value label
    1   non
    2   oui
```

```
class(v)
```

```
[1] "haven_labelled" "vctrs_vctr"      "double"
```

Comme on peut le voir avec cet exemple simple :

- l'ajout d'étiquettes de valeurs modifie la class de l'objet (qui est maintenant un vecteur de la classe `haven_labelled`) ;
- l'objet obtenu est multi-classes, la classe `double` indiquant ici qu'il s'agit d'un vecteur numérique ;
- il n'est pas obligatoire d'associer une étiquette de valeurs à toutes les valeurs observées dans le vecteur (ici, nous n'avons pas défini d'étiquettes pour la valeur 9).

La fonction `labelled::val_label()` (notez l'absence d'un `s` à la fin du nom de la fonction) permet d'accéder / de modifier l'étiquette associée à une valeur spécifique.

```
val_label(v, 1)
```

```
[1] "non"
```

```
val_label(v, 9)
```

```
NULL
```

```
val_label(v, 9) <- "(manquant)"  
val_label(v, 2) <- NULL  
v
```

```
<labelled<double>[4]>  
[1] 1 2 1 9
```

```
Labels:
```

value	label
1	non
9	(manquant)

Pour supprimer, toutes les étiquettes de valeurs, on attribuera NULL avec `labelled::val_labels()`.

```
val_labels(v) <- NULL  
v
```

```
[1] 1 2 1 9
```

```
class(v)
```

```
[1] "numeric"
```

On remarquera que, lorsque toutes les étiquettes de valeurs sont supprimées, la nature de l'objet change à nouveau et il redevient un simple vecteur numérique.

Mise en garde

Il est essentiel de bien comprendre que l'ajout d'étiquettes de valeurs ne change pas fondamentalement la nature du vecteur. **Cela ne le transforme pas en variable catégorielle.** À ce stade, le vecteur n'a pas été transformé en facteur. Cela reste un vecteur numérique qui est considéré comme tel par **R**. On peut ainsi en calculer une moyenne, ce qui serait impossible avec un facteur.

```
v <- c(1, 2, 1, 2)
val_labels(v) <- c(non = 1, oui = 2)
mean(v)
```

```
[1] 1.5
```

```
f <- factor(v, levels = c(1, 2), labels = c("non", "oui"))
mean(f)
```

```
Warning in mean.default(f): l'argument n'est ni numérique, ni logique : renvoi
de NA
```

```
[1] NA
```

Les fonctions `labelled::val_labels()` et `labelled::val_label()` peuvent également être utilisées sur les colonnes d'un tableau.

```
df <- dplyr::tibble(
  x = c(1, 2, 1, 2),
  y = c(3, 9, 9, 3)
)
val_labels(df$x) <- c(non = 1, oui = 2)
val_label(df$y, 9) <- "(manquant)"
df
```

```
# A tibble: 4 x 2
```

```
  x     y
<dbl> <dbl>
1 1 [non] 3
```

```
2 2 [oui]    9 [(manquant)]
3 1 [non]    9 [(manquant)]
4 2 [oui]    3
```

On pourra noter, que si notre tableau est un *tibble*, les étiquettes sont rendues dans la console quand on affiche le tableau.

La fonction `labelled::look_for()` est également un bon moyen d'afficher les étiquettes de valeurs.

```
df |>
  look_for()
```

```
pos variable label col_type values
1   x          -   dbl+lbl  [1] non
                        [2] oui
2   y          -   dbl+lbl  [9] (manquant)
```

12.3 Manipulation sur un tableau de données

{labelled} fournit 3 fonctions directement applicables sur un tableau de données : `labelled::set_value_labels()`, `labelled::add_value_labels()` et `labelled::remove_value_labels()`.

La première remplace l'ensemble des étiquettes de valeurs associées à une variable, la seconde ajoute des étiquettes de valeurs (et conserve celles déjà définies), la troisième supprime les étiquettes associées à certaines valeurs spécifiques (et laisse les autres inchangées).

```
df |>
  look_for()
```

```
pos variable label col_type values
1   x          -   dbl+lbl  [1] non
                        [2] oui
2   y          -   dbl+lbl  [9] (manquant)
```

```
df <- df |>
  set_value_labels(
    x = c(yes = 2),
    y = c("a répondu" = 3, "refus de répondre" = 9)
  )
df |>
  look_for()
```

pos	variable	label	col_type	values
1	x	-	dbl+lbl	[2] yes
2	y	-	dbl+lbl	[3] a répondu
				[9] refus de répondre

```
df <- df |>
  add_value_labels(
    x = c(no = 1)
  ) |>
  remove_value_labels(
    y = 9
  )
df |>
  look_for()
```

pos	variable	label	col_type	values
1	x	-	dbl+lbl	[2] yes
				[1] no
2	y	-	dbl+lbl	[3] a répondu

12.4 Conversion

12.4.1 Quand convertir les vecteurs labellisés ?

La classe `haven_labelled` permet d'ajouter des métadonnées aux variables sous la forme d'étiquettes de valeurs. Lorsque les données sont importées depuis **SAS**, **SPSS** ou **Stata**, cela permet notamment de conserver le codage original du fichier importé.

Mais il faut noter que ces *étiquettes de valeur* n'indiquent pas pour autant de manière systématique le type de variable (catégorielle ou continue). Les vecteurs labellisés n'ont donc pas vocation à être utilisés pour l'analyse, notamment le calcul de modèles statistiques. Ils doivent être convertis en facteurs (pour les variables catégorielles) ou en vecteurs numériques (pour les variables continues).

La question qui peut se poser est donc de choisir à quel moment cette conversion doit avoir lieu dans un processus d'analyse. On peut considérer deux approches principales.

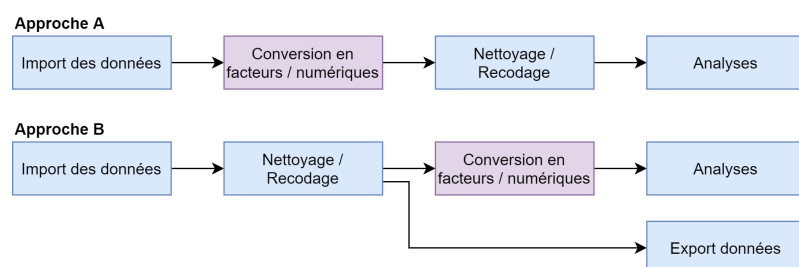


Figure 12.1: Deux approches possibles pour la conversion des étiquettes de valeurs

Dans l'**approche A**, les vecteurs labellisés sont convertis juste après l'import des données, en utilisant les fonctions `labelled::unlabelled()`, `labelled::to_factor()` ou `base::unclass()` qui sont présentées ci-après. Dès lors, toute la partie de nettoyage et de recodage des données se fera en utilisant les fonctions classiques de **R**. Si l'on n'a pas besoin de conserver le codage original, cette approche a l'avantage de s'inscrire dans le fonctionnement usuel de **R**.

Dans l'**approche B**, les vecteurs labellisés sont conservés pour l'étape de nettoyage et de recodage des données. Dans ce cas là, on pourra avoir recours aux fonctions de l'extension `{labelled}` qui facilitent la gestion des données labellisées. Cette approche est particulièrement intéressante quand (i) on veut pouvoir se référer au dictionnaire de codification fourni avec les données sources et donc on veut conserver le codage original et/ou (ii) quand les données devront faire l'objet d'un réexport après transformation. Par contre, comme dans

l'approche A, il faudra prévoir une conversion des variables labellisées au moment de l'analyse.

Avertissement

Dans tous les cas, il est recommandé d'adopter l'une ou l'autre approche, mais d'éviter de mélanger les différents types de vecteur. Une organisation rigoureuse de ses données et de son code est essentielle !

12.4.2 Convertir un vecteur labellisé en facteur

Il est très facile de convertir un vecteur labellisé en facteur à l'aide la fonction `labelled::to_factor()` du package `{labelled}`¹⁷.

```
v <- c(1,2,9,3,3,2,NA)
val_labels(v) <- c(
  oui = 1, "peut-être" = 2,
  non = 3, "ne sait pas" = 9
)
v
```

```
<labelled<double>[7]>
[1] 1 2 9 3 3 2 NA
```

Labels:

value	label
1	oui
2	peut-être
3	non
9	ne sait pas

```
to_factor(v)
```

```
[1] oui      peut-être ne sait pas non      non      peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

¹⁷ On privilégiera la fonction `labelled::to_factor()` à la fonction `haven::as_factor()` de l'extension `{haven}`, la première ayant plus de possibilités et un comportement plus consistant.

