guide-R

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

Joseph Larmarange

22 septembre 2022

Table des matières

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.

Ce guide porte sur l'analyse de données d'enquêtes avec le logiciel R, un logiciel libre de statitistiques 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'autre 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 oppose[1] 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

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 eté 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 pacakges 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 suffisament 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 fuat 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 permets 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 décimaux[3]), par exemple `5.23` ;
- les nombres entiers, que l'on saisi en ajoutant le suffixe `L`[4], 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 majuscules[5]).

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 12).

[3] Pour rappel, **R** étant anglophone, le caractère utilisé pour indiqué 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 1er 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 combiner[6].

```
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.

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

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

```r
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 :

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

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

```r
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 :

```r
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 :

```r
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 :

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

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

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

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

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

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

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

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

```r
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 :

```r
1:5
```

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

```r
24:32
```

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

```r
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()` :

```r
length(taille)
```

```
[1] 6
```

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

```
[1] 2
```

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

```r
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 crochets[7] (`[]`) 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"
```

```r
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…

```r
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 :

```r
sexe
```

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

```r
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
`==`	égal à
`%in%`	appartient à
`!=`	différent de
`>`	strictement supérieur à
`<`	strictement inférieur à
`>=`	supérieur ou égal à
`<=`	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

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" ...
```

> **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.

```
l
```

```
[[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"
```

```
l[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 pourrons 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
```

```r
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()` :

```r
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.

```r
df[1]
```

```
         sexe
Anna        f
Mary-Ann    f
Michael     h
John        h
```

```r
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")`.

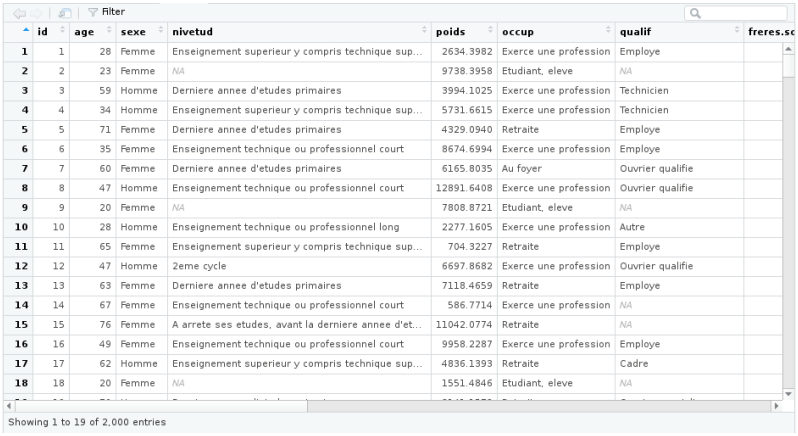
4.3 Afficher les données

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)
```



Filter

	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
```

```r
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.

```r
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, O~
$ 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, O~
$ 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, 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, O~
$ 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, O~
$ 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 13) 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)
```

```
-- 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()
```

```
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) !

```r
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 fonctionalité, 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 differents 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*, différent 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.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**setosa**			**versicolor**			**virginica**		
	Beta	**95% CI**	**p-value**	**Beta**	**95% CI**	**p-value**	**Beta**	**95% CI**	**p-value**
Petal.Length	0.40	-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 nottament :

- `class` : la classe de l'objet
- `lenghth` : 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 spéficiant 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'enchainer 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 entraine 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'occurence 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 effecture `library(tidyverse)`.

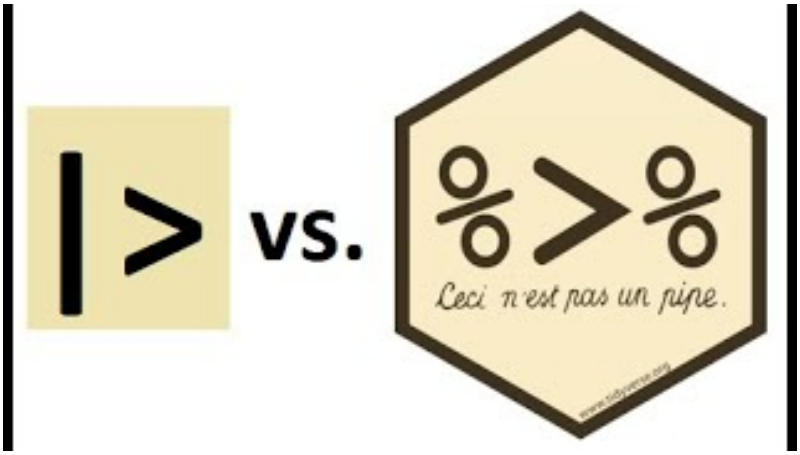
Cet opérateur s'écrit `%>%` et il dispose lui aussi d'un *placeholder* qui est le `.`. La syntaxe du *placeholder* est un peu plus souple puisqu'il peut être passé à tout type d'argument, y compris un argument sans nom. Si l'on reprend notre exemple précédent.

```
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.Witdh")` est équivalent à `iris$Petal.Width`. Voyons un example 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'élement 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 données[10] (*data frame* ou *tibble*) et renvoient systématiquement un *tibble*.

[10] Le package `{dbplyr}` permets 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}`.

```
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)
```

```
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 renvoit `TRUE` (vrai) sont conservées[11].

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

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

```
filter(flights, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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   Chefornak 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 TUL   Tulsa Intl              36.2  -95.9   677    -6 A     America/Chicago
2 PIT   Pittsburgh Intl         40.5  -80.2  1204    -5 A     America/New_York
3 IAG   Niagara Falls Intl      43.1  -78.9   589    -5 A     America/New_York
4 MTM   Metlakatla Seaplane Base 55.1 -132.      0    -9 A     America/Anchora~
5 KVL   Kivalina Airport        67.7 -165.     13    -9 A     America/Anchora~
```

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     9    15     1823       1829      -6    1916    1939     -23 MQ
 2  2013    12    31     1931       1932      -1      28      28       0 B6
 3  2013     2    23     1915       1900      15    2149    2133      16 FL
 4  2013     9    17     1334       1325       9    1519    1528      -9 EV
 5  2013    10    28     2025       2029      -4    2330    2337      -7 UA
 6  2013     3    28     1015       1020      -5    1304    1330     -26 AA
 7  2013     1    29       NA       1650      NA      NA    1826      NA UA
 8  2013    12    23     1932       1936      -4    2250    2244       6 B6
 9  2013     1     9     1507       1505       2    1746    1734      12 UA
10  2013     7     9     1545       1535      10    1918    1901      17 DL
# ... 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 <dttm>, 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 <dttm>, 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 Airport   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_width()`, `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* incluses[12] :

```
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()` permets 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* :

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

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

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

```
   <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 0S9   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()` permets 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 variables[14].

[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.

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()
```

```
[1] 1001.416
```

> **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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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 <dttm>, 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
```

```r
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 <dttm>, 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 du coup 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())
```

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

[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.

```
# 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, renvoit un tableau non groupé :

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

```
# A tibble: 1,113 x 3
   month dest      n
   <int> <chr> <int>
 1     1 ALB      64
 2     1 ATL    1396
 3     1 AUS     169
 4     1 AVL       2
 5     1 BDL      37
 6     1 BHM      25
 7     1 BNA     399
 8     1 BOS    1245
 9     1 BQN      93
10     1 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 fecteurs 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 permets pas d'indiquer un ordre spécifique aux modalités, à la différence des facteurs.

> **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 abordés dans un prochain chapitre (voir Chapitre 11).
> 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 fecteur 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 siliencieusement 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 ordinale), on peut avoir avoir besoin d'indiquer que les modalités du facteur sont ordonnées héarchiquement. 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 nombre 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 pacakge `{forcats}`, chargé par défaut lorsque l'on exécute la commande `library(tidyverse)`, fournie 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 inverser 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)
```

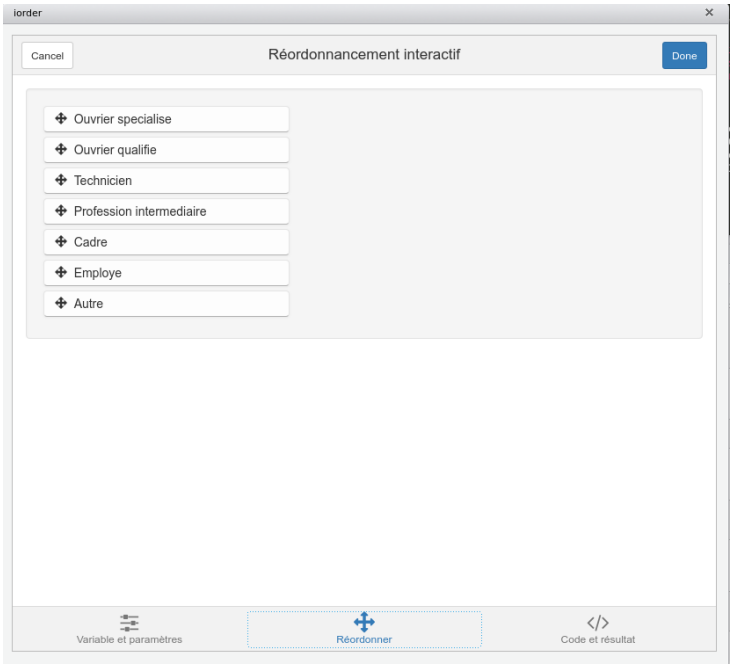
```
[1] c a d b a c
Levels: c a d b
```

La fonction `forcats::fct_reorder()` permets 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) :

```r
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
```

