

Introduction à R et au tidyverse

Julien Barnier

2022-01-26

Table des Matières

À propos de ce document	7
Remerciements	7
Licence	7
1 Présentation	9
1.1 À propos de R	9
1.2 À propos de RStudio	10
1.3 À propos du <i>tidyverse</i>	10
1.4 Structure du document	10
1.5 Prérequis	10
I Introduction à R	11
2 Prise en main	13
2.1 La console	13
2.2 Objets	15
2.3 Fonctions	18
2.4 Regrouper ses commandes dans des scripts	21
2.5 Installer et charger des extensions (<i>packages</i>)	22
2.6 Exercices	23
3 Premier travail avec des données	27
3.1 Jeu de données d'exemple	27
3.2 Tableau de données (<i>data frame</i>)	28
3.3 Analyse univariée	32
3.4 Exercices	43
4 Analyse bivariée	45
4.1 Croisement de deux variables qualitatives	45
4.2 Croisement d'une variable quantitative et d'une variable qualitative	49
4.3 Croisement de deux variables quantitatives	54
4.4 Exercices	61

5 Organiser ses scripts	63
5.1 Les projets dans RStudio	63
5.2 Créer des sections dans un script	64
5.3 Répartir son travail entre plusieurs scripts	65
5.4 Désactiver la sauvegarde de l'espace de travail	67
II Introduction au tidyverse	69
6 Le tidyverse	71
6.1 Extensions	71
6.2 Installation	71
6.3 tidy data	72
6.4 tibbles	73
7 Importer et exporter des données	75
7.1 Import de fichiers textes	75
7.2 Import depuis un fichier Excel	78
7.3 Import de fichiers SAS, SPSS et Stata	80
7.4 Import de fichiers dBase	80
7.5 Connexion à des bases de données	80
7.6 Export de données	83
8 Visualiser avec ggplot2	85
8.1 Préparation	85
8.2 Initialisation	85
8.3 Exemples de geom	89
8.4 Mappages	110
8.5 Représentation de plusieurs geom	121
8.6 Faceting	128
8.7 Scales	132
8.8 Thèmes	152
8.9 L'add-in esquisse	155
8.10 Ressources	159
8.11 Exercices	159
9 Recoder des variables	169
9.1 Rappel sur les variables et les vecteurs	169
9.2 Tests et comparaison	170
9.3 Recoder une variable qualitative	173
9.4 Combiner plusieurs variables	181
9.5 Découper une variable numérique en classes	183
9.6 Exercices	185

10 Manipuler les données avec dplyr	191
10.1 Préparation	191
10.2 Les verbes de dplyr	192
10.3 Enchaîner les opérations avec le <i>pipe</i>	199
10.4 Opérations groupées	201
10.5 Autres fonctions utiles	210
10.6 Tables multiples	215
10.7 Ressources	224
10.8 Exercices	224
11 Manipuler du texte avec stringr	235
11.1 Expressions régulières	235
11.2 Concaténer des chaînes	236
11.3 Convertir en majuscules / minuscules	237
11.4 Découper des chaînes	237
11.5 Extraire des sous-chaînes par position	238
11.6 Déetecter des motifs	238
11.7 Extraire des motifs	239
11.8 Remplacer des motifs	240
11.9 Modificateurs de motifs	240
11.10 Ressources	241
11.11 Exercices	241
12 Mettre en ordre avec tidyr	243
12.1 Tidy data	243
12.2 Trois règles pour des données bien rangées	244
12.3 Les verbes de tidyr	246
12.4 Ressources	253
13 Diffuser et publier avec rmarkdown	255
13.1 Créer un nouveau document	258
13.2 Éléments d'un document R Markdown	259
13.3 Personnaliser le document généré	263
13.4 Options des blocs de code R	265
13.5 Rendu des tableaux	268
13.6 Modèles de documents	270
13.7 Ressources	273

III Aller plus loin	275
14 Écrire ses propres fonctions	277
14.1 Introduction et exemples	277
14.2 Arguments et résultat d'une fonction	286
14.3 Portée des variables	291
14.4 Les fonctions comme objets	294
14.5 Ressources	296
14.6 Exercices	297
15 dplyr avancé	303
15.1 Appliquer ses propres fonctions	303
15.2 <code>across()</code> : appliquer des fonctions à plusieurs colonnes	308
15.3 Fonctions anonymes et syntaxes abrégées	315
15.4 <code>rowwise()</code> et <code>c_across()</code> : appliquer une transformation ligne par ligne	317
15.5 Ressources	321
15.6 Exercices	321
16 Structures de données	327
16.1 Vecteurs atomiques	327
16.2 Listes	332
16.3 Tableaux de données (<i>data frame</i> et <i>tibble</i>)	338
16.4 Ressources	341
16.5 Exercices	342
17 Exécution conditionnelle et boucles	347
17.1 <code>if</code> et <code>else</code> : exécuter du code sous certaines conditions	347
17.2 Contrôle de l'exécution et gestion des erreurs	354
17.3 <code>for</code> et <code>while</code> : répéter des instructions dans une boucle	358
17.4 Ressources	364
17.5 Exercices	364
18 Itérer avec purrr	367
18.1 Exemple d'application	367
18.2 <code>map</code> et ses variantes	369
18.3 Itérer sur les colonnes d'un tableau de données	374
18.4 <code>modify</code>	376
18.5 <code>imap</code>	377
18.6 <code>walk</code>	379
18.7 <code>map2</code> et <code>pmap</code> : itérer sur plusieurs vecteurs en parallèle	380
18.8 Répéter une opération	384
18.9 Quand (ne pas) utiliser <code>map</code>	384
18.10 <code>purrr</code> vs <code>*apply</code>	386
18.11 Ressources	386
18.12 Exercices	386

19 Programmer avec le <i>tidyverse</i>	393
19.1 Spécificités du <i>tidyverse</i>	393
19.2 Programmer avec <i>dplyr</i> et <i>tidyr</i>	398
19.3 Programmer avec <i>ggplot2</i>	404
19.4 Aide-mémoire	409
19.5 Ressources	411
19.6 Exercices	411
20 Déboggage et performance	421
20.1 Débugguer une fonction	421
20.2 benchmarking : mesurer et comparer les temps d'exécution	427
20.3 Quelques conseils d'optimisation	430
20.4 Ressources	434
21 Organiser un projet avec <i>targets</i>	435
21.1 Définition du pipeline	435
21.2 Exécution du pipeline	439
21.3 Modification du pipeline	440
21.4 RMarkdown	441
21.5 Gestion des données en cache	443
21.6 Avantages et limites	444
21.7 Ressources	445
Appendix	445
A Ressources	447
A.1 Aide	447
A.2 Ouvrages, blogs, MOOCs...	450
A.3 Extensions	450

À propos de ce document

Ce document est une introduction à **R**, logiciel libre de traitement et d'analyse de données. Il se veut le plus accessible possible, y compris pour ceux qui ne sont pas particulièrement familiers avec l'informatique. Il se base à la fois sur les fonctionnalités de R “de base”, et sur une série d'extensions regroupées sous l'appellation *tidyverse*.

Ce document *n'est pas* une introduction aux méthodes statistiques d'analyse de données.

Il est basé sur R version 4.1.2 (2021-11-01).

Ce document est régulièrement corrigé et mis à jour. La version de référence est disponible en ligne à l'adresse :

- <https://juba.github.io/tidyverse>

Il est généré par l'excellente extension **bookdown** de [Yihui Xie](#), et le code source est disponible [sur GitHub](#).

Pour toute suggestion ou correction, il est possible de me contacter [par mail](#) ou [sur Twitter](#).

Remerciements

Un remerciement tout particulier à mes formidables collègues Sofiane Bouzid, Behnaz Khosravi et Karine Pietropaoli pour leurs nombreux et utiles retours sur la partie *Aller plus loin*.

Ce document a également bénéficié de la relecture, des suggestions et des corrections de Diane Rodet, Mayeul Kauffmann, Jimmy Raturat, Fabienne Marquant, Julien Biaudet, Frédérique Giraud, Joël Gombin, Milan Bouchet-Valat et Joseph Larmarange.

Licence

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



Figure 1: Licence Creative Commons

Chapitre 1

Présentation

1.1 À propos de R

R est un langage orienté vers le traitement et l'analyse quantitative de données, dérivé du langage S. Il est développé depuis les années 90 par un groupe de volontaires de différents pays et par une large communauté d'utilisateurs et utilisatrices. C'est un logiciel libre, publié sous [licence GNU GPL](#).

L'utilisation de R présente plusieurs avantages :

- c'est un logiciel multiplateforme, qui fonctionne aussi bien sur des systèmes Linux, Mac OS X ou Windows.
- c'est un logiciel libre, développé par ses utilisateurs et utilisatrices, diffusable et modifiable par tout un chacun.
- c'est un logiciel gratuit.
- c'est un logiciel puissant, dont les fonctionnalités de base peuvent être étendues à l'aide d'extensions développées par la communauté. Il en existe plusieurs milliers.
- c'est un logiciel avec d'excellentes capacités graphiques.

Comme rien n'est parfait, on peut également trouver quelques inconvénients :

- le logiciel, la documentation de référence et les principales ressources sont en anglais. Il est toutefois parfaitement possible d'utiliser R sans spécialement maîtriser cette langue et il existe de plus en plus de ressources francophones.
- R n'est pas un logiciel au sens classique du terme, mais plutôt un langage de programmation. Il fonctionne à l'aide de scripts (des petits programmes) édités et exécutés au fur et à mesure de l'analyse.
- en tant que langage de programmation, R a la réputation d'être difficile d'accès, notamment pour ceux n'ayant jamais programmé auparavant.

Ce document ne demande aucun prérequis en informatique ou en programmation. Juste un peu de motivation pour l'apprentissage du langage et, si possible, des données intéressantes sur lesquelles appliquer les connaissances acquises.

L'aspect langage de programmation et la difficulté qui en découle peuvent sembler des inconvénients importants. Le fait de structurer ses analyses sous forme de scripts (suite d'instructions effectuant les différentes opérations d'une analyse) présente cependant de nombreux avantages :

- le script conserve l'ensemble des étapes d'une analyse, de l'importation des données à leur analyse en passant par les manipulations et les recodages.
- on peut à tout moment revenir en arrière et corriger ou modifier ce qui a été fait.
- il est très rapide de réexécuter une suite d'opérations complexes.
- on peut très facilement mettre à jour les résultats en cas de modification des données sources.
- le script garantit, sous certaines conditions, la reproductibilité des résultats obtenus.

1.2 À propos de RStudio

RStudio n'est pas à proprement parler une interface graphique pour R, il s'agit plutôt d'un *environnement de développement intégré*, qui propose des outils facilitant l'écriture de scripts et l'usage de R au quotidien. C'est une interface bien supérieure à celles fournies par défaut lorsqu'on installe R sous Windows ou sous Mac¹.

Pour paraphraser [Hadrien Commenges](#), il n'y a pas d'obligation à utiliser RStudio, mais il y a une obligation à ne pas utiliser les interfaces de R par défaut.

RStudio est également un logiciel libre et gratuit. Une version payante existe, mais elle ne propose pas de fonctionnalités indispensables.

1.3 À propos du *tidyverse*

Le *tidyverse* est un ensemble d'extensions pour R (code développé par la communauté permettant de rajouter des fonctionnalités à R) construites autour d'une philosophie commune et conçues pour fonctionner ensemble. Elles facilitent l'utilisation de R dans les domaines les plus courants : manipulation des données, recodages, production de graphiques, etc.

La deuxième partie de ce document est entièrement basée sur les extensions du *tidyverse*, qui est présenté plus en détail chapitre 6.

1.4 Structure du document

Ce document est composé de trois grandes parties :

- Une *Introduction à R*, qui présente les bases du langage R et de l'interface RStudio
- Une *Introduction au tidyverse* qui présente cet ensemble d'extensions pour la visualisation, la manipulation des données et l'export de résultats
- Une partie *Aller plus loin* qui présente comment créer ses propres fonctions et introduit des notions de programmation plus avancées

Les personnes déjà familières avec R "de base" peuvent passer directement à l'*Introduction au tidyverse*.

1.5 Prérequis

Le seul prérequis pour suivre ce document est d'avoir installé R et RStudio sur votre ordinateur. Il s'agit de deux logiciels libres, gratuits, téléchargeables en ligne et fonctionnant sous PC, Mac et Linux.

Pour installer R, il suffit de se rendre sur une des pages suivantes ² :

- [Installer R sous Windows](#)
- [Installer R sous Mac](#)

Pour installer RStudio, rendez-vous sur [la page de téléchargement du logiciel](#) et installez la version adaptée à votre système.

¹Sous Linux R n'est fourni que comme un outil en ligne de commande.

²Sous Linux, utilisez votre gestionnaire de packages habituel.

Partie I

Introduction à R

Chapitre 2

Prise en main

Une fois R et RStudio installés sur votre machine, nous n'allons pas lancer R mais plutôt RStudio.

RStudio n'est pas à proprement parler une interface graphique qui permettrait d'utiliser R de manière "classique" via la souris, des menus et des boîtes de dialogue. Il s'agit plutôt de ce qu'on appelle un *Environnement de développement intégré* (IDE) qui facilite l'utilisation de R et le développement de scripts (voir section 1.2).

2.1 La console

2.1.1 L'invite de commandes

Au premier lancement de RStudio, l'interface est organisée en trois grandes zones.

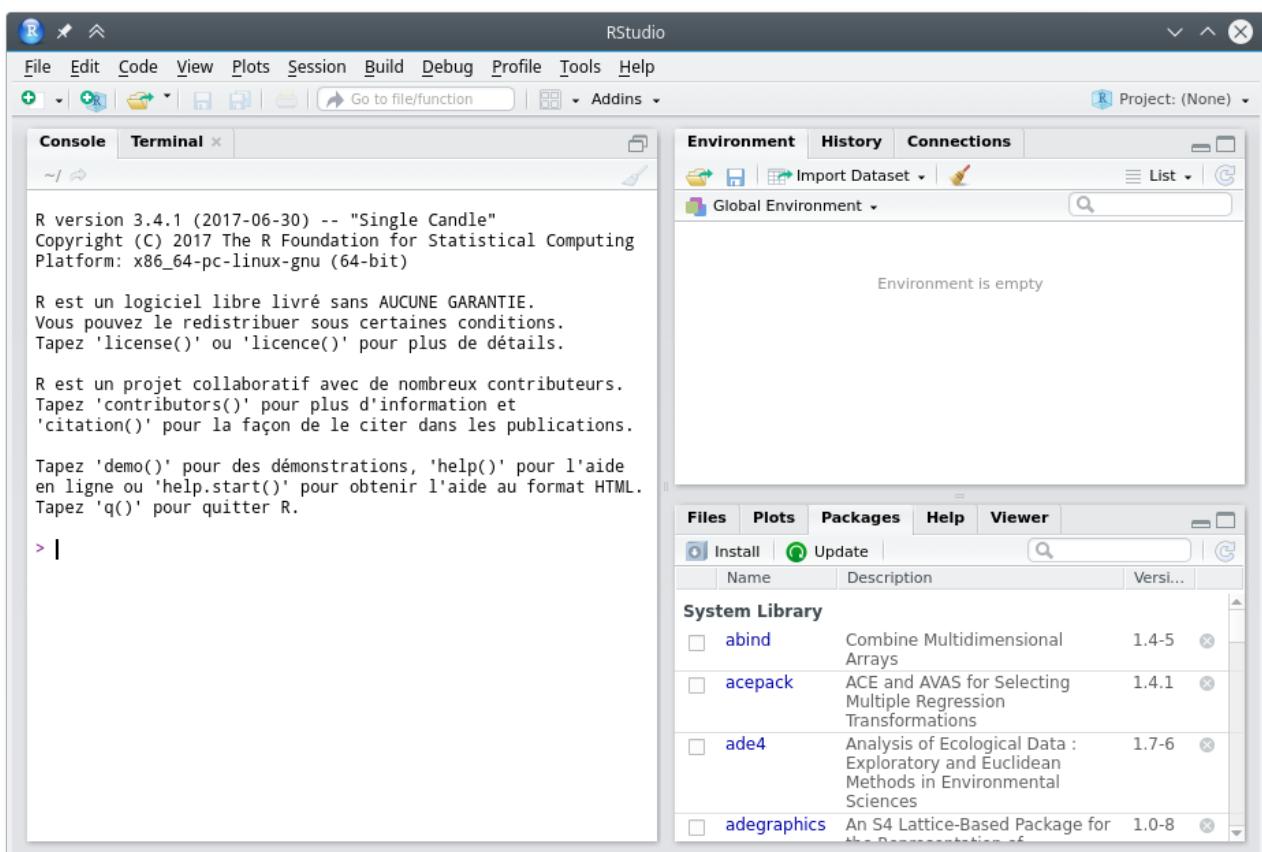


Figure 2.1: Interface de Rstudio

La zone de gauche se nomme la *Console*. À son démarrage, RStudio a lancé une nouvelle session de R et c'est dans cette fenêtre que nous allons pouvoir interagir avec lui.

La *Console* doit normalement afficher un texte de bienvenue ressemblant à ceci :

```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.
```

suivi d'une ligne commençant par le caractère > et sur laquelle devrait se trouver votre curseur. Cette ligne est appelée l'*invite de commande* (ou *prompt* en anglais). Elle signifie que R est disponible et en attente de votre prochaine instruction.

Nous pouvons tout de suite lui fournir une première commande, en saisissant le texte suivant puis en appuyant sur Entrée :

```
2 + 2
#> [1] 4
```

R nous répond immédiatement, et nous pouvons constater avec soulagement qu'il sait faire des additions à un chiffre¹. On peut donc continuer avec d'autres opérations :

```
5 - 7
#> [1] -2
4 * 12
#> [1] 48
-10 / 3
#> [1] -3.333333
5^2
#> [1] 25
```

Cette dernière opération utilise le symbole \wedge qui représente l'opération *puissance*. 5^2 signifie donc “5 au carré”.

2.1.2 Précisions concernant la saisie des commandes

Lorsqu'on saisit une commande, les espaces autour des opérateurs n'ont pas d'importance. Les trois commandes suivantes sont donc équivalentes, mais on privilégie en général la deuxième pour des raisons de lisibilité du code.

¹On peut ignorer pour le moment la présence du [1] en début de ligne.

```
10+2
10 + 2
10      +      2
```

Quand vous êtes dans la console, vous pouvez utiliser les flèches vers le haut ↑ et vers le bas ↓ de votre clavier pour naviguer dans l'historique des commandes que vous avez tapées précédemment. Vous pouvez à tout moment modifier la commande affichée, et l'exécuter en appuyant sur **Entrée**.

Enfin, il peut arriver qu'on saisisse une commande de manière incomplète : oubli d'une parenthèse, faute de frappe, etc. Dans ce cas, R remplace l'invite de commande habituel par un signe +.

```
4 *
+
+
```

Cela signifie qu'il “attend la suite”. On peut alors soit compléter la commande sur cette nouvelle ligne et appuyer sur **Entrée**, soit, si on est perdu, tout annuler et revenir à l'invite de commandes normal en appuyant sur **Esc** ou **Échap**.

2.2 Objets

2.2.1 Objets simples

Faire des calculs c'est bien, mais il serait intéressant de pouvoir stocker un résultat quelque part pour pouvoir le réutiliser ultérieurement sans avoir à faire du copier/coller.

Pour conserver le résultat d'une opération, on peut le stocker dans un *objet* à l'aide de l'opérateur d'assignation <- . Cette “flèche” stocke ce qu'il y a à sa droite dans un objet dont le nom est indiqué à sa gauche.

Prenons tout de suite un exemple.

```
x <- 2
```

Cette commande peut se lire “prend la valeur 2 et mets la dans un objet qui s'appelle x”.

Si on exécute une commande comportant juste le nom d'un objet, R affiche son contenu.

```
x
#> [1] 2
```

On voit donc que notre objet x contient bien la valeur 2.

On peut évidemment réutiliser cet objet dans d'autres opérations : R le remplacera alors par sa valeur.

```
x + 4
#> [1] 6
```

On peut créer autant d'objets qu'on le souhaite.

```
x <- 2
y <- 5
resultat <- x + y
resultat
#> [1] 7
```



Les noms d'objets peuvent contenir des lettres, des chiffres, les symboles `.` et `_`. Ils ne peuvent pas commencer par un chiffre. Attention, R fait la différence entre minuscules et majuscules dans les noms d'objets, ce qui signifie que `x` et `X` seront deux objets différents, tout comme `resultat` et `Resultat`.

De manière générale, il est préférable d'éviter les majuscules (pour les risques d'erreur) et les caractères accentués (pour des questions d'encodage) dans les noms d'objets.

De même, il faut essayer de trouver un équilibre entre clarté du nom (comprendre à quoi sert l'objet, ce qu'il contient) et sa longueur. Par exemple, on préférera comme nom d'objet `taille_conj1` à `taille_du_conjoint_numero_1` (trop long) ou à `t1` (pas assez explicite).

Quand on assigne une nouvelle valeur à un objet déjà existant, la valeur précédente est perdue. Les objets n'ont pas de mémoire.

```
x <- 2
x <- 5
x
#> [1] 5
```

De la même manière, assigner un objet à un autre ne crée pas de “lien” entre les deux. Cela copie juste la valeur de l'objet de droite dans celui de gauche :

```
x <- 1
y <- 3
x <- y
x
#> [1] 3
## Si on modifie y, cela ne modifie pas x
y <- 4
x
#> [1] 3
```

On le verra, les objets peuvent contenir tout un tas d'informations. Jusqu'ici on n'a stocké que des nombres, mais ils peuvent aussi contenir des chaînes de caractères (du texte), qu'on délimite avec des guillemets simples ou doubles (' ou ") :

```
chien <- "Chihuahua"
chien
#> [1] "Chihuahua"
```

2.2.2 Vecteurs

Imaginons maintenant qu'on a demandé la taille en centimètres de 5 personnes et qu'on souhaite calculer leur taille moyenne. On pourrait créer autant d'objets que de tailles et faire l'opération mathématique qui va bien :

```
taille1 <- 156
taille2 <- 164
taille3 <- 197
taille4 <- 147
taille5 <- 173
(taille1 + taille2 + taille3 + taille4 + taille5) / 5
#> [1] 167.4
```

Cette manière de faire n'est clairement pas pratique du tout. On va donc plutôt stocker l'ensemble de nos tailles dans un seul objet, de type *vecteur*, avec la syntaxe suivante :

```
tailles <- c(156, 164, 197, 147, 173)
```

Si on affiche le contenu de cet objet, on voit qu'il contient bien l'ensemble des tailles saisies.

```
tailles
#> [1] 156 164 197 147 173
```

Un *vecteur* dans R est un objet qui peut contenir plusieurs informations du même type, potentiellement en très grand nombre.

L'avantage d'un vecteur est que lorsqu'on lui applique une opération, celle-ci s'applique à toutes les valeurs qu'il contient. Ainsi, si on veut la taille en mètres plutôt qu'en centimètres, on peut faire :

```
tailles_m <- tailles / 100
tailles_m
#> [1] 1.56 1.64 1.97 1.47 1.73
```

Cela fonctionne pour toutes les opérations de base.

```
tailles + 10
#> [1] 166 174 207 157 183
tailles^2
#> [1] 24336 26896 38809 21609 29929
```

Imaginons maintenant qu'on a aussi demandé aux cinq mêmes personnes leur poids en kilos. On peut créer un deuxième vecteur :

```
poids <- c(45, 59, 110, 44, 88)
```

On peut alors effectuer des calculs utilisant nos deux vecteurs **tailles** et **poids**. On peut par exemple calculer l'indice de masse corporelle (IMC) de chacun de nos enquêtés en divisant leur poids en kilo par leur taille en mètre au carré :

```
imc <- poids / (tailles / 100) ^ 2
imc
#> [1] 18.49112 21.93635 28.34394 20.36189 29.40292
```

Un vecteur peut contenir des nombres, mais il peut aussi contenir du texte. Imaginons qu'on a demandé aux 5 mêmes personnes leur niveau de diplôme : on peut regrouper l'information dans un vecteur de *chaînes de caractères*. Une chaîne de caractère contient du texte libre, délimité par des guillemets simples ou doubles.

```
diplome <- c("CAP", "Bac", "Bac+2", "CAP", "Bac+3")
diplome
#> [1] "CAP"   "Bac"   "Bac+2" "CAP"   "Bac+3"
```

L'opérateur `:`, lui, permet de générer rapidement un vecteur comprenant tous les nombres entre deux valeurs, opération assez courante sous R :

```
x <- 1:10
x
#> [1] 1 2 3 4 5 6 7 8 9 10
```

Enfin, notons qu'on peut accéder à un élément particulier d'un vecteur en faisant suivre le nom du vecteur de crochets contenant le numéro de l'élément désiré.

```
diplome[2]
#> [1] "Bac"
```

Cette opération, qui utilise l'opérateur `[]`, permet donc la sélection d'éléments d'un vecteur.

Dernière remarque, si on affiche dans la console un vecteur avec beaucoup d'éléments, ceux-ci seront répartis sur plusieurs lignes. Par exemple, si on a un vecteur de 50 nombres on peut obtenir quelque chose comme :

```
[1] 294 425 339 914 114 896 716 648 915 587 181 926 489
[14] 848 583 182 662 888 417 133 146 322 400 698 506 944
[27] 237 324 333 443 487 658 793 288 897 588 697 439 697
[40] 914 694 126 969 744 927 337 439 226 704 635
```

On remarque que R ajoute systématiquement un nombre entre crochets au début de chaque ligne : il s'agit en fait de la position du premier élément de la ligne dans le vecteur. Ainsi, le 848 de la deuxième ligne est le 14e élément du vecteur, le 914 de la dernière ligne est le 40e, etc.

Ceci explique le `[1]` qu'on obtient quand on affiche un simple nombre² :

```
[1] 4
```

2.3 Fonctions

2.3.1 Principe

Nous savons désormais effectuer des opérations arithmétiques de base sur des nombres et des vecteurs, et stocker des valeurs dans des objets pour pouvoir les réutiliser plus tard.

Pour aller plus loin, nous devons aborder les *fonctions* qui sont, avec les objets, un deuxième concept de base de R. On utilise des fonctions pour effectuer des calculs, obtenir des résultats et accomplir des actions.

²Et permet de constater que pour R, un nombre est un vecteur à un seul élément.

Formellement, une fonction a un *nom*, elle prend en entrée entre parenthèses un ou plusieurs *arguments* (ou *paramètres*), et retourne un *résultat*.

Prenons tout de suite un exemple. Si on veut connaître le nombre d'éléments du vecteur `tailles` que nous avons construit précédemment, on peut utiliser la fonction `length`, de cette manière :

```
length(tailles)
#> [1] 5
```

Ici, `length` est le nom de la fonction, on l'appelle en lui passant un argument entre parenthèses (en l'occurrence notre vecteur `tailles`), et elle nous renvoie un résultat, à savoir le nombre d'éléments du vecteur passé en paramètre.

Autre exemple, les fonctions `min` et `max` retournent respectivement les valeurs minimales et maximales d'un vecteur de nombres.

```
min(tailles)
#> [1] 147
max(tailles)
#> [1] 197
```

La fonction `mean` calcule et retourne la moyenne d'un vecteur de nombres.

```
mean(tailles)
#> [1] 167.4
```

La fonction `sum` retourne la somme de tous les éléments du vecteur.

```
sum(tailles)
#> [1] 837
```

Jusqu'à présent on n'a vu que des fonctions qui calculent et retournent un unique nombre. Mais une fonction peut renvoyer d'autres types de résultats. Par exemple, la fonction `range` (étendue) renvoie un vecteur de deux nombres, le minimum et le maximum.

```
range(tailles)
#> [1] 147 197
```

Ou encore, la fonction `unique`, qui supprime toutes les valeurs en double dans un vecteur, qu'il s'agisse de nombres ou de chaînes de caractères.

```
diplome <- c("CAP", "Bac", "Bac+2", "CAP", "Bac+3")
unique(diplome)
#> [1] "CAP"    "Bac"    "Bac+2"   "Bac+3"
```

2.3.2 Arguments

Une fonction peut prendre plusieurs arguments, dans ce cas on les indique entre parenthèses en les séparant par des virgules.

On a déjà rencontré un exemple de fonction acceptant plusieurs arguments : la fonction `c`, qui combine l'ensemble de ses arguments en un vecteur³.

³`c` est l'abréviation de *combine*, son nom est très court car on l'utilise très souvent

```
tailles <- c(156, 164, 197, 181, 173)
```

Ici, `c` est appelée en lui passant cinq arguments, les cinq tailles séparées par des virgules, et elle renvoie un vecteur numérique regroupant ces cinq valeurs.

Supposons maintenant que dans notre vecteur `tailles` nous avons une valeur manquante (une personne a refusé de répondre, ou notre mètre mesureur était en panne). On symbolise celle-ci dans R avec le code interne `NA`.

```
tailles <- c(156, 164, 197, NA, 173)
tailles
#> [1] 156 164 197 NA 173
```



`NA` est l'abréviation de *Not available*, non disponible. Cette valeur particulière peut être utilisée pour indiquer une valeur manquante, qu'il s'agisse d'un nombre, d'une chaîne de caractères, etc.

Si on calcule maintenant la taille moyenne à l'aide de la fonction `mean`, on obtient :

```
mean(tailles)
#> [1] NA
```

En effet, R considère par défaut qu'il ne peut pas calculer la moyenne si une des valeurs n'est pas disponible. Dans ce cas il considère que la moyenne est elle-même “non disponible” et renvoie donc `NA` comme résultat.

On peut cependant indiquer à `mean` d'effectuer le calcul en ignorant les valeurs manquantes. Ceci se fait en ajoutant un argument supplémentaire, nommé `na.rm` (abréviation de *NA remove*, “enlever les NA”), et de lui attribuer la valeur `TRUE` (code interne de R signifiant *vrai*).

```
mean(tailles, na.rm = TRUE)
#> [1] 172.5
```

Positionner le paramètre `na.rm` à `TRUE` indique à la fonction `mean` de ne pas tenir compte des valeurs manquantes dans le calcul.

Si on ne dit rien à la fonction `mean`, cet argument a une valeur par défaut, en l'occurrence `FALSE` (faux), qui fait qu'il ne supprime pas les valeurs manquantes. Les deux commandes suivantes sont donc rigoureusement équivalentes :

```
mean(tailles)
#> [1] NA
mean(tailles, na.rm = FALSE)
#> [1] NA
```



Lorsqu'on passe un argument à une fonction de cette manière, c'est-à-dire sous la forme `nom = valeur`, on parle d'*argument nommé*.

2.3.3 Aide sur une fonction

Il est fréquent de ne pas savoir (ou d'avoir oublié) quels sont les arguments d'une fonction, ou comment ils se nomment. On peut à tout moment faire appel à l'aide intégrée à R en passant le nom de la fonction (entre guillemets) à la fonction `help`.

```
help("mean")
```

On peut aussi utiliser le raccourci `?mean`.

Ces deux commandes affichent une page (en anglais) décrivant la fonction, ses paramètres, son résultat, le tout accompagné de diverses notes, références et exemples. Ces pages d'aide contiennent à peu près tout ce que vous pourrez chercher à savoir, mais elles ne sont pas toujours d'une lecture aisée.

Dans RStudio, les pages d'aide en ligne s'ouvriront par défaut dans la zone en bas à droite, sous l'onglet *Help*. Un clic sur l'icône en forme de maison vous affichera la page d'accueil de l'aide.

2.4 Regrouper ses commandes dans des scripts

Jusqu'ici on a utilisé R de manière “interactive”, en saisissant des commandes directement dans la console. Ça n'est cependant pas la manière dont on va utiliser R au quotidien, pour une raison simple : lorsque R ou RStudio redémarre, tout ce qui a été effectué dans la console est perdu.

Plutôt que de saisir nos commandes dans la console, on va donc les regrouper dans des scripts (de simples fichiers texte), qui vont garder une trace de toutes les opérations effectuées, et ce sont ces scripts, sauvegardés régulièrement, qui seront le “coeur” de notre travail. C'est en rouvrant les scripts et en réexécutant les commandes qu'ils contiennent qu'on pourra “reproduire” le chargement des données, leur traitement, les analyses et leurs résultats.

Pour créer un script, il suffit de sélectionner le menu *File*, puis *New file* et *R script*. Une quatrième zone apparaît alors en haut à gauche de l'interface de RStudio. On peut enregistrer notre script à tout moment dans un fichier avec l'extension `.R`, en cliquant sur l'icône de disquette ou en choisissant *File* puis *Save*.

Un script est un fichier texte brut, qui s'édite de la manière habituelle. À la différence de la console, quand on appuie sur **Entrée**, cela n'exécute pas la commande en cours mais insère un saut de ligne (comme on pouvait s'y attendre).

Pour exécuter une commande saisie dans un script, il suffit de positionner le curseur sur la ligne de la commande en question, et de cliquer sur le bouton *Run* dans la barre d'outils juste au-dessus de la zone d'édition du script. On peut aussi utiliser le raccourci clavier **Ctrl + Entrée** (**Cmd + Entrée** sous Mac). On peut enfin sélectionner plusieurs lignes avec la souris ou le clavier et cliquer sur *Run* (ou utiliser le raccourci clavier), et l'ensemble des lignes est exécuté d'un coup.

Au final, un script pourra ressembler à quelque chose comme ça :

```
tailles <- c(156, 164, 197, 147, 173)
poids <- c(45, 59, 110, 44, 88)

mean(tailles)
mean(poids)

imc <- poids / (tailles / 100) ^ 2
min(imc)
max(imc)
```

2.4.1 Commentaires

Les commentaires sont un élément très important d'un script. Il s'agit de texte libre, ignoré par R, et qui permet de décrire les étapes du script, sa logique, les raisons pour lesquelles on a procédé de telle ou telle manière... Il est primordial de documenter ses scripts à l'aide de commentaires, car il est très facile de ne plus se retrouver dans un programme qu'on a produit soi-même, même après une courte interruption.

Pour ajouter un commentaire, il suffit de le faire précéder d'un ou plusieurs symboles `#`. En effet, dès que R rencontre ce caractère, il ignore tout ce qui se trouve derrière, jusqu'à la fin de la ligne.

On peut donc documenter le script précédent :

```
# Saisie des tailles et poids des enquêtés
tailles <- c(156, 164, 197, 147, 173)
poids <- c(45, 59, 110, 44, 88)

# Calcul des tailles et poids moyens
mean(tailles)
mean(poids)

# Calcul de l'IMC (poids en kilo divisé par les tailles en mètre au carré)
imc <- poids / (tailles / 100) ^ 2
# Valeurs extrêmes de l'IMC
min(imc)
max(imc)
```

2.5 Installer et charger des extensions (*packages*)

R étant un logiciel libre, il bénéficie d'un développement communautaire riche et dynamique. L'installation de base de R permet de faire énormément de choses, mais le langage dispose en plus d'un système d'extensions permettant d'ajouter facilement de nouvelles fonctionnalités. La plupart des extensions sont développées et maintenues par la communauté des utilisateurs et utilisatrices de R, et diffusées via un réseau de serveurs nommé **CRAN** (*Comprehensive R Archive Network*).

Pour installer une extension, si on dispose d'une connexion Internet, on peut utiliser le bouton *Install* de l'onglet *Packages* de RStudio.

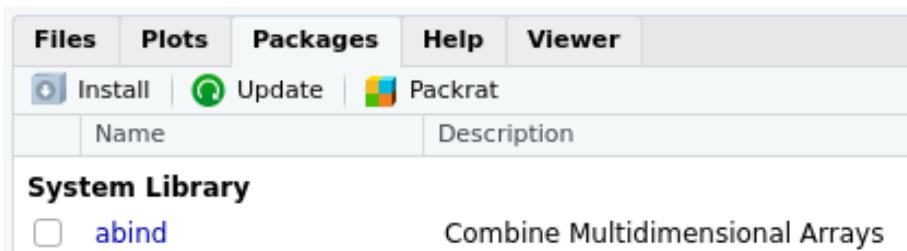


Figure 2.2: Installer une extension

Il suffit alors d'indiquer le nom de l'extension dans le champ *Package* et de cliquer sur *Install*.



Figure 2.3: Installation d'une extension

On peut aussi installer des extensions en utilisant la fonction `install.packages()` directement dans la console. Par exemple, pour installer le package `questionr` on peut exécuter la commande :

```
install.packages("questionr")
```

Installer une extension via l'une des deux méthodes précédentes va télécharger l'ensemble des fichiers nécessaires depuis l'une des machines du CRAN, puis installer tout ça sur le disque dur de votre ordinateur. Vous n'avez besoin de le faire qu'une fois, comme vous le faites pour installer un programme sur votre Mac ou PC.

Une fois l'extension installée, il faut la “charger” avant de pouvoir utiliser les fonctions qu'elle propose. Ceci se fait avec la fonction `library`. Par exemple, pour pouvoir utiliser les fonctions de `questionr`, vous devrez exécuter la commande suivante :

```
library(questionr)
```

Ainsi, on regroupe en général en début de script toute une série d'appels à `library` qui permettent de charger tous les packages utilisés dans le script. Quelque chose comme :

```
library(readxl)
library(ggplot2)
library(questionr)
```

Si vous essayez d'exécuter une fonction d'une extension et que vous obtenez le message d'erreur `impossible de trouver la fonction`, c'est certainement parce que vous n'avez pas exécuté la commande `library` correspondante.

2.6 Exercices

Exercice 1

Construire le vecteur `x` suivant :

```
#> [1] 120 134 256 12
```

Utiliser ce vecteur `x` pour générer les deux vecteurs suivants :

```
#> [1] 220 234 356 112
#> [1] 240 268 512 24
```

Exercice 2

On a demandé à 4 ménages le revenu des deux conjoints, et le nombre de personnes du ménage :

```
conjoint1 <- c(1200, 1180, 1750, 2100)
conjoint2 <- c(1450, 1870, 1690, 0)
nb_personnes <- c(4, 2, 3, 2)
```

Calculer le revenu total de chaque ménage, puis diviser par le nombre de personnes pour obtenir le revenu par personne de chaque ménage.

Exercice 3

Dans l'exercice précédent, calculer le revenu minimum et maximum parmi ceux du premier conjoint.

```
conjoint1 <- c(1200, 1180, 1750, 2100)
```

Recommencer avec les revenus suivants, parmi lesquels l'un des enquêtés n'a pas voulu répondre :

```
conjoint1 <- c(1200, 1180, 1750, NA)
```

Exercice 4

Les deux vecteurs suivants représentent les précipitations (en mm) et la température (en °C) moyennes pour chaque mois de l'année pour la ville de Lyon (moyennes calculées sur la période 1981-2010) :

```
temperature <- c(3.4, 4.8, 8.4, 11.4, 15.8, 19.4, 22.2, 21.6, 17.6, 13.4, 7.6, 4.4)
precipitations <- c(47.2, 44.1, 50.4, 74.9, 90.8, 75.6, 63.7, 62, 87.5, 98.6, 81.9, 55.2)
```

Calculer la température moyenne sur l'année.

Calculer la quantité totale de précipitations sur l'année.

À quoi correspond et comment peut-on interpréter le résultat de la fonction suivante ? Vous pouvez vous aider de la page d'aide de la fonction si nécessaire.

```
cumsum(precipitations)
#> [1] 47.2 91.3 141.7 216.6 307.4 383.0 446.7 508.7 596.2 694.8 776.7 831.9
```

Même question pour :

```
diff(temperature)
#> [1] 1.4 3.6 3.0 4.4 3.6 2.8 -0.6 -4.0 -4.2 -5.8 -3.2
```

Exercice 5

On a relevé les notes en maths, anglais et sport d'une classe de 6 élèves et on a stocké ces données dans trois vecteurs :

```
maths <- c(12, 16, 8, 18, 6, 10)
anglais <- c(14, 9, 13, 15, 17, 11)
sport <- c(18, 11, 14, 10, 8, 12)
```

Calculer la moyenne des élèves de la classe en anglais.

Calculer la moyenne générale de chaque élève (la moyenne des ses notes dans les trois matières).

Essayez de comprendre le résultat des deux fonctions suivantes (vous pouvez vous aider de la page d'aide de ces fonctions) :

```
pmin(maths, anglais, sport)
#> [1] 12 9 8 10 6 10
```

```
pmax(maths, anglais, sport)
#> [1] 18 16 14 18 17 12
```


Chapitre 3

Premier travail avec des données

3.1 Jeu de données d'exemple

Dans cette partie nous allons (enfin) travailler sur des “vraies” données, et utiliser un jeu de données présent dans l'extension `questionr`. Nous devons donc avant toute chose installer cette extension.

Pour installer ce package, deux possibilités :

- Dans l'onglet *Packages* de la zone de l'écran en bas à droite, cliquez sur le bouton *Install*. Dans le dialogue qui s'ouvre, entrez “`questionr`” dans le champ *Packages* puis cliquez sur *Install*.
- Saisissez directement la commande suivante dans la console : `install.packages("questionr")`

Dans les deux cas, tout un tas de messages devraient s'afficher dans la console. Attendez que l'invite de commandes > apparaisse à nouveau.

Pour plus d'informations sur les extensions et leur installation, voir la section [2.5](#).

Le jeu de données que nous allons utiliser est un extrait de l'enquête *Histoire de vie* réalisée par l'INSEE en 2003. Il contient 2000 individus et 20 variables. Pour une description plus complète et une liste des variables, voir la section [A.3.2.2](#).

Pour pouvoir utiliser ces données, il faut d'abord charger l'extension `questionr` (après l'avoir installée, bien entendu) :

```
library(questionr)
```

L'utilisation de `library` permet de rendre “disponibles”, dans notre session R, les fonctions et jeux de données inclus dans l'extension.

Nous devons ensuite indiquer à R que nous souhaitons accéder au jeu de données à l'aide de la commande `data` :

```
data(hdv2003)
```

Cette commande ne renvoie aucun résultat particulier (sauf en cas d'erreur), mais vous devriez voir apparaître dans l'onglet *Environment* de RStudio un nouvel objet nommé `hdv2003`.



Figure 3.1: Onglet *Environment*

Cet objet est d'un type nouveau : il s'agit d'un tableau de données.

3.2 Tableau de données (*data frame*)

Un *data frame* (ou tableau de données, ou table) est un type d'objet R qui contient des données au format tabulaire, avec les observations en ligne et les variables en colonnes, comme dans une feuille de tableur de type LibreOffice ou Excel.

Si on se contente d'exécuter le nom de notre tableau de données R va, comme à son habitude, nous l'afficher dans la console, ce qui est tout sauf utile.

```
hdv2003
```

Une autre manière d'afficher le contenu du tableau est de cliquer sur l'icône en forme de tableau à droite du nom de l'objet dans l'onglet *Environment* :



Figure 3.2: View icon

Ou d'utiliser la fonction `View` :

```
View(hdv2003)
```

Dans les deux cas votre tableau devrait s'afficher dans RStudio avec une interface de type tableur :

	id	age	sexe	nivetud	poids	occup	qualif	freres.sc
1	1	28	Femme	Enseignement supérieur y compris technique sup...	2634.3982	Exerce une profession	Employe	
2	2	23	Femme	NA	9738.3958	Etudiant, elevé	NA	
3	3	59	Homme	Dernière année d'études primaires	3994.1025	Exerce une profession	Technicien	
4	4	34	Homme	Enseignement supérieur y compris technique sup...	5731.6615	Exerce une profession	Technicien	
5	5	71	Femme	Dernière année d'études primaires	4329.0940	Retraite	Employe	
6	6	35	Femme	Enseignement technique ou professionnel court	8674.6994	Exerce une profession	Employe	
7	7	60	Femme	Dernière année d'études primaires	6165.8035	Au foyer	Ouvrier qualifié	
8	8	47	Homme	Enseignement technique ou professionnel court	12891.6408	Exerce une profession	Ouvrier qualifié	
9	9	20	Femme	NA	7808.8721	Etudiant, elevé	NA	
10	10	28	Homme	Enseignement technique ou professionnel long	2277.1605	Exerce une profession	Autre	
11	11	65	Femme	Enseignement supérieur y compris technique sup...	704.3227	Retraite	Employe	
12	12	47	Homme	2ème cycle	6697.8682	Exerce une profession	Ouvrier qualifié	
13	13	63	Femme	Dernière année d'études 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 arrête ses études, avant la dernière année 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 supérieur y compris technique sup...	4836.1393	Retraite	Cadre	
18	18	20	Femme	NA	1551.4846	Etudiant, elevé	NA	

Showing 1 to 19 of 2,000 entries

Figure 3.3: Interface “View”

Il est important de comprendre que l’objet `hdv2003` contient *l’intégralité* des données du tableau. On voit donc qu’un objet peut contenir des données de types très différents (simple nombre, texte, vecteur, tableau de données entier), et être potentiellement de très grande taille¹.



Sous R, on peut importer ou créer autant de tableaux de données qu’on le souhaite, dans les limites des capacités de sa machine.

Un *data frame* peut être manipulé comme les autres objets vus précédemment. On peut par exemple faire :

```
d <- hdv2003
```

ce qui va entraîner la copie de l’ensemble de nos données dans un nouvel objet nommé `d`. Ceci peut paraître parfaitement inutile mais a en fait l’avantage de fournir un objet avec un nom beaucoup plus court, ce qui diminuera la quantité de texte à saisir par la suite.

Pour résumer, comme nous avons désormais décidé de saisir nos commandes dans un script et non plus directement dans la console, les premières lignes de notre fichier de travail sur les données de l’enquête *Histoire de vie* pourraient donc ressembler à ceci :

```
## Chargement des extensions nécessaires
library(questionr)

## Jeu de données hdv2003
data(hdv2003)
d <- hdv2003
```

¹La seule limite pour la taille d’un objet étant la mémoire vive (RAM) de la machine sur laquelle tourne la session R.

3.2.1 Structure du tableau

Un tableau étant un objet comme un autre, on peut lui appliquer des fonctions. Par exemple, `nrow` et `ncol` retournent le nombre de lignes et de colonnes du tableau.

```
nrow(d)
#> [1] 2000
```

```
ncol(d)
#> [1] 20
```

La fonction `dim` renvoie ses dimensions, donc les deux nombres précédents.

```
dim(d)
#> [1] 2000 20
```

La fonction `names` retourne les noms des colonnes du tableau, c'est-à-dire la liste de nos *variables*.

```
names(d)
#> [1] "id"           "age"          "sexe"         "nivetud"
#> [5] "poids"        "occup"        "qualif"       "freres.soeurs"
#> [9] "clso"          "relig"        "trav.imp"     "trav.satisf"
#> [13] "hard.rock"   "lecture.bd"   "peche.chasse" "cuisine"
#> [17] "bricol"       "cinema"      "sport"        "heures.tv"
```

Enfin, la fonction `str` renvoie un descriptif plus détaillé de la structure du tableau. Elle liste les différentes variables, indique leur type² et affiche les premières valeurs.

```
str(d)
#> 'data.frame': 2000 obs. of 20 variables:
#> $ id            : int 1 2 3 4 5 6 7 8 9 10 ...
#> $ age           : int 28 23 59 34 71 35 60 47 20 28 ...
#> $ sexe          : Factor w/ 2 levels "Homme","Femme": 2 2 1 1 2 2 2 1 2 1 ...
#> $ nivetud        : Factor w/ 8 levels "N'a jamais fait d'etudes",...: 8 NA 3 8 3 6 3 6 NA 7 ...
#> $ poids          : num 2634 9738 3994 5732 4329 ...
#> $ occup          : Factor w/ 7 levels "Exerce une profession",...: 1 3 1 1 4 1 6 1 3 1 ...
#> $ qualif         : Factor w/ 7 levels "Ouvrier specialise",...: 6 NA 3 3 6 6 2 2 NA 7 ...
#> $ freres.soeurs: int 8 2 2 1 0 5 1 5 4 2 ...
#> $ clso           : Factor w/ 3 levels "Oui","Non","Ne sait pas": 1 1 2 2 1 2 1 2 1 2 ...
#> $ relig           : Factor w/ 6 levels "Pratiquant regulier",...: 4 4 4 3 1 4 3 4 3 2 ...
#> $ trav.imp        : Factor w/ 4 levels "Le plus important",...: 4 NA 2 3 NA 1 NA 4 NA 3 ...
#> $ trav.satisf    : Factor w/ 3 levels "Satisfaction",...: 2 NA 3 1 NA 3 NA 2 NA 1 ...
#> $ hard.rock       : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 ...
#> $ lecture.bd     : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 ...
#> $ peche.chasse   : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 2 2 1 1 ...
#> $ cuisine         : Factor w/ 2 levels "Non","Oui": 2 1 1 2 1 1 2 2 1 1 ...
#> $ bricol          : Factor w/ 2 levels "Non","Oui": 1 1 1 2 1 1 1 2 1 1 ...
#> $ cinema          : Factor w/ 2 levels "Non","Oui": 1 2 1 2 1 2 1 1 2 2 ...
#> $ sport           : Factor w/ 2 levels "Non","Oui": 1 2 2 2 1 2 1 1 1 2 ...
#> $ heures.tv       : num 0 1 0 2 3 2 2.9 1 2 2 ...
```

²Les différents types de variables seront décrits plus en détail dans le chapitre 9 sur les recodages.

À noter que sous RStudio, on peut afficher à tout moment la structure d'un objet en cliquant sur l'icône de triangle sur fond bleu à gauche du nom de l'objet dans l'onglet *Environment*.

The screenshot shows the RStudio interface with the 'Environment' tab selected. A blue circular icon with a white triangle is positioned to the left of the variable name 'd'. To the right of the icon, the text '2000 obs. of 20 variables' is displayed. Below this, a list of 20 variables is shown, each with its type and a partial list of values. The variables include: id, age, sexe, nivetud, poids, occup, qualif, freres.soeurs, cuso, relig, trav.imp, trav.satisf, hard.rock, and a few unnamed variables ending in ellipses.

```

d 2000 obs. of 20 variables
  id : int 1 2 3 4 5 6 7 8 9 10 ...
  age : int 28 23 59 34 71 35 60 47 20 28 ...
  sexe : Factor w/ 2 levels "Homme","Femme": 2 2 1 1 2 2 2 1 2 1 ...
  nivetud : Factor w/ 8 levels "N'a jamais fait d'etudes",...: 8 NA 3 8 3 6 3 6 NA 7 ...
  poids : num 2634 9738 3994 5732 4329 ...
  occup : Factor w/ 7 levels "Exerce une profession",...: 1 3 1 1 4 1 6 1 3 1 ...
  qualif : Factor w/ 7 levels "Ouvrier specialise",...: 6 NA 3 3 6 6 2 2 NA 7 ...
  freres.soeurs: int 8 2 2 1 0 5 1 5 4 2 ...
  cuso : Factor w/ 3 levels "Oui","Non","Ne sait pas": 1 1 2 2 1 2 1 2 1 2 ...
  relig : Factor w/ 6 levels "Pratiquant regulier",...: 4 4 4 3 1 4 3 4 3 2 ...
  trav.imp : Factor w/ 4 levels "Le plus important",...: 4 NA 2 3 NA 1 NA 4 NA 3 ...
  trav.satisf : Factor w/ 3 levels "Satisfaction",...: 2 NA 3 1 NA 3 NA 2 NA 1 ...
  hard.rock : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...

```

Figure 3.4: Structure d'un objet

3.2.2 Accéder aux variables d'un tableau

Une opération très importante est l'accès aux variables du tableau (à ses colonnes) pour pouvoir les manipuler, effectuer des calculs, etc. On utilise pour cela l'opérateur \$, qui permet d'accéder aux colonnes du tableau. Ainsi, si l'on tape :

```

d$sexe
#> [1] Femme Femme Homme Homme Femme Femme Homme Femme Homme Femme Homme
#> [13] Femme Femme Femme Femme Homme Femme Femme Homme Femme Femme Femme
#> [25] Femme Homme Femme Homme Homme Homme Homme Femme Femme Femme Femme
#> [37] Homme Femme Femme Homme Femme Homme Femme Femme Femme Femme Femme Femme
#> [49] Femme Femme Homme Femme Homme Femme Femme Femme Femme Femme Femme Femme
#> [61] Femme Homme Homme Homme Femme Homme Femme Femme Femme Homme Homme
#> [73] Femme Femme Femme Femme Homme Femme Femme Femme Femme Femme Femme Femme
#> [85] Homme Femme Homme Homme Homme Homme Femme Homme Femme Femme Femme Femme
#> [97] Homme Homme Femme Femme Homme Femme Homme Femme Femme Femme Femme Femme
#> [109] Femme Homme Homme Homme Homme Femme Homme Femme Femme Homme Homme
#> [121] Femme Femme Femme Homme Femme Femme Homme Femme Femme Femme Femme Homme
#> [133] Femme Femme Femme Homme Homme Homme Homme Homme Femme Femme Femme Femme
#> [145] Homme Homme Homme Femme Femme Homme Femme Femme Femme Femme Femme Homme
#> [157] Femme Homme Homme Homme Femme Femme Homme Femme Femme Homme Homme Femme
#> [169] Femme Femme Homme Femme Homme Femme Femme Femme Homme Femme Homme Femme
#> [181] Homme Femme Femme Homme Homme Femme Femme Femme Femme Femme Femme Homme Homme
#> [193] Femme Homme Homme Femme Homme Femme Femme Femme Femme Femme Femme Homme Femme
#> [ reached getOption("max.print") -- omitted 1800 entries ]
#> Levels: Homme Femme

```

R va afficher l'ensemble des valeurs de la variable `sexe` dans la console, ce qui est à nouveau fort peu utile. Mais cela nous permet de constater que `d$sexe` est un vecteur de chaînes de caractères tels qu'on en a déjà rencontré précédemment.

La fonction `table$colonne` renvoie donc la colonne nommée `colonne` du tableau `table`, c'est-à-dire un vecteur, en général de nombres ou de chaînes de caractères.

Si on souhaite afficher seulement les premières ou dernières valeurs d'une variable, on peut utiliser les fonctions `head` et `tail`.

```
head(d$age)
#> [1] 28 23 59 34 71 35
```

```
tail(d$age, 10)
#> [1] 52 42 50 41 46 45 46 24 24 66
```

Le deuxième argument numérique permet d'indiquer le nombre de valeurs à afficher.

3.2.3 Créer une nouvelle variable

On peut aussi utiliser l'opérateur `$` pour créer une nouvelle variable dans notre tableau : pour cela, il suffit de lui assigner une valeur.

Par exemple, la variable `heures.tv` contient le nombre d'heures passées quotidiennement devant la télé.

```
head(d$heures.tv, 10)
#> [1] 0.0 1.0 0.0 2.0 3.0 2.0 2.9 1.0 2.0 2.0
```

On peut vouloir créer une nouvelle variable dans notre tableau qui contienne la même durée convertie en minutes. On va donc créer une nouvelle variables `minutes.tv` de la manière suivante :

```
d$minutes.tv <- d$heures.tv * 60
```

On peut alors constater, soit visuellement soit dans la console, qu'une nouvelle variable (une nouvelle colonne) a bien été ajoutée au tableau.

```
head(d$minutes.tv)
#> [1] 0 60 0 120 180 120
```

3.3 Analyse univariée

On a donc désormais accès à un tableau de données `d`, dont les lignes sont des observations (des individus enquêtés), et les colonnes des variables (des caractéristiques de chacun de ces individus), et on sait accéder à ces variables grâce à l'opérateur `$`.

Si on souhaite analyser ces variables, les méthodes et fonctions utilisées seront différentes selon qu'il s'agit d'une variable *quantitative* (variable numérique pouvant prendre un grand nombre de valeurs : l'âge, le revenu, un pourcentage...) ou d'une variable *qualitative* (variable pouvant prendre un nombre limité de valeurs appelées modalités : le sexe, la profession, le dernier diplôme obtenu, etc.).

3.3.1 Analyser une variable quantitative

Une variable quantitative est une variable de type numérique (un nombre) qui peut prendre un grand nombre de valeurs. On en a plusieurs dans notre jeu de données, notamment l'âge (variable `age`) ou le nombre d'heures passées devant la télé (`heures.tv`).

3.3.1.1 Indicateurs de centralité

Caractériser une variable quantitative, c'est essayer de décrire la manière dont ses valeurs se répartissent, ou se distribuent.

Pour cela on peut commencer par regarder les valeurs extrêmes, avec les fonctions `min`, `max` ou `range`.

```
min(d$age)
#> [1] 18
max(d$age)
#> [1] 97
range(d$age)
#> [1] 18 97
```

On peut aussi calculer des indicateurs de *centralité* : ceux-ci indiquent autour de quel nombre se répartissent les valeurs de la variable. Il y en a plusieurs, le plus connu étant la moyenne, qu'on peut calculer avec la fonction `mean`.

```
mean(d$age)
#> [1] 48.157
```

Il existe aussi la médiane, qui est la valeur qui sépare notre population en deux : on a la moitié de nos observations en-dessous, et la moitié au-dessus. Elle se calcule avec la fonction `median`.

```
median(d$age)
#> [1] 48
```

Une différence entre les deux indicateurs est que la médiane est beaucoup moins sensible aux valeurs “extrêmes” : on dit qu’elle est plus *robuste*. Ainsi, en 2019, le salaire net *moyen* des salariés à temps plein dans le secteur privé en France était de 2424 euros, tandis que le salaire net *médian* n’était que de 1940 euros. La différence étant due à des très hauts salaires qui “tirent” la moyenne vers le haut.

3.3.1.2 Indicateurs de dispersion

Les indicateurs de dispersion permettent de mesurer si les valeurs sont plutôt regroupées ou au contraire plutôt dispersées.

L’indicateur le plus simple est l’étendue de la distribution, qui décrit l’écart maximal observé entre les observations :

```
max(d$age) - min(d$age)
#> [1] 79
```

Les indicateurs de dispersion les plus utilisés sont la variance ou, de manière équivalente, l’écart-type (qui est égal à la racine carrée de la variance). On obtient la première avec la fonction `var`, et le second avec `sd` (abréviation de *standard deviation*).

```
var(d$age)
#> [1] 287.0249
```

```
sd(d$age)
#> [1] 16.94181
```

Plus la variance ou l'écart-type sont élevés, plus les valeurs sont dispersées autour de la moyenne. À l'inverse, plus ils sont faibles et plus les valeurs sont regroupées.

Une autre manière de mesurer la dispersion est de calculer les quartiles :

- le premier quartile est la valeur pour laquelle on a 25% des observations en dessous et 75% au dessus
- le deuxième quartile est la valeur pour laquelle on a 50% des observations en dessous et 50% au dessus (c'est donc la médiane)
- le troisième quartile est la valeur pour laquelle on a 75% des observations en dessous et 25% au dessus

On peut les calculer avec la fonction `quantile` :

```
## Premier quartile
quantile(d$age, prob = 0.25)
#> 25%
#> 35
```

```
## Troisième quartile
quantile(d$age, prob = 0.75)
#> 75%
#> 60
```

`quantile` prend deux arguments principaux : le vecteur dont on veut calculer le quantile, et un argument `prob` qui indique quel quantile on souhaite obtenir. `prob` prend une valeur entre 0 et 1 : 0.5 est la médiane, 0.25 le premier quartile, 0.1 le premier décile, etc.

Notons enfin que la fonction `summary` permet d'obtenir d'un seul coup plusieurs indicateurs classiques :

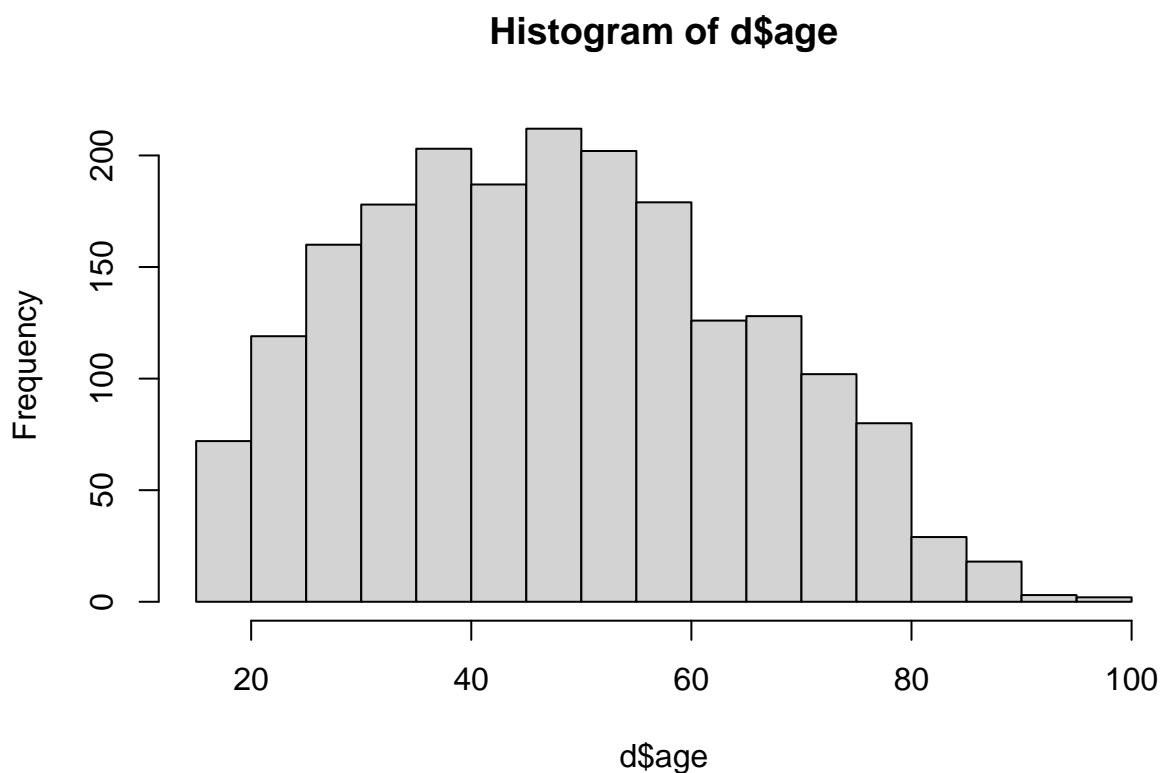
```
summary(d$age)
#>   Min. 1st Qu. Median  Mean 3rd Qu.  Max.
#> 18.00 35.00 48.00 48.16 60.00 97.00
```

3.3.1.3 Représentation graphique

L'outil le plus utile pour étudier la distribution des valeurs d'une variable quantitative reste la représentation graphique.

La représentation la plus courante est sans doute l'histogramme. On peut l'obtenir avec la fonction `hist`.

```
hist(d$age)
```



Cette fonction n'a pas pour effet direct d'effectuer un calcul ou de nous renvoyer un résultat : elle génère un graphique qui va s'afficher dans l'onglet *Plots* de RStudio.

On peut personnaliser l'apparence de l'histogramme en ajoutant des arguments supplémentaires à la fonction `hist`. L'argument le plus important est `breaks`, qui permet d'indiquer le nombre de classes que l'on souhaite.

```
hist(d$age, breaks = 10)
```



```
hist(d$age, breaks = 70)
```



Le choix d'un "bon" nombre de classes pour un histogramme n'est pas un problème simple : si on a trop peu

de classes, on risque d'effacer quasiment toutes les variations, et si on en a trop on risque d'avoir trop de détails et de masquer les grandes tendances.

Les arguments de `hist` permettent également de modifier la présentation du graphique. On peut ainsi changer la couleur des barres avec `col`³, le titre avec `main`, les étiquettes des axes avec `xlab` et `ylab`, etc. :

```
hist(d$age, col = "skyblue",
  main = "Répartition des âges des enquêtés",
  xlab = "Âge",
  ylab = "Effectif")
```



La fonction `hist` fait partie des fonctions graphiques de base de R. On verra plus en détail d'autres fonctions graphiques dans la partie 8 de ce document, consacrée à l'extension `ggplot2`, qui fait partie du `tidyverse` et qui permet la production et la personnalisation de graphiques complexes.

3.3.2 Analyser une variable qualitative

Une variable qualitative est une variable qui ne peut prendre qu'un nombre limité de valeurs, appelées modalités. Dans notre jeu de données on trouvera par exemple le sexe (`sex`), le niveau d'études (`niveau`), la catégorie socio-professionnelle (`qualif`)...

À noter qu'une variable qualitative peut tout-à-fait être numérique, et que certaines variables peuvent être traitées soit comme quantitatives, soit comme qualitatives : c'est le cas par exemple du nombre d'enfants ou du nombre de frères et sœurs.

3.3.2.1 Tri à plat

L'outil le plus utilisé pour représenter la répartition des valeurs d'une variable qualitative est le *tri à plat* : il s'agit simplement de compter, pour chacune des valeurs possibles de la variable (pour chacune des modalités), le nombre d'observations ayant cette valeur. Un tri à plat s'obtient sous R à l'aide de la fonction `table`.

³Les différentes manières de spécifier des couleurs sont indiquées dans l'encadré de la section 8.7.3.

```
table(d$sexe)
#>
#> Homme Femme
#> 899 1101
```

Ce tableau nous indique donc que parmi nos enquêtés on trouve 899 hommes et 1101 femmes.

```
table(d$qualif)
#>
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>              203                  292                   86
#> Profession intermediaire      Cadre                 Employe
#>                      160                  260                   594
#>      Autre
#>              58
```

Un tableau de ce type peut être affiché ou stocké dans un objet, et on peut à son tour lui appliquer des fonctions. Par exemple, la fonction `sort` permet de trier le tableau selon la valeur de l'effectif.

```
tab <- table(d$qualif)
sort(tab)
#>
#>      Autre      Technicien Profession intermediaire
#>      58          86                  160
#>      Ouvrier specialise      Cadre      Ouvrier qualifie
#>              203              260                  292
#>      Employe
#>      594
```



Attention, par défaut la fonction `table` n'affiche pas les valeurs manquantes (`NA`). Si on souhaite les inclure il faut utiliser l'argument `useNA = "always"`, soit : `table(d$qualif, useNA = "always")`.

À noter qu'on peut aussi appliquer `summary` à une variable qualitative. Le résultat est également le tri à plat de la variable, avec en plus le nombre de valeurs manquantes éventuelles.

```
summary(d$qualif)
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>              203                  292                   86
#> Profession intermediaire      Cadre                 Employe
#>                      160                  260                   594
#>      Autre      NA's
#>              58                  347
```

Par défaut ces tris à plat sont en effectifs et ne sont pas toujours très lisibles, notamment quand on a des effectifs importants. On leur rajoute donc en général la répartition en pourcentages. Pour cela, nous allons utiliser la fonction `freq` de l'extension `questionr`, qui devra donc avoir précédemment été chargée avec `library(questionr)`.

```
## À rajouter en haut de script et à exécuter
library(questionr)
```

Une fois l'extension chargée on peut utiliser la fonction `freq`.

```
freq(d$qualif)
#>          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
```

La colonne `n` représente les effectifs de chaque catégorie, la colonne `%` le pourcentage, et la colonne `val%` le pourcentage calculé sur les valeurs valides, donc en excluant les `NA`. Une ligne a également été rajoutée pour indiquer le nombre et la proportion de `NA`.

`freq` accepte un certain nombre d'arguments pour personnaliser son affichage. Par exemple :

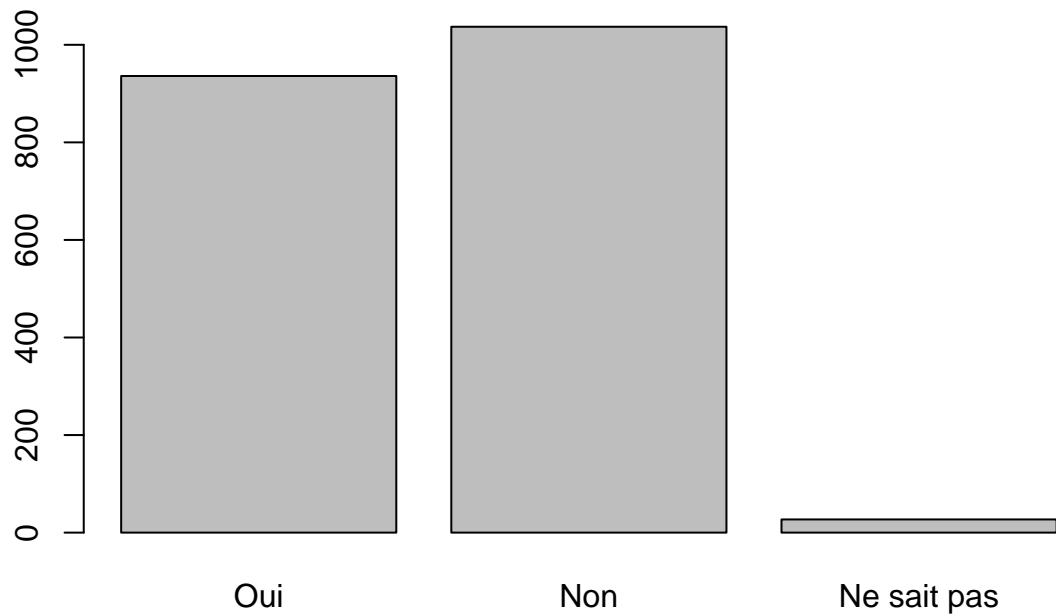
- `valid` indique si on souhaite ou non afficher les pourcentages sur les valeurs valides
- `cum` indique si on souhaite ou non afficher les pourcentages cumulés
- `total` permet d'ajouter une ligne avec les effectifs totaux
- `sort` permet de trier le tableau par fréquence croissante (`sort="inc"`) ou décroissante (`sort="dec"`).

```
freq(d$qualif, valid = FALSE, total = TRUE, sort = "dec")
#>          n   %
#> Employe      594 29.7
#> Ouvrier qualifie 292 14.6
#> Cadre        260 13.0
#> Ouvrier specialise 203 10.2
#> Profession intermediaire 160  8.0
#> Technicien    86  4.3
#> Autre          58  2.9
#> NA              347 17.3
#> Total         2000 100.0
```

3.3.2.2 Représentations graphiques

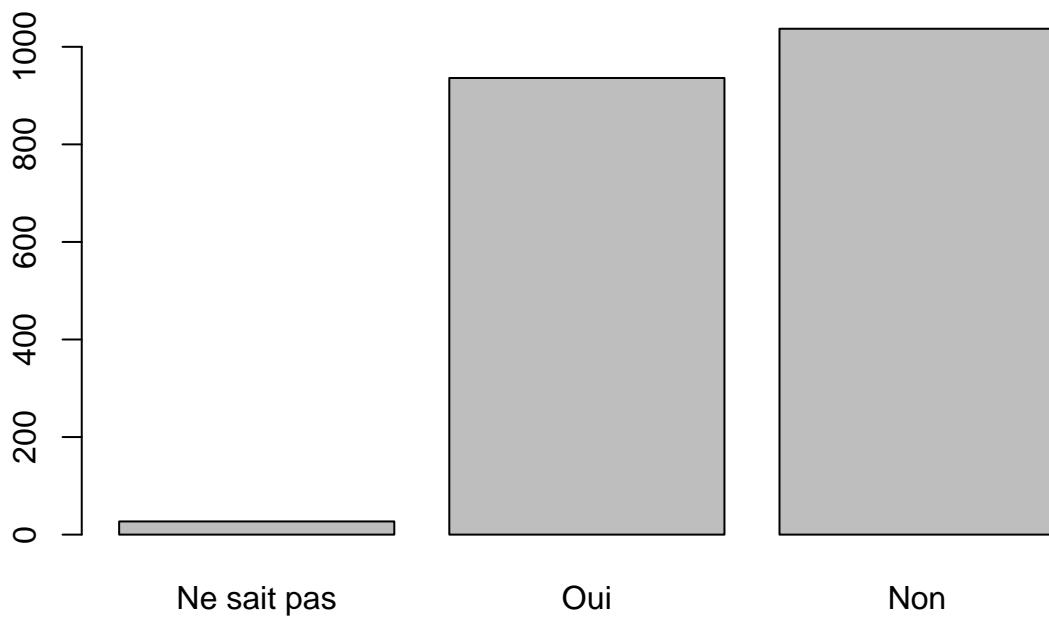
On peut représenter graphiquement le tri à plat d'une variable qualitative avec un diagramme en barres, obtenu avec la fonction `barplot`. Attention, contrairement à `hist` cette fonction ne s'applique pas directement à la variable mais au résultat du tri à plat de cette variable, calculé avec `table`. Il faut donc procéder en deux étapes.

```
tab <- table(d$cuso)
barplot(tab)
```



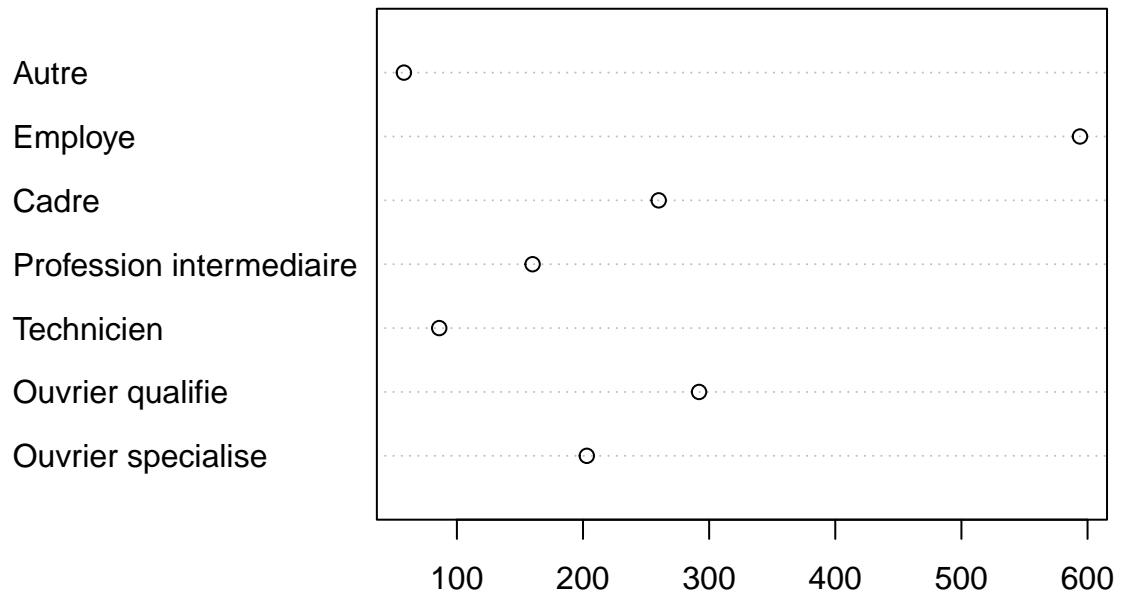
On peut aussi trier le tri à plat avec la fonction `sort` avant de le représenter graphiquement, ce qui peut faciliter la lecture du graphique :

```
barplot(sort(tab))
```



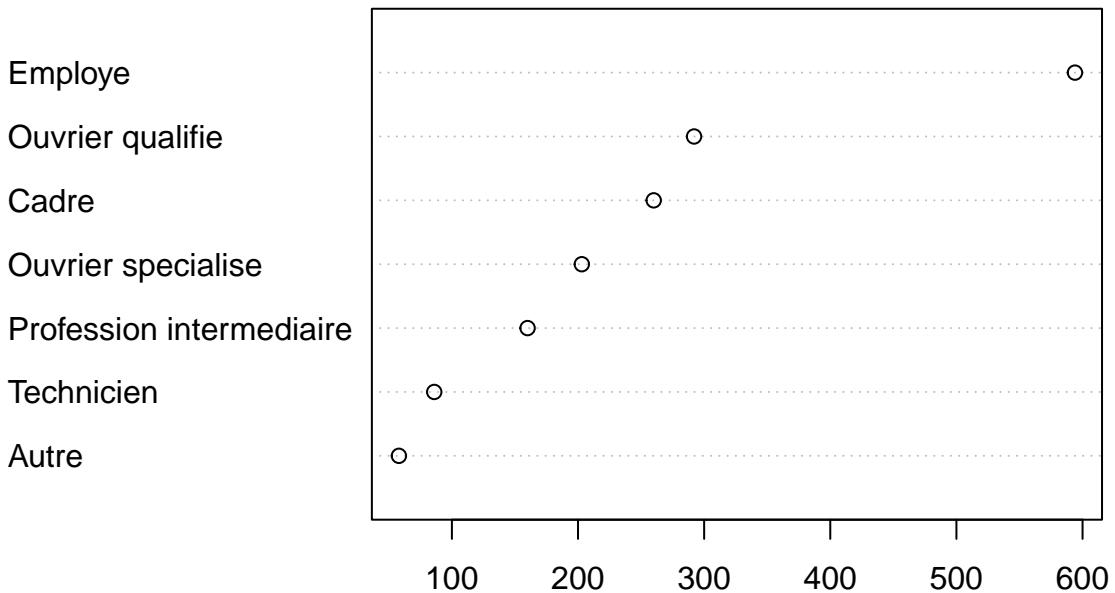
Une alternative au graphique en barres est le *diagramme de Cleveland*, qu'on peut obtenir avec la fonction `dotchart`. Celle-ci s'applique elle aussi au tri à plat de la variable calculé avec `table`.

```
dotchart(table(d$qualif))
```



Là aussi, pour améliorer la lisibilité du graphique il est préférable de trier le tri à plat de la variable avant de le représenter :

```
dotchart(sort(table(d$qualif)))
```



3.4 Exercices

Exercice 1

Créer un nouveau script qui effectue les actions suivantes :

- charger l'extension `questionr`
- charger le jeu de données nommé `hdv2003`
- copier le jeu de données dans un nouvel objet nommé `df`
- afficher les dimensions et la liste des variables de `df`

Exercice 2

On souhaite étudier la répartition du temps passé devant la télévision par les enquêtés (variable `heures.tv`). Pour cela, affichez les principaux indicateurs de cette variable : valeur minimale, maximale, moyenne, médiane et écart-type. Représentez ensuite sa distribution par un histogramme en 10 classes.

Exercice 3

On s'intéresse maintenant à l'importance accordée par les enquêtés à leur travail (variable `trav.imp`). Faites un tri à plat des effectifs des modalités de cette variable avec la commande `table`.

Faites un tri à plat affichant à la fois les effectifs et les pourcentages de chaque modalité. Y'a-t-il des valeurs manquantes ?

Représentez graphiquement les effectifs des modalités à l'aide d'un graphique en barres.

Utilisez l'argument `col` de la fonction `barplot` pour modifier la couleur du graphique en `tomato`.

Tapez `colors()` dans la console pour afficher l'ensemble des noms de couleurs disponibles dans R. Testez chaque couleur une à une pour trouver votre couleur préférée.

Chapitre 4

Analyse bivariée

Faire une analyse bivariée, c'est étudier la relation entre deux variables : sont-elles liées ? les valeurs de l'une influencent-elles les valeurs de l'autre ? ou sont-elles au contraire indépendantes ?

À noter qu'on va parler ici d'influence ou de lien, mais pas de relation de cause à effet. Les outils présentés permettent de visualiser ou de déterminer une relation, mais la mise en évidence de liens de causalité proprement dit est nettement plus complexe : il faut en effet vérifier que c'est bien telle variable qui influence telle autre et pas l'inverse, qu'il n'y a pas de "variable cachée", etc.

Là encore, le type d'analyse ou de visualisation est déterminé par la nature qualitative ou quantitative des deux variables.

4.1 Croisement de deux variables qualitatives

4.1.1 Tableaux croisés

On continue à travailler avec le jeu de données tiré de l'enquête *Histoire de vie* inclus dans l'extension `questionr`. On commence donc par charger l'extension, le jeu de données, et à le renommer en un nom plus court pour gagner un peu de temps de saisie au clavier.

```
library(questionr)
data(hdv2003)
d <- hdv2003
```

Quand on veut croiser deux variables qualitatives, on fait un *tableau croisé*. Comme pour un tri à plat ceci s'obtient avec la fonction `table` de R, mais à laquelle on passe cette fois deux variables en argument. Par exemple, si on veut croiser la catégorie socio-professionnelle et le sexe des enquêtés :

```
table(d$qualif, d$sex)
#>                               Homme Femme
#>   Ouvrier specialise        96   107
#>   Ouvrier qualifie         229    63
#>   Technicien                 66    20
#>   Profession intermediaire  88    72
#>   Cadre                      145   115
#>   Employe                   96   498
#>   Autre                      21     37
```

Pour pouvoir interpréter ce tableau on doit passer du tableau en effectifs au tableau en pourcentages ligne ou colonne. Pour cela, on peut utiliser les fonctions `lprop` et `cprop` de l'extension `questionr`, qu'on applique au tableau croisé précédent.

Pour calculer les pourcentages ligne :

```
tab <- table(d$qualif, d$sex)
lprop(tab)
#>
#>                               Homme Femme Total
#> Ouvrier specialise      47.3  52.7 100.0
#> Ouvrier qualifie       78.4  21.6 100.0
#> Technicien                76.7  23.3 100.0
#> Profession intermediaire 55.0  45.0 100.0
#> Cadre                      55.8  44.2 100.0
#> Employe                   16.2  83.8 100.0
#> Autre                      36.2  63.8 100.0
#> Ensemble                  44.8  55.2 100.0
```

Et pour les pourcentages colonne :

```
cprop(tab)
#>
#>                               Homme Femme Ensemble
#> Ouvrier specialise      13.0  11.7  12.3
#> Ouvrier qualifie       30.9   6.9  17.7
#> Technicien                 8.9   2.2   5.2
#> Profession intermediaire 11.9   7.9   9.7
#> Cadre                      19.6  12.6  15.7
#> Employe                   13.0  54.6  35.9
#> Autre                      2.8   4.1   3.5
#> Total                      100.0 100.0 100.0
```



Pour savoir si on doit faire des pourcentages ligne ou colonne, on pourra se référer à l'article suivant :

<http://alain-leger.lescigales.org/textes/lignecolonne.pdf>

En résumé, quand on fait un tableau croisé, celui-ci est parfaitement symétrique : on peut inverser les lignes et les colonnes, ça ne change pas son interprétation. Par contre, on a toujours en tête un “sens” de lecture dans le sens où on considère que l’une des variables *dépend* de l’autre. Par exemple, si on croise sexe et type de profession, on dira que le type de profession dépend du sexe, et non l’inverse : le type de profession est alors la variable *dépendante* (à expliquer), et le sexe la variable *indépendante* (explicative).

Pour faciliter la lecture d’un tableau croisé, il est recommandé de **faire les pourcentages sur la variable indépendante**. Dans notre exemple, la variable indépendante est le sexe, elle est en colonne, on calcule donc les pourcentages colonnes qui permettent de comparer directement, pour chaque sexe, la répartition des catégories socio-professionnelles.

4.1.2 Test du χ^2

Comme on travaille sur un échantillon et pas sur une population entière, on peut compléter ce tableau croisé par un test d’indépendance du χ^2 . Celui-ci permet de tester, et éventuellement de rejeter, l’hypothèse d’indépendance des lignes et des colonnes du tableau, c’est à dire l’hypothèse que les écarts à l’indépendance observés seraient uniquement dus au biais d’échantillonnage (au fait qu’on n’a pas interrogé toute notre population).

Pour effectuer un test de ce type, on applique la fonction `chisq.test` au tableau croisé calculé précédemment.

```
chisq.test(tab)
#>
#> Pearson's Chi-squared test
#>
#> data: tab
#> X-squared = 387.56, df = 6, p-value < 2.2e-16
```

Le résultat nous indique trois valeurs :

- **X-squared**, la valeur de la statistique du χ^2 pour notre tableau, c'est-à-dire une “distance” entre notre tableau observé et celui attendu si les deux variables étaient indépendantes.
- **df**, le nombre de degrés de libertés du test, qui dépend des dimensions du tableau.
- **p-value**, le fameux p , qui indique la probabilité d’obtenir une valeur de la statistique du χ^2 au moins aussi extrême sous l’hypothèse d’indépendance.

Ici, le p est extrêmement petit (la notation $< 2.2e-16$ indique qu'il est plus petit que la plus petite valeur proche de zéro calculable par R), donc certainement en-dessous du seuil de décision choisi préalablement au test (souvent 5%, soit 0.05). On peut donc rejeter l’hypothèse d’indépendance des lignes et des colonnes du tableau.

En complément du test du χ^2 , on peut aussi regarder les *résidus* de ce test pour affiner la lecture du tableau. Ceux-ci s’obtiennent avec la fonction `chisq.residuals` de `questionr` :

```
chisq.residuals(tab)
#>
#>                               Homme   Femme
#> Ouvrier specialise        0.52 -0.47
#> Ouvrier qualifie         8.57 -7.73
#> Technicien                 4.42 -3.98
#> Profession intermediaire  1.92 -1.73
#> Cadre                      2.64 -2.38
#> Employe                  -10.43  9.41
#> Autre                      -0.98  0.88
```

L’interprétation des résidus est la suivante :