Il est possible d'indiquer si l'on souhaite, comme étiquettes du facteur, utiliser les étiquettes de valeur (par défaut), les valeurs elles-mêmes, ou bien les étiquettes de valeurs préfixées par la valeur d'origine indiquée entre crochets.

```
to_factor(v, 'l')
```

```
[1] oui          peut-être   ne sait pas non          non          peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

```
to_factor(v, 'v')
```

```
[1] 1    2    9    3    3    2    <NA>
Levels: 1 2 3 9
```

```
to_factor(v, 'p')
```

```
[1] [1] oui          [2] peut-être   [9] ne sait pas [3] non
[5] [3] non          [2] peut-être   <NA>
Levels: [1] oui [2] peut-être [3] non [9] ne sait pas
```

Par défaut, les modalités du facteur seront triées selon l'ordre des étiquettes de valeur. Mais cela peut être modifié avec l'argument `sort_levels` si l'on préfère trier selon les valeurs ou selon l'ordre alphabétique des étiquettes.

```
to_factor(v, sort_levels = 'v')
```

```
[1] oui          peut-être   ne sait pas non          non          peut-être
[7] <NA>
Levels: oui peut-être non ne sait pas
```

```
to_factor(v, sort_levels = 'l')
```

```
[1] oui          peut-être   ne sait pas non          non          peut-être
[7] <NA>
Levels: ne sait pas non oui peut-être
```

12.4.3 Convertir un vecteur labellisé en numérique ou en texte

Pour rappel, il existe deux types de vecteurs labellisés : des vecteurs numériques labellisés (x dans l'exemple ci-dessous) et des vecteurs textuels labellisés (y dans l'exemple ci-dessous).

```
x <- c(1, 2, 9, 3, 3, 2, NA)
val_labels(x) <- c(
  oui = 1, "peut-être" = 2,
  non = 3, "ne sait pas" = 9
)

y <- c("f", "f", "h", "f")
val_labels(y) <- c(femme = "f", homme = "h")
```

Pour leur retirer leur caractère labellisé et revenir à leur classe d'origine, on peut utiliser la fonction `unclass()`.

```
unclass(x)
```

```
[1] 1 2 9 3 3 2 NA
attr(,"labels")
      oui      peut-être      non ne sait pas
      1          2          3          9
```

```
unclass(y)
```

```
[1] "f" "f" "h" "f"
attr(,"labels")
femme homme
"f"      "h"
```

À noter que dans ce cas-là, les étiquettes sont conservées comme attributs du vecteur.

Une alternative est d'utiliser `labelled::remove_labels()` qui supprimera toutes les étiquettes, y compris les étiquettes de variable. Pour conserver les étiquettes de variables et

ne supprimer que les étiquettes de valeurs, on indiquera `keep_var_label = TRUE`.

```
var_label(x) <- "Etiquette de variable"
remove_labels(x)
```

```
[1] 1 2 9 3 3 2 NA
```

```
remove_labels(x, keep_var_label = TRUE)
```

```
[1] 1 2 9 3 3 2 NA
attr("label")
[1] "Etiquette de variable"
```

```
remove_labels(y)
```

```
[1] "f" "f" "h" "f"
```

Dans le cas d'un vecteur numérique labellisé que l'on souhaiterait convertir en variable textuelle, on pourra utiliser `labelled::to_character()` à la place de `labelled::to_factor()` qui, comme sa grande soeur, utilisera les étiquettes de valeurs.

```
to_character(x)
```

```
[1] "oui"          "peut-être"    "ne sait pas" "non"          "non"
[6] "peut-être"    NA
attr("label")
[1] "Etiquette de variable"
```

12.4.4 Conversion conditionnelle en facteurs

Il n'est pas toujours possible de déterminer la nature d'une variable (continue ou catégorielle) juste à partir de la présence ou l'absence d'étiquettes de valeur. En effet, on peut utiliser des étiquettes de valeur dans le cadre d'une variable continue pour indiquer certaines valeurs spécifiques.

Une bonne pratique est de vérifier chaque variable incluse dans une analyse, une à une.

Cependant, une règle qui fonctionne dans 90% des cas est de convertir un vecteur labellisé en facteur si et seulement si toutes les valeurs observées dans le vecteur disposent d'une étiquette de valeur correspondante. C'est ce que propose la fonction `labelled::unlabelled()` qui peut même être appliqué à tout un tableau de données. Par défaut, elle fonctionne ainsi :

1. les variables non labellisées restent inchangées (variables *f* et *g* dans l'exemple ci-dessous);
2. si toutes les valeurs observées d'une variable labellisées ont une étiquette, elles sont converties en facteurs (variables *b* et *c*);
3. sinon, on leur applique `base::unclass()` (variables *a*, *d* et *e*).

```
df <- dplyr::tibble(  
  a = c(1, 1, 2, 3),  
  b = c(1, 1, 2, 3),  
  c = c(1, 1, 2, 2),  
  d = c("a", "a", "b", "c"),  
  e = c(1, 9, 1, 2),  
  f = 1:4,  
  g = as.Date(c(  
    "2020-01-01", "2020-02-01",  
    "2020-03-01", "2020-04-01"  
  ))  
) |>  
  set_value_labels(  
    a = c(No = 1, Yes = 2),  
    b = c(No = 1, Yes = 2, DK = 3),
```

```

    c = c(No = 1, Yes = 2, DK = 3),
    d = c(No = "a", Yes = "b"),
    e = c(No = 1, Yes = 2)
  )
df |> look_for()

```

pos	variable	label	col_type	values
1	a	-	dbl+lbl	[1] No [2] Yes
2	b	-	dbl+lbl	[1] No [2] Yes [3] DK
3	c	-	dbl+lbl	[1] No [2] Yes [3] DK
4	d	-	chr+lbl	[a] No [b] Yes
5	e	-	dbl+lbl	[1] No [2] Yes
6	f	-	int	
7	g	-	date	

```

to_factor(df) |> look_for()

```

pos	variable	label	col_type	values
1	a	-	fct	No Yes 3
2	b	-	fct	No Yes DK
3	c	-	fct	No Yes DK
4	d	-	fct	No Yes c
5	e	-	fct	No

				Yes
6	f	-	int	9
7	g	-	date	

```
unlabelled(df) |> look_for()
```

pos	variable	label	col_type	values
1	a	-	dbl	
2	b	-	fct	No Yes DK
3	c	-	fct	No Yes DK
4	d	-	chr	
5	e	-	dbl	
6	f	-	int	
7	g	-	date	

On peut indiquer certaines options, par exemple `drop_unused_labels = TRUE` pour supprimer des facteurs créés les niveaux non observés dans les données (voir la variable *c*).

```
unlabelled(df, drop_unused_labels = TRUE) |>
  look_for()
```

pos	variable	label	col_type	values
1	a	-	dbl	
2	b	-	fct	No Yes DK
3	c	-	fct	No Yes
4	d	-	chr	
5	e	-	dbl	
6	f	-	int	
7	g	-	date	


```
unlabelled(df, levels = "prefixed") |>
  look_for()
```

pos	variable	label	col_type	values
1	a	-	dbl	
2	b	-	fct	[1] No [2] Yes [3] DK
3	c	-	fct	[1] No [2] Yes [3] DK
4	d	-	chr	
5	e	-	dbl	
6	f	-	int	
7	g	-	date	

13 Valeurs manquantes

Dans **R** base, les valeurs manquantes sont indiquées par la valeurs logiques **NA** que l'on peut utiliser dans tous types de vecteurs.

Dans certains cas, par exemple dans la fonction `dplyr::if_else()` qui vérifie que les deux options sont du même type, on peut avoir besoin de spécifier une valeur manquante d'un certains types précis (numérique, entier, textuel...) ce que l'on peut faire avec les constantes `NA_real_`, `NA_integer_` ou encore `NA_character_`.

De base, il n'existe qu'un seul type de valeurs manquantes dans **R**. D'autres logiciels statistiques ont mis en place des systèmes pour distinguer plusieurs types de valeurs manquantes, ce qui peut avoir une importance dans le domaine des grandes enquêtes, par exemple pour distinguer des ne sait pas d'un refus de répondre ou d'un oubli enquêteur.

Ainsi, **Stata** et **SAS** ont un système de valeurs manquantes étiquettées ou *tagged NAs*, où les valeurs manquantes peuvent recevoir une étiquette (une lettre entre a et z). De son côté, **SPSS** permet d'indiquer, sous la forme de métadonnées, que certaines valeurs devraient être traitées comme des valeurs manquantes (par exemple que la valeur 8 correspond à des refus et que la valeur 9 correspond à des ne sait pas). Il s'agit alors de valeurs manquantes définies par l'utilisateur ou *user NAs*.

Dans tous les cas, il appartient à l'analyste de décider au cas par cas comment ces valeurs manquantes doivent être traitées. Dans le cadre d'une variable numérique, il est essentiel d'exclure ces valeurs manquantes pour le calcul de statistiques telles que la moyenne ou l'écart-type. Pour des variables catégorielles, les pourcentages peuvent être calculées sur l'ensemble de l'échantillon (les valeurs manquantes étant

alors traitées comme des modalités à part entière) ou bien uniquement sur les réponses valides, en fonction du besoin de l'analyse et de ce que l'on cherche à montrer.

Afin d'éviter toute perte d'informations lors d'un import de données depuis **Stata**, **SAS** et **SPSS**, le package `{haven}` propose une implémentation sous **R** des *tagged NAs* et des *user NAs*. Le package `{labelled}` fournit quant à lui différentes fonctions pour les manipuler aisément.

```
library(labelled)
```

13.1 Valeurs manquantes étiquetées (*tagged NAs*)

13.1.1 Création et test

Les *tagged NAs* sont de véritables valeurs manquantes (**NA**) au sens de **R**, auxquelles a été attachées sur étiquette, une lettre unique minuscule (a-z) ou majuscule (A-Z). On peut les créer avec `labelled::tagged_na()`.

```
x <- c(1:3, tagged_na("a"), tagged_na("z"), NA)
```

Pour la plupart des fonctions de **R**, les *tagged NAs* sont juste considérées comme des valeurs manquantes régulières (*regular NAs*). Dès lors, par défaut, elles sont justes affichées à l'écran comme n'importe quelle valeur manquante et la fonction `is.na()` renvoie **TRUE**.

```
x
```

```
[1] 1 2 3 NA NA NA
```

```
is.na(x)
```

```
[1] FALSE FALSE FALSE TRUE TRUE TRUE
```

Pour afficher les étiquettes associées à ces valeurs manquantes, il faut avoir recours à `labelled::na_tag()`, `labelled::print_tagged_na()` ou encore `labelled::format_tagged_na()`.

```
na_tag(x)
```

```
[1] NA  NA  NA  "a" "z" NA
```

```
print_tagged_na(x)
```

```
[1]      1      2      3 NA(a) NA(z)      NA
```

```
format_tagged_na(x)
```

```
[1] "      1" "      2" "      3" "NA(a)" "NA(z)" "      NA"
```

Pour tester si une certaine valeur manquante est une *regular NA* ou une *tagged NA*, on aura recours à `labelled::is_regular_na()` et à `labelled::is_tagged_na()`.

```
is.na(x)
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
is_regular_na(x)
```

```
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
is_tagged_na(x)
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

Il est possible de tester une étiquette particulière en passant un deuxième argument à `labelled::is_tagged_na()`.

```
is_tagged_na(x, "a")
```

```
[1] FALSE FALSE FALSE TRUE FALSE FALSE
```

Note

Il n'est possible de définir des *tagged NAs* seulement pour des vecteurs numériques (*double*). Si l'on ajoute une *tagged NA* à un vecteur d'entiers, ce vecteur sera converti en vecteur numérique. Si on l'ajoute à un vecteur textuel, la valeur manquante sera convertie en *regular NA*.

```
y <- c("a", "b", tagged_na("z"))
y
```

```
[1] "a" "b" NA
```

```
is_tagged_na(y)
```

```
[1] FALSE FALSE FALSE
```

```
format_tagged_na(y)
```

Error: `x` must be a double vector

```
z <- c(1L, 2L, tagged_na("a"))
typeof(z)
```

```
[1] "double"
```

```
format_tagged_na(z)
```

```
[1] "      1" "      2" "NA(a)"
```

13.1.2 Valeurs uniques, doublons et tris

Par défaut, les fonctions classiques de **R** `unique()`, `duplicated()`, `ordered()` ou encore `sort()` traiteront les *tagged NAs* comme des valeurs manquantes tout ce qu'il y a de plus classique, et ne feront pas de différences entre des *tagged NAs* ayant des étiquettes différentes.

Pour traiter des *tagged NAs* ayant des étiquettes différentes comme des valeurs différentes, on aura recours aux fonctions `labelled::unique_tagged_na()`, `labelled::duplicated_tagged_na()`, `labelled::order_tagged_na()` ou encore `labelled::sort_tagged_na()`.

```
x <- c(1, 2, tagged_na("a"), 1, tagged_na("z"), 2, tagged_na("a"), NA)
x |>
  print_tagged_na()
```

```
[1]      1      2 NA(a)      1 NA(z)      2 NA(a)      NA
```

```
x |>
  unique() |>
  print_tagged_na()
```

```
[1]      1      2 NA(a)
```

```
x |>
  unique_tagged_na() |>
  print_tagged_na()
```

```
[1]      1      2 NA(a) NA(z)      NA
```

```
x |>
  duplicated()
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
x |>
  duplicated_tagged_na()
```

```
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

```
x |>
  sort(na.last = TRUE) |>
  print_tagged_na()
```

```
[1]      1      1      2      2 NA(a) NA(z) NA(a)      NA
```

```
x |>
  sort_tagged_na() |>
  print_tagged_na()
```

```
[1]      1      1      2      2 NA(a) NA(a) NA(z)      NA
```

13.1.3 Tagged NAs et étiquettes de valeurs

Il est tout à fait possible d'associer une étiquette de valeurs (cf. Chapitre 12) à des *tagged NAs*.

```
x <- c(
  1, 0,
  1, tagged_na("r"),
  0, tagged_na("d"),
  tagged_na("z"), NA
)
val_labels(x) <- c(
  no = 0,
  yes = 1,
  "don't know" = tagged_na("d"),
  refusal = tagged_na("r")
)
x
```

```
<labelled<double>[8]>
[1]      1      0      1 NA(r)      0 NA(d) NA(z)      NA
```

```
Labels:
 value      label
    0         no
    1         yes
NA(d) don't know
NA(r)   refusal
```

Lorsqu'un vecteur labellisé est converti en facteur avec `labelled::to_factor()`, les *tagged NAs* sont, par défaut convertis en en valeurs manquantes classiques (*regular NAs*). Il n'est pas possible de définir des *tagged NAs* pour des facteurs.

```
x |> to_factor()
```

```
[1] yes  no   yes  <NA> no   <NA> <NA> <NA>
Levels: no yes
```

L'option `explicit_tagged_na` de `labelled::to_factor()` permet de convertir les *tagged NAs* en modalités explicites du facteur.

```
x |>
  to_factor(explicit_tagged_na = TRUE)
```

```
[1] yes      no      yes      refusal    no      don't know NA(z)
[8] <NA>
Levels: no yes don't know refusal NA(z)
```

```
x |>
  to_factor(
    levels = "prefixed",
    explicit_tagged_na = TRUE
  )
```



```

[1] [1] yes           [0] no           [1] yes           [NA(r)] refusal
[5] [0] no           [NA(d)] don't know [NA(z)] NA(z)    <NA>
Levels: [0] no [1] yes [NA(d)] don't know [NA(r)] refusal [NA(z)] NA(z)

```

13.1.4 Conversion en user NAs

La fonction `labelled::tagged_na_to_user_na()` permet de convertir des *tagged NAs* en *user NAs*.

```

x |>
  tagged_na_to_user_na()

```

```

<labelled_spss<double>[8]>
[1] 1 0 1 3 0 2 4 NA
Missing range: [2, 4]

```

```

Labels:
value    label
0        no
1        yes
2 don't know
3    refusal
4      NA(z)

```

```

x |>
  tagged_na_to_user_na(user_na_start = 10)

```

```

<labelled_spss<double>[8]>
[1] 1 0 1 11 0 10 12 NA
Missing range: [10, 12]

```

```

Labels:
value    label
0        no
1        yes
10 don't know
11    refusal
12      NA(z)

```

La fonction `labelled::tagged_na_to_regular_na()` convertit les *tagged NAs* en valeurs manquantes classiques (*regular NAs*).

```
x |>
  tagged_na_to_regular_na()
```

```
<labelled<double>[8]>
[1]  1  0  1 NA  0 NA NA NA
```

Labels:

value	label
0	no
1	yes

```
x |>
  tagged_na_to_regular_na() |>
  is_tagged_na()
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

13.2 Valeurs manquantes définies par l'utilisateur (*user NAs*)

Le package `{haven}` a introduit la classe `haven_labelled_spss`, une extension de la classe `haven_labelled` permettant d'indiquer des valeurs à considérer comme manquantes à la manière de **SPSS**.

! Important

Cela revient à associer à un vecteur des attributs (cf. Chapitre 6) additionnels pour indiquer des valeurs que l'utilisateur pourrait/devrait considérer comme manquante. Cependant, il ne s'agit que de métadonnées et en interne ces valeurs ne sont pas stockées sous forme de `NA` mais restent des valeurs valides.

Il convient de garder en mémoire que la très grande majorité des fonctions de **R** ne prendront pas en compte ces métadonnées et traiteront donc ces valeurs comme des valeurs valides. C'est donc à l'utilisateur de convertir, au besoin, ces les valeurs indiquées comme manquantes en réelles valeurs manquantes (NA).

13.2.1 Création

Il est possible d'indiquer des valeurs à considérer comme manquantes (*user NAs*) de deux manières :

- soit en indiquant une liste de valeurs individuelles avec `labelled::na_values()` (on peut indiquer NULL pour supprimer les déclarations existantes) ;
- soit en indiquant deux valeurs représentant une plage de valeurs à considérées comme manquantes avec `labelled::na_range()` (seront considérées comme manquantes toutes les valeurs supérieures ou égale au premier chiffre et inférieures ou égales au second chiffre¹⁸).

```
v <- c(1, 2, 3, 9, 1, 3, 2, NA)
val_labels(v) <- c(
  faible = 1,
  fort = 3,
  "ne sait pas" = 9
)
na_values(v) <- 9
v
```

```
<labelled_spss<double>[8]>
[1] 1 2 3 9 1 3 2 NA
Missing values: 9
```

Labels:

value	label
1	faible
3	fort

¹⁸ On peut utiliser `-Inf` et `Inf` qui représentent respectivement moins l'infini et l'infini.

9 ne sait pas

```
na_values(v) <- NULL
v
```

```
<labelled<double>[8]>
[1] 1 2 3 9 1 3 2 NA
```

Labels:

value	label
1	faible
3	fort
9	ne sait pas

```
na_range(v) <- c(5, Inf)
v
```

```
<labelled_spss<double>[8]>
[1] 1 2 3 9 1 3 2 NA
Missing range: [5, Inf]
```

Labels:

value	label
1	faible
3	fort
9	ne sait pas

On peut noter que les *user NAs* peuvent cohabiter avec des *regular NAs* ainsi qu'avec des étiquettes de valeurs (*value labels*, cf. Chapitre 12).

Pour manipuler les variables d'un tableau de données, on peut également avoir recours à `labelled::set_na_values()` et `labelled::set_na_range()`.

```
df <-
  dplyr::tibble(
    s1 = c("M", "M", "F", "F"),
```

```

    s2 = c(1, 1, 2, 9)
  ) |>
  set_na_values(s2 = 9)
df$s2

```

```

<labelled_spss<double>[4]>
[1] 1 1 2 9
Missing values: 9

```

```

df <-
df |>
  set_na_values(s2 = NULL)
df$s2

```

```

<labelled<double>[4]>
[1] 1 1 2 9

```

13.2.2 Tests

La fonction `is.na()` est l'une des rares fonctions de base **R** à reconnaître les *user NAs* et donc à renvoyer `TRUE` dans ce cas. Pour des tests plus spécifiques, on aura recours à `labelled::is_user_na()` et `labelled::is_regular_na()`.

```

v

```

```

<labelled_spss<double>[8]>
[1] 1 2 3 9 1 3 2 NA
Missing range: [5, Inf]

```

Labels:

value	label
1	faible
3	fort
9	ne sait pas

```
v |> is.na()
```

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

```
v |> is_user_na()
```

```
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

```
v |> is_regular_na()
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

13.2.3 Conversion

Comme dit précédemment, pour la plupart des fonctions de **R**, les *users NAs* sont toujours des valeurs valides.

```
x <- c(1:5, 11:15)
na_range(x) <- c(10, Inf)
x
```

```
<labelled_spss<integer>[10]>
[1]  1  2  3  4  5 11 12 13 14 15
Missing range:  [10, Inf]
```

```
mean(x)
```

```
[1] 8
```

On aura alors recours à `labelled::user_na_to_regular_na()` pour convertir les *users NAs* en véritables valeurs manquantes avant de procéder à un calcul statistique.

```
x |>
  user_na_to_na()
```

```
<labelled<integer>[10]>
[1]  1  2  3  4  5 NA NA NA NA NA
```

```
x |>
  user_na_to_na() |>
  mean(na.rm = TRUE)
```

```
[1] 3
```

Une alternative consiste à transformer les *user NAs* en *tagged NAs* avec `labelled::user_na_to_tagged_na()`.

```
x |>
  user_na_to_tagged_na() |>
  print_tagged_na()
```

'x' has been converted into a double vector.

```
[1]      1      2      3      4      5 NA(a) NA(b) NA(c) NA(d) NA(e)
```

```
x |>
  user_na_to_tagged_na() |>
  mean(na.rm = TRUE)
```

'x' has been converted into a double vector.

```
[1] 3
```

Pour supprimer les métadonnées relatives aux *user NAs* sans les convertir en valeurs manquantes, on aura recours à `labelled::remove_user_na()`.

```
x |>
  remove_user_na()
```

```
<labelled<integer>[10]>
[1]  1  2  3  4  5 11 12 13 14 15
```

```
x |>
  remove_user_na() |>
  mean()
```

```
[1] 8
```

Enfin, lorsque l'on convertit un vecteur labellisé en facteur avec `labelled::to_factor()`, on pourra utiliser l'argument `user_na_to_na` pour indiquer si les *users NAs* doivent être convertis ou non en valeurs manquantes classiques (NA).

```
x <- c(1, 2, 9, 2)
val_labels(x) <- c(oui = 1, non = 2, refus = 9)
na_values(x) <- 9
x |>
  to_factor(user_na_to_na = TRUE)
```

```
[1] oui  non  <NA> non
Levels: oui non
```

```
x |>
  to_factor(user_na_to_na = FALSE)
```

```
[1] oui  non  refus non
Levels: oui non refus
```


14 Import & Export de données

14.1 Importer un fichier texte

Les fichiers texte constituent un des formats les plus largement supportés par la majorité des logiciels statistiques. Presque tous permettent d'exporter des données dans un format texte, y compris les tableurs comme **Libre Office**, **Open Office** ou **Excel**.

Cependant, il existe une grande variété de format texte, qui peuvent prendre différents noms selon les outils, tels que texte tabulé ou *texte (séparateur : tabulation)*, **CSV** (pour *comma-separated value*, sachant que suivant les logiciels le séparateur peut être une virgule ou un point-virgule).

14.1.1 Structure d'un fichier texte

Dès lors, avant d'importer un fichier texte dans **R**, il est indispensable de regarder comment ce dernier est structuré. Il importe de prendre note des éléments suivants :

- La première ligne contient-elle le nom des variables ?
- Quel est le caractère séparateur entre les différentes variables (encore appelé séparateur de champs) ? Dans le cadre d'un fichier **CSV**, il aurait pu s'agir d'une virgule ou d'un point-virgule.
- Quel est le caractère utilisé pour indiquer les décimales (le séparateur décimal) ? Il s'agit en général d'un point (à l'anglo-saxonne) ou d'une virgule (à la française).
- Les valeurs textuelles sont-elles encadrées par des guillemets et, si oui, s'agit-il de guillemets simple (') ou de guillemets doubles (") ?

- Pour les variables textuelles, y a-t-il des valeurs manquantes et si oui comment sont-elles indiquées ? Par exemple, le texte NA est parfois utilisé.

Il ne faut pas hésitez à ouvrir le fichier avec un éditeur de texte pour le regarder de plus près.

14.1.2 Interface graphique avec RStudio

RStudio fournit une interface graphique pour faciliter l'import d'un fichier texte. Pour cela, il suffit d'aller dans le menu *File* > *Import Dataset* et de choisir l'option *From CSV*¹⁹. Cette option est également disponible via l'onglet *Environment* dans le quadrant haut-droite.

¹⁹ L'option CSV fonctionne pour tous les fichiers de type texte, même si votre fichier a une autre extension, .txt par exemple

Pour la suite, nous allons utiliser ce [fichier texte à titre d'exemple](#).

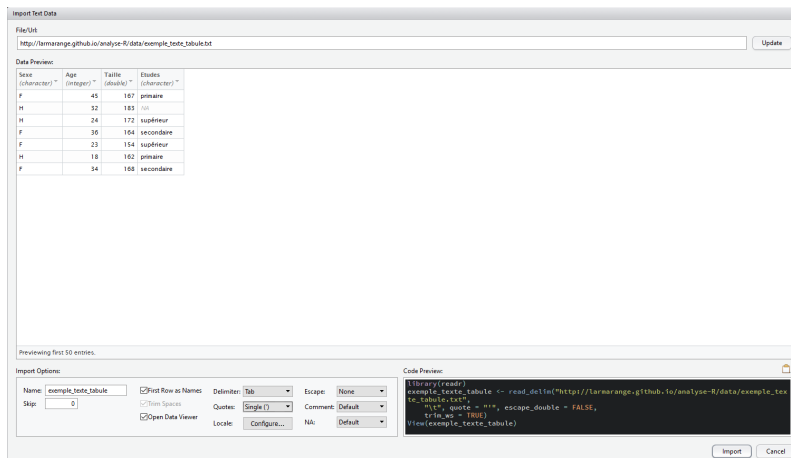


Figure 14.1: Importer un fichier texte avec RStudio

L'interface de **RStudio** vous présente sous *Import Options* les différentes options d'import disponible. La section *Data Preview* vous permet de voir en temps réel comment les données sont importées. La section *Code Preview* vous indique le code **R** correspondant à vos choix. Il n'y a plus qu'à le copier/coller dans un de vos scripts ou à cliquer sur **Import** pour l'exécuter.

Vous pourrez remarquer que **RStudio** fait appel à l'extension `{readr}` du tidyverse pour l'import des données via la fonction `readr::read_csv()`.

`{readr}` essaie de deviner le type de chacune des colonnes, en se basant sur les premières observations. En cliquant sur le nom d'une colonne, il est possible de modifier le type de la variable importée. Il est également possible d'exclure une colonne de l'import (*skip*).

14.1.3 Dans un script

L'interface graphique de **RStudio** fournit le code d'import. On peut également l'adapter à ces besoins en consultant la page d'aide de `readr::read_csv()` pour plus de détails. Par exemple :

```
library(readr)
d <- read_delim(
  "http://larmarange.github.io/analyse-R/data/exemple_texte_tabule.txt",
  delim = "\t",
  quote = ""
)
```

On peut indiquer le chemin local vers un fichier (le plus courant) ou bien directement l'URL d'un fichier sur Internet.

`{readr}` propose plusieurs fonctions proches : `readr::read_delim()`, `readr::read_csv()`, `readr::read_csv2()` et `readr::read_tsv()`. Elles fonctionnent toutes de manière identique et ont les mêmes arguments. Seule différence, les valeurs par défaut de certains paramètres.

Fichiers de très grande taille

Si vous travaillez sur des données de grandes dimensions, les formats texte peuvent être lents à exporter et importer. Dans ce cas là, on pourra jeter un œil au package `{vroom}` et/ou aux fonctions `data.table::fread()` et `data.table::fwrite()`.

Dans des manuels ou des exemples en ligne, vous trouverez parfois mention des fonctions `utils::read.table()`, `utils::read.csv()`, `utils::read.csv2()`, `utils::read.delim()` ou encore `utils::read.delim2()`. Il s'agit des fonctions natives et historiques de **R** (extension `{utils}`) dédiées à l'import de fichiers textes. Elles sont similaires à celles de `{readr}` dans l'idée générale mais diffèrent dans leurs détails et les traitements effectués sur les données (pas de détection des dates par exemple). Pour plus d'information, vous pouvez vous référer à la page d'aide de ces fonctions.

14.2 Importer un fichier Excel

Une première approche pour importer des données **Excel** dans **R** consiste à les exporter depuis **Excel** dans un fichier texte (texte tabulé ou **CSV**) puis de suivre la procédure d'importation d'un fichier texte.

Une feuille **Excel** peut également être importée directement avec l'extension `{readxl}` du *tidyverse*.

La fonction `readxl::read_excel()` permet d'importer à la fois des fichiers `.xls` (**Excel** 2003 et précédents) et `.xlsx` (**Excel** 2007 et suivants).

```
library(readxl)
donnees <- read_excel("data/fichier.xlsx")
```

Une seule feuille de calculs peut être importée à la fois. On pourra préciser la feuille désirée avec `sheet` en indiquant soit le nom de la feuille, soit sa position (première, seconde, ...).

```
donnees <- read_excel("data/fichier.xlsx", sheet = 3)
donnees <- read_excel("data/fichier.xlsx", sheet = "mes_donnees")
```

On pourra préciser avec `col_names` si la première ligne contient le nom des variables.

Par défaut, `readxl::read_excel()` va essayer de deviner le type (numérique, textuelle, date) de chaque colonne. Au besoin,

on pourra indiquer le type souhaité de chaque colonne avec `col_types`.

RStudio propose également pour les fichiers **Excel** un assistant d'importation, similaire à celui pour les fichiers texte, permettant de faciliter l'import.

14.3 Importer depuis des logiciels de statistique

Le package `{haven}` du *tidyverse* a été développé spécifiquement pour permettre l'importation de données depuis les formats des logiciels **Stata**, **SAS** et **SPSS**.

Il vise à offrir une importation unifiée depuis ces trois logiciels (là où le package `{foreign}` distribué en standard avec **R** adopte des conventions différentes selon le logiciel source).

Afin de ne pas perdre d'information lors de l'import, `{haven}` a introduit la notion d'étiquettes de variables (cf. Chapitre 11), une classe de vecteurs pour la gestion des étiquettes de valeurs (cf. Chapitre 12), des mécanismes pour reproduire la gestion des valeurs manquantes de ces trois logiciels (cf. Chapitre 13), mais également une gestion et un import correct des dates, dates-heures et des variables horaires (cf. le package `{hms}`).

À noter que **RStudio** intègre également une interface graphique pour l'import des fichiers **Stata**, **SAS** et **SPSS**.

14.3.1 SPSS

Les fichiers générés par **SPSS** sont de deux types : les fichiers **SPSS natifs** (extension `.sav`) et les fichiers au format **SPSS export** (extension `.por`).

Dans les deux cas, on aura recours à la fonction `haven::read_spss()` :

```
library(haven)
donnees <- read_spss("data/fichier.sav", user_na = TRUE)
```

💡 Valeurs manquantes

Dans **SPSS**, il est possible de définir des valeurs à considérées comme manquantes ou *user NAs*, voir Chapitre 13. Par défaut, `haven::read_spss()` convertir toutes ces valeurs en `NA` lors de l'import.

Or, il est parfois important de garder les différentes valeurs originelles. Dans ce cas, on appellera `haven::read_spss()` avec l'option `user_na = TRUE`.

14.3.2 SAS

Les fichiers **SAS** se présentent en général sous deux format : format **SAS export** (extension `.xport` ou `.xpt`) ou format **SAS natif** (extension `.sas7bdat`).

Les fichiers **SAS natifs** peuvent être importées directement avec `haven::read_sas()` de l'extension `{haven}` :

```
library(haven)
donnees <- read_sas("data/fichier.sas7bdat")
```

Au besoin, on pourra préciser en deuxième argument le nom d'un fichier **SAS catalogue** (extension `.sas7bcatalog`) contenant les métadonnées du fichier de données.

```
library(haven)
donnees <- read_sas(
  "data/fichier.sas7bdat",
  catalog_file = "data/fichier.sas7bcatalog"
)
```

i Note

Les fichiers au format **SAS export** peuvent être importés via la fonction `foreign::read.xport()` de l'extension `{foreign}`. Celle-ci s'utilise très simplement, en lui passant le nom du fichier en argument :

```
library(foreign)
donnees <- read.xport("data/fichier.xpt")
```

14.3.3 Stata

Pour les fichiers **Stata** (extension `.dta`), on aura recours aux fonctions `haven::read_dta()` et `haven::read_stata()` de l'extension `{haven}`. Ces deux fonctions sont identiques.

```
library(haven)
donnees <- read_dta("data/fichier.dta")
```

! Important

Gestion des valeurs manquantes

Dans **Stata**, il est possible de définir plusieurs types de valeurs manquantes, qui sont notées sous la forme `.a` à `.z`. Elles sont importées par `{haven}` sous formes de *tagged NAs*, cf. Chapitre 13.

14.3.4 dBase

L'Insee et d'autres producteurs de données diffusent leurs fichiers au format **dBase** (extension `.dbf`). Ceux-ci sont directement lisibles dans **R** avec la fonction `foreign::read.dbf()` de l'extension `{foreign}`.

```
library(foreign)
donnees <- read.dbf("data/fichier.dbf")
```

14.4 Sauver ses données

R dispose également de son propre format pour sauvegarder et échanger des données. On peut sauver n'importe quel objet créé avec **R** et il est possible de sauver plusieurs objets dans

un même fichier. L'usage est d'utiliser l'extension `.RData` pour les fichiers de données **R**. La fonction à utiliser s'appelle tout simplement `save()`.

Par exemple, si l'on souhaite sauvegarder son tableau de données `d` ainsi que les objets `tailles` et `poids` dans un fichier `export.RData` :

```
save(d, tailles, poids, file = "export.RData")
```

À tout moment, il sera toujours possible de recharger ces données en mémoire à l'aide de la fonction `load()` :

```
load("export.RData")
```

Mise en garde

Si entre temps vous aviez modifié votre tableau `d`, vos modifications seront perdues. En effet, si lors du chargement de données, un objet du même nom existe en mémoire, ce dernier sera remplacé par l'objet importé.

La fonction `save.image()` est un raccourci pour sauvegarder tous les objets de la session de travail dans le fichier `.RData` (un fichier un peu étrange car il n'a pas de nom mais juste une extension). Lors de la fermeture de **RStudio**, il vous sera demandé si vous souhaitez enregistrer votre session. Si vous répondez *Oui*, c'est cette fonction `save.image()` qui sera appliquée.

```
save.image()
```

Un autre mécanisme possible est le format **RDS** de **R**. La fonction `saveRDS()` permet de sauvegarder **un et un seul** objet **R** dans un fichier.

```
saveRDS(d, file = "mes_donnees.rds")
```

Cet objet pourra ensuite être lu avec la fonction `readRDS()`. Mais au lieu d'être directement chargé dans la mémoire de l'environnement de travail, l'objet lu sera retourné

par la fonction `readRDS()` et ce sera à l'utilisateur de le sauvegarder.

```
donnees <- readRDS("mes_donnees.rds")
```

14.5 Export de tableaux de données

On peut avoir besoin d'exporter un tableau de données **R** vers différents formats. La plupart des fonctions d'import disposent d'un équivalent permettant l'export de données. On citera notamment :

- `readr::write_csv()` et `readr::write_tsv()` permettent d'exporter au format **CSV** et texte tabulé respectivement, `readr::write_delim()` offrant de multiples options pour l'export au format texte ;
- `haven::write_sas()` permet d'exporter au format **SAS** ;
- `haven::write_sav()` au format **SPSS** ;
- `haven::write_dta()` au format **Stata** ;
- `foreign::write.dbf()` au format **dBase**.

L'extension `readxl` ne fournit pas de fonction pour exporter au format **Excel**. Par contre, on pourra passer par la fonction `openxlsx::write.xlsx()` du package `{openxlsx}` ou la fonction `xlsx::write.xlsx()` de l'extension `{xlsx}`. L'intérêt de `{openxlsx}` est de ne pas dépendre de **Java** à la différence de `{xlsx}`.

partie III

Analyses

15 Graphiques avec ggplot2

Le package `{ggplot2}` fait partie intégrante du *tidyverse*. Développé par Hadley Wickham, ce package met en œuvre la grammaire graphique théorisée par Leland Wilkinson. Il devient vite indispensable lorsque l'on souhaite réaliser des graphiques un peu complexe.

15.1 Ressources

Il existe de très nombreuses ressources traitant de `{ggplot2}`.

Pour une introduction en français, on pourra se référer au chapitre [Visualiser avec ggplot2](#) de l'*Introduction à R et au tidyverse* de Julien Barnier, au chapitre [Introduction à ggplot2, la grammaire des graphiques](#) du site *analyse-R* et adapté d'une séance de cours de François Briatte, ou encore au chapitre [Graphiques](#) du cours *Logiciel R et programmation* d'Ewen Gallic.

Pour les anglophones, la référence reste encore l'ouvrage *ggplot2: Elegant Graphics for Data Analysis* d'Hadley Wickham lui-même, dont la troisième édition est librement accessible en ligne (<https://ggplot2-book.org/>). D'un point de vue pratique, l'ouvrage *R Graphics Cookbook: practical recipes for visualizing data* de Winston Chang est une mine d'informations, ouvrage là encore librement accessible en ligne (<https://r-graphics.org/>).

15.2 Les bases de ggplot2

`{ggplot2}` nécessite que les données du graphique soient sous la forme d'un tableau de données (*data.frame* ou *tibble*) au format

tidy, c'est-à-dire avec une ligne par observation et les différentes valeurs à représenter sous forme de variables du tableau.

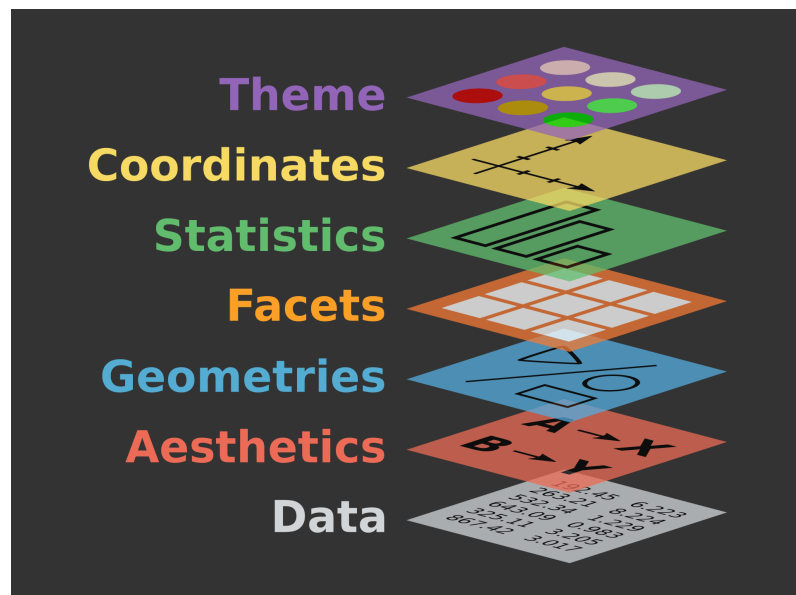


Figure 15.1: La grammaire des graphiques

Tous les graphiques avec `{ggplot2}` suivent une même logique. En **premier** lieu, on appellera la fonction `ggplot2::ggplot()` en lui passant en paramètre le fichier de données.

`{ggplot2}` nomme *esthétiques* les différentes propriétés visuelles d'un graphique, à savoir l'axe des x (`x`), celui des y (`y`), la couleur des lignes (`colour`), celle de remplissage des polygones (`fill`), le type de lignes (`linetype`), la forme des points (`shape`), etc. Une représentation graphique consiste donc à représenter chacune de nos variables d'intérêt selon une esthétique donnée. En **second** lieu, on appellera donc la fonction `ggplot2::aes()` pour indiquer la correspondance entre les variables de notre fichier de données et les esthétiques du graphique.

A minima, il est nécessaire d'indiquer en **troisième** lieu une *géométrie*, autrement dit la manière dont les éléments seront représentés visuellement. À chaque géométrie correspond une fonction commençant par `geom_`, par exemple `ggplot2::geom_point()` pour dessiner des points,

`ggplot2::geom_line()` pour des lignes, `ggplot2::geom_bar()` pour des barres ou encore `ggplot2::geom_area()` pour des aires. Il existe de nombreuses géométries différentes²⁰, chacune prenant en compte certaines esthétiques, certaines étant requises pour cette géométrie et d'autres optionnelles. La liste des esthétiques prises en compte par chaque géométrie est indiquée dans l'aide en ligne de cette dernière.

²⁰ On trouvera une liste dans la *cheat sheet* de `{ggplot2}`, voir Section 15.3.

Voici un exemple minimal de graphique avec `{ggplot2}` :

```
library(ggplot2)
p <-
  ggplot(iris) +
  aes(
    x = Petal.Length,
    y = Petal.Width,
    colour = Species
  ) +
  geom_point()
p
```

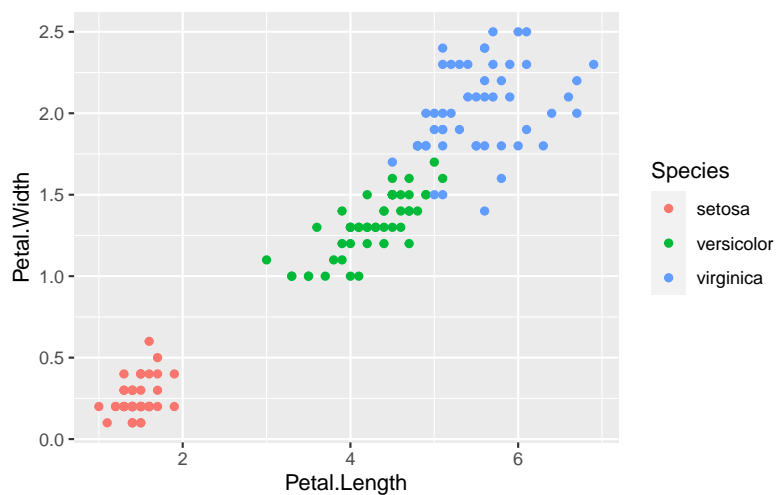


Figure 15.2: Un exemple simple de nuage de points avec `ggplot2`

! Syntaxe additive

Le développement de `{ggplot2}` a débuté avant celui du *tidyverse* et la généralisation du *pipe*. Dès lors, on ne sera pas étonné que la syntaxe de `{ggplot2}` n'ait pas recours à ce dernier mais repose sur une approche *additive*. Un graphique est dès lors initialisé avec la fonction `ggplot2::ggplot()` et l'on ajoutera successivement des éléments au graphique en appelant différentes fonctions et en utilisant l'opérateur `+`.

Il est ensuite possible de personnaliser de nombreux éléments d'un graphique et notamment :

- les *étiquettes* ou *labs* (titre, axes, légendes) avec `ggplot2::ggtitle()`, `ggplot2::xlab()`, `ggplot2::ylab()` ou encore la fonction plus générique `ggplot2::labs()` ;
- les *échelles* (*scales*) des différentes esthétiques avec les fonctions commençant par `scale_` ;
- le système de *coordonnées* avec les fonctions commençant par `coord_` ;
- les *facettes* (*facets*) avec les fonctions commençant par `facet_` ;
- la *légende* (*guides*) avec les fonctions commençant par `guide_` ;
- le *thème* du graphiques (mise en forme des différents éléments) avec `ggplot2::theme()`.

```
p +
  labs(
    x = "Longueur du pétale",
    y = "Largeur du pétale",
    colour = "Espèce"
  ) +
  ggtitle(
    "Relation entre longueur et largeur des pétales",
    subtitle = "Jeu de données Iris"
  ) +
  scale_x_continuous(breaks = 1:7) +
  scale_y_continuous(
```

```

    labels = scales::label_number(decimal.mark = ",")
  ) +
  coord_equal() +
  facet_grid(cols = vars(Species)) +
  guides(
    color = guide_legend(nrow = 2)
  ) +
  theme_light() +
  theme(
    legend.position = "bottom",
    axis.title = element_text(face = "bold")
  )
)

```

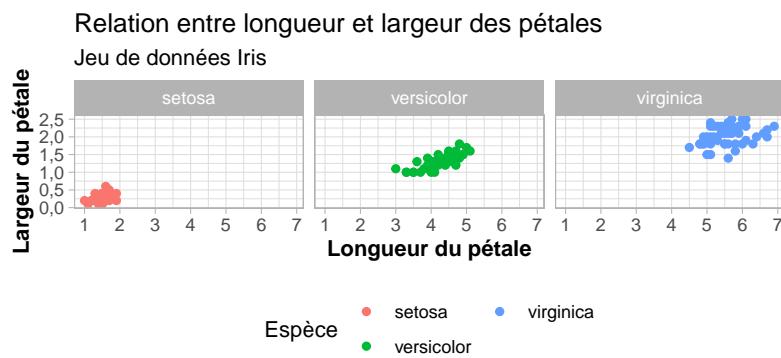


Figure 15.3: Un exemple avancé de nuage de points avec ggplot2



Figure 15.4: Cheatsheet ggplot2

15.3 Cheatsheet

15.4 Exploration visuelle avec esquisse

Le package `{esquisse}` propose un *addin* offrant une interface visuelle pour la création de graphiques `{ggplot2}`. Après installation du package, on pourra lancer `{esquisse}` directement à partir du menu *addins* de **RStudio**.

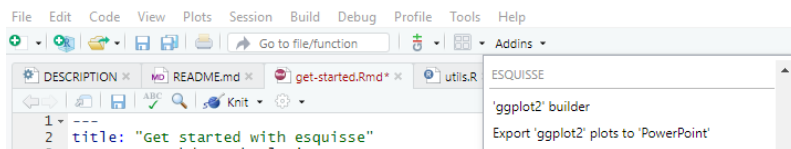


Figure 15.5: Lancement d'*esquisse* à partir du menu *Addins* de **RStudio**

Au lancement de l'*addin*, une interface permettra de choisir le tableau de données à partir duquel générer le graphique. Le plus simple est de choisir un tableau présent dans l'environnement. Mais `{esquisse}` offre aussi la possibilité d'importer des fichiers externes, voir de procéder à quelques modifications des données.

Le principe général d'`{esquisse}` consiste à associer des variables à des esthétiques par glisser/déposer²¹. L'outil déterminera automatiquement une géométrie adaptée en fonction de la nature des variables (continues ou catégorielles). Un clic sur le nom de la géométrie en haut à gauche permet de sélectionner une autre géométrie.

Les menus situés en bas de l'écran permettent d'ajouter/modifier des étiquettes, de modifier certaines options du graphique, de

²¹ Si une esthétique n'est pas visible à l'écran, on pourra cliquer en haut à droite sur l'icône en forme de roue dentée afin de choisir d'afficher plus d'esthétiques.

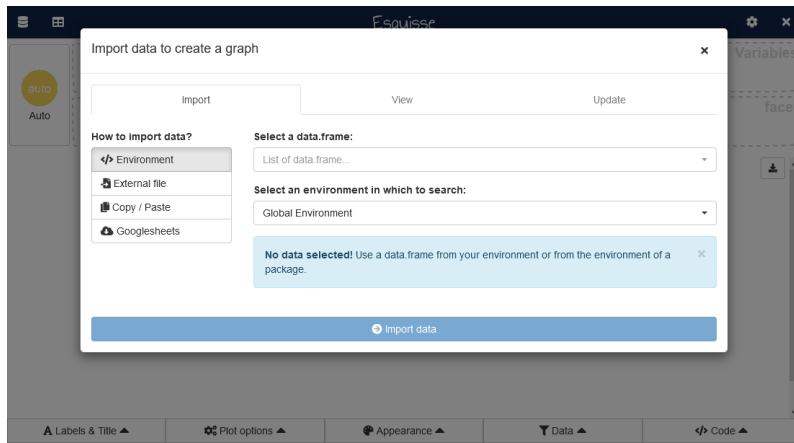


Figure 15.6: Import de données au lancement d'esquisse

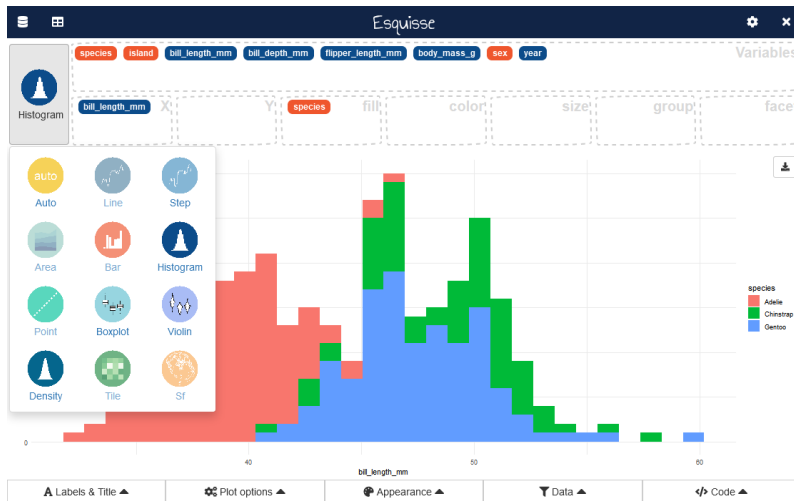


Figure 15.7: Choix d'une géométrie dans esquisse

modifier les échelles de couleurs et l'apparence du graphique, et de filtrer les observations incluses dans le graphique.

Le menu **Code** permet de récupérer le code correspondant au graphique afin de pouvoir le copier/coller dans un script.

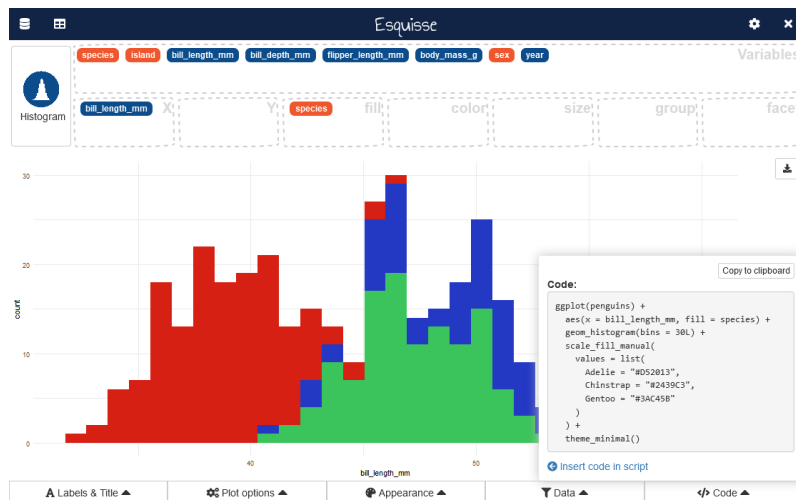


Figure 15.8: Obtenir le code du graphique obtenu avec **esquisse**

`{esquisse}` offre également la possibilité d'exporter le graphique obtenu dans différents formats.

15.5 webin-R

L'utilisation d'`{esquisse}` est présentée dans le webin-R #03 (*statistiques descriptives avec gtsummary et esquisse*) sur [YouTube](https://youtu.be/oEF_8GXyP5c).

https://youtu.be/oEF_8GXyP5c

`{ggplot2}` est abordé plus en détails dans le webin-R #08 (*ggplot2 et la grammaire des graphiques*) sur [YouTube](https://youtu.be/msnwENny_cg).

https://youtu.be/msnwENny_cg

15.6 Combiner plusieurs graphiques

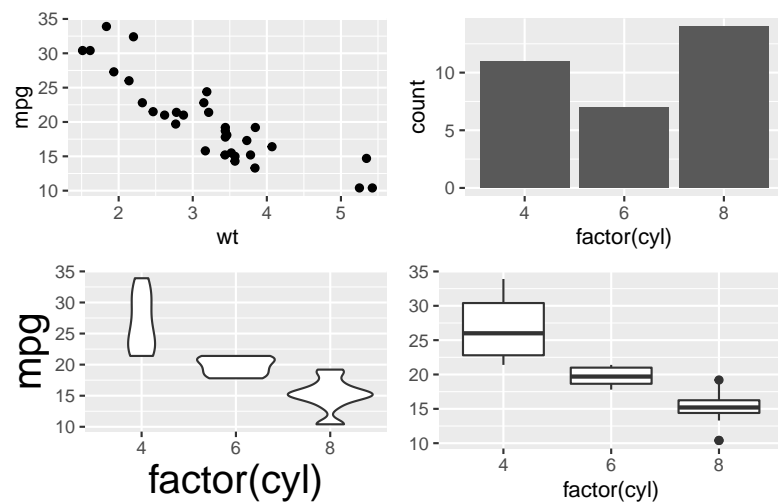
Plusieurs packages proposent des fonctions pour combiner ensemble des graphiques `{ggplot2}`, comme `{patchwork}`, `{ggpubr}` ou `{egg}`. Ici, nous privilégierons la fonction `cowplot::plot_grid()` du package `{cowplot}` car elle permet notamment d'aligner les graphiques les uns par rapports aux autres.

Commençons par créer quelques graphiques avec `{ggplot2}`.

```
p1 <- ggplot(mtcars) +  
  aes(x = wt, y = mpg) +  
  geom_point()  
p2 <- ggplot(mtcars) +  
  aes(x = factor(cyl)) +  
  geom_bar()  
p3 <- ggplot(mtcars) +  
  aes(x = factor(cyl), y = mpg) +  
  geom_violin() +  
  theme(axis.title = element_text(size = 20))  
p4 <- ggplot(mtcars) +  
  aes(x = factor(cyl), y = mpg) +  
  geom_boxplot() +  
  ylab(NULL)
```

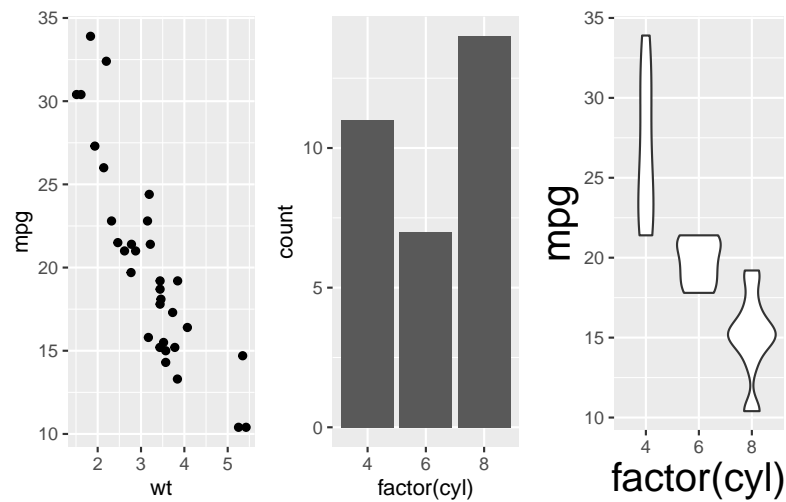
La fonction `cowplot::plot_grid()` permet de combiner ces différents graphiques entre eux.

```
library(cowplot)  
plot_grid(p1, p2, p3, p4)
```



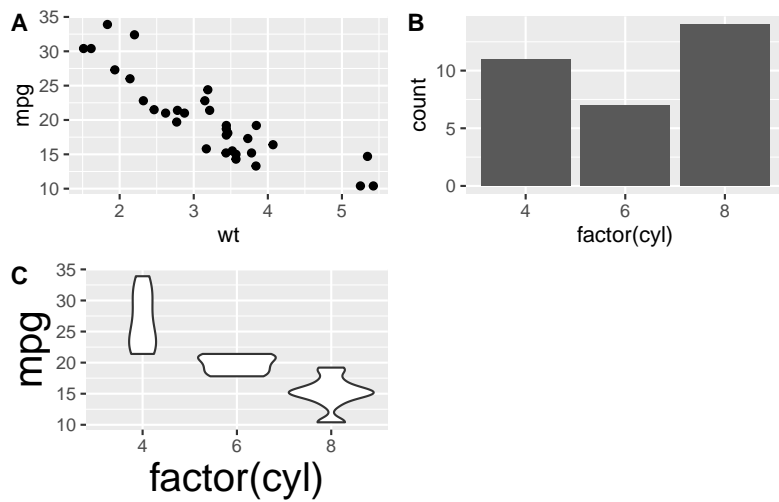
Le nombre de lignes / colonnes peut être personnalisé avec les paramètres `nrow` et `ncol`.

```
plot_grid(p1, p2, p3, nrow = 1)
```



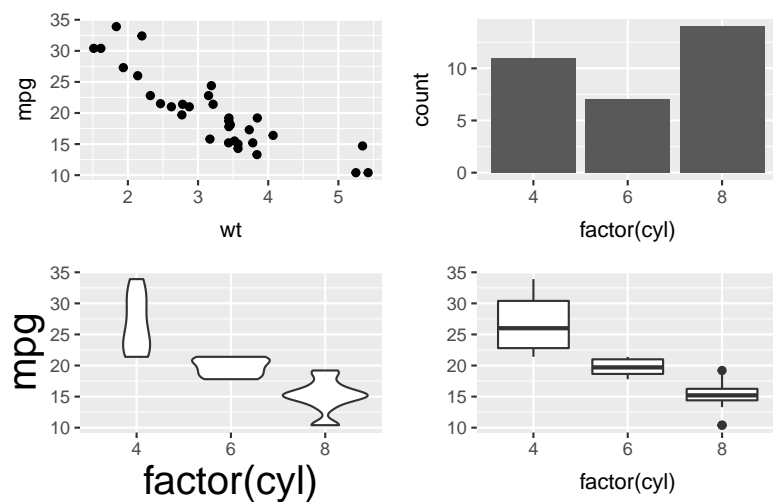
On peut également ajouter des étiquettes avec l'argument `labels`.

```
plot_grid(p1, p2, p3, labels = "AUTO", label_size = 12)
```



Dans certain cas, on peut avoir besoin d'aligner les axes des graphiques, ce qui s'obtient avec l'argument `align`.

```
plot_grid(p1, p2, p3, p4, align = "hv")
```

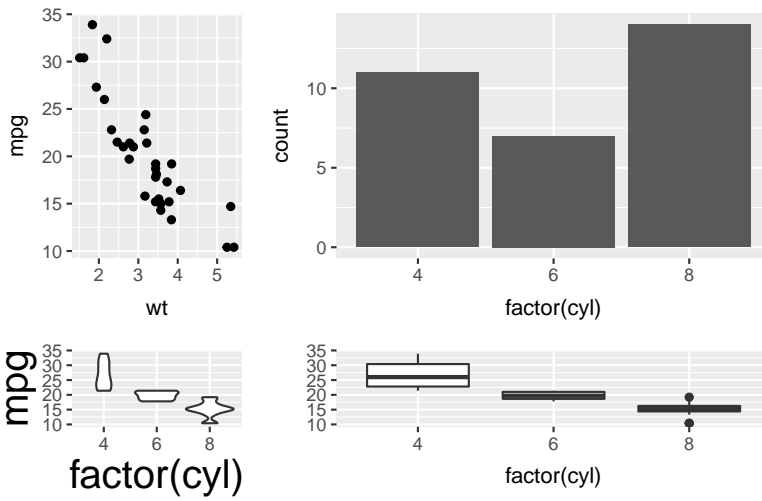


Enfin, on peut jouer les hauteurs/largeurs relatives des lignes/colonnes avec `rel_heights` et `rel_widths`.

```

plot_grid(
  p1, p2, p3, p4,
  align='hv',
  rel_heights=c(2,1),
  rel_widths=c(1,2)
)

```



partie IV

Manipulation avancée

16 Dates avec lubridate

17 Réorganisation avec tidyr