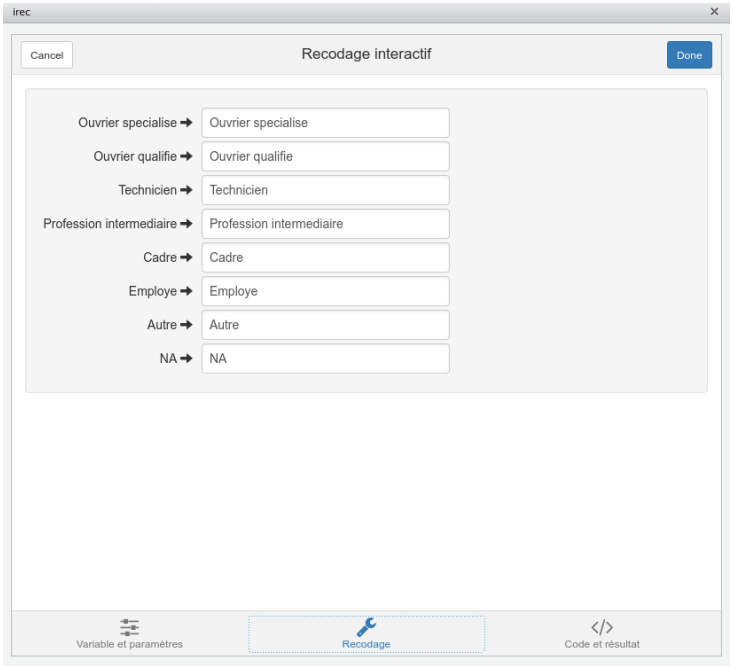
```
hdv2003$instruction <-
  hdv2003$nivetud |>
  fct_recode(
    "primaire" = "N'a jamais fait d'etudes",
    "primaire" = "A arrete ses etudes, avant la derniere annee d'etudes primaires",
    "primaire" = "Derniere annee d'etudes primaires",
    "secondaire" = "1er cycle",
    "secondaire" = "2eme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



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()` permets 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 catgéorielles (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 inclues 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` :

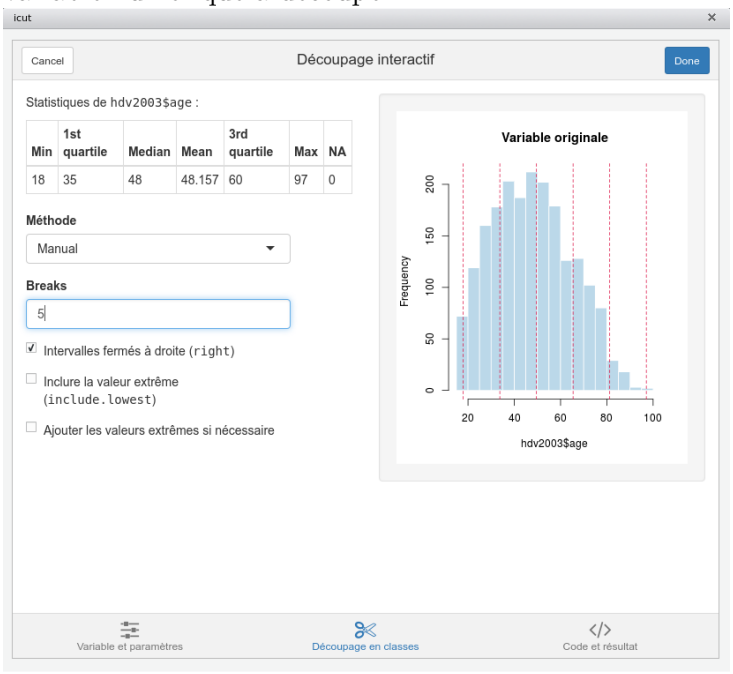
```
hdv2003 <-
  hdv2003 |>
  mutate(groupe_ages = cut(
    age,
    c(18, 20, 40, 60, 80, 97),
    include.lowest = TRUE,
    right = FALSE
  ))
hdv2003$groupe_ages |> questionr::freq()
```

```
          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.



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=2795). https://youtu.be/CokvTbtWdwc

9.5

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)
```

```
-- 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()
```

```
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 :

```r
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 :

```r
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()`. Suppons 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.

```r
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
```

```r
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 renvoit 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 fournie 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 valeurs

partie III

Manipulation avancée

12 Dates avec `lubridate`

13 Réorganisation avec `tidyr`