guide-R

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

Joseph Larmarange

17 septembre 2022

Table des matières

Préface **4**
Remerciements 6
Licence 6

I Bases du langage **7**

1 Packages **8**
1.1 Installation (CRAN) 9
1.2 Chargement 9
1.3 Mise à jour 10
1.4 Installation depuis GitHub 11
1.5 Le tidyverse 12

2 Vecteurs **15**
2.1 Types et classes 15
2.2 Création d'un vecteur 16
2.3 Longueur d'un vecteur 19
2.4 Combiner des vecteurs 20
2.5 Vecteurs nommés 20
2.6 Indexation par position 22
2.7 Indexation par nom 23
2.8 Indexation par condition 24
2.9 Assignation par indexation 28
2.10 En résumé 29
2.11 webin-R 30

3 Listes **31**
3.1 Propriétés et création 31
3.2 Indexation 34
3.3 En résumé 38
3.4 webin-R 39

4 Tableaux de données **40**
4.1 Propriétés et création 40
4.2 Indexation 42
4.3 Afficher les données 46
4.4 En résumé 53
4.5 webin-R 54

5 Tibbles **55**
5.1 Le concept de tidy data 55
5.2 tibbles : des tableaux de données améliorés . . . 55
5.3 Données et tableaux imbriqués 60

6 Attributs **63**

II Manipulation de données **66**

7 Le pipe **67**
7.1 Le pipe natif de R : `|>` 68
7.2 Le pipe du tidyverse : `%>%` 69
7.3 Vaut-il mieux utiliser `|>` ou `%>%` ? 70
7.4 Accéder à un élément avec `purrr::pluck()` et `purrr::chuck()` 70

8 Facteurs avec `forcats` **73**

III Manipulation avancée **74**

9 Dates avec `lubridate` **75**

10 Réorganisation avec `tidyr` **76**

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 8) ou les dates (voir Chapitre 9).

[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].

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

```
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
```

```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")`.

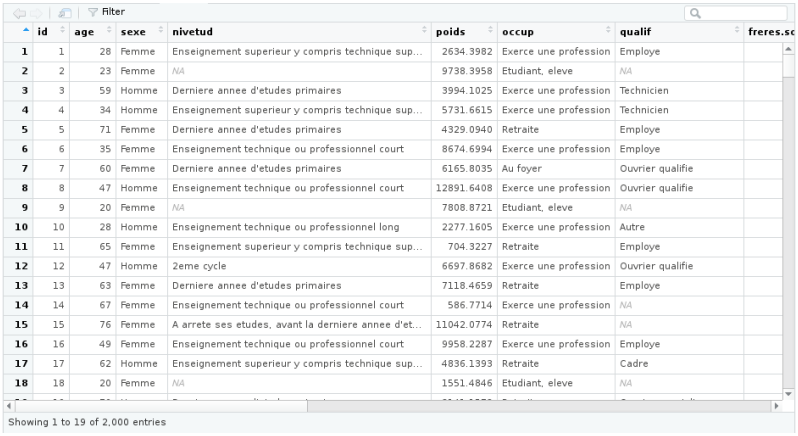
4.3 Afficher les données

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

Showing 1 to 19 of 2,000 entries

Figure 4.1: Interface View() de R RStudio

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

```
head(hdv2003)
```

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

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

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

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

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

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

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

```
$ id            <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
$ age           <int> 28, 23, 59, 34, 71, 35, 60, 47, 20, 28, 65, 47, 63, 67, ~
$ sexe          <fct> Femme, Femme, Homme, Homme, Femme, Femme, Femme, Homme, ~
$ nivetud       <fct> "Enseignement superieur y compris technique superieur", ~
$ poids         <dbl> 2634.3982, 9738.3958, 3994.1025, 5731.6615, 4329.0940, 8~
$ occup         <fct> "Exerce une profession", "Etudiant, eleve", "Exerce une ~
$ qualif        <fct> Employe, NA, Technicien, Technicien, Employe, Employe, 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 10) 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

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

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

[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.

```
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)`.

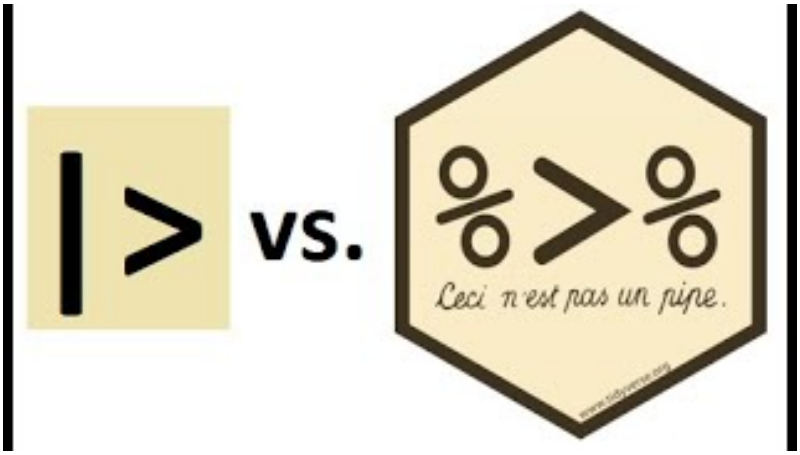
Cet opérateur s'écrit `%>%` et il dispose lui aussi d'un *placeholder* qui est le `.`. La syntaxe du *placeholder* est un peu plus souple puisqu'il peut être passé à tout type d'argument, y compris un argument sans nom. Si l'on reprend notre exemple précédent.

```
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 Facteurs avec `forcats`

partie III

Manipulation avancée

9 Dates avec `lubridate`

10 Réorganisation avec `tidyr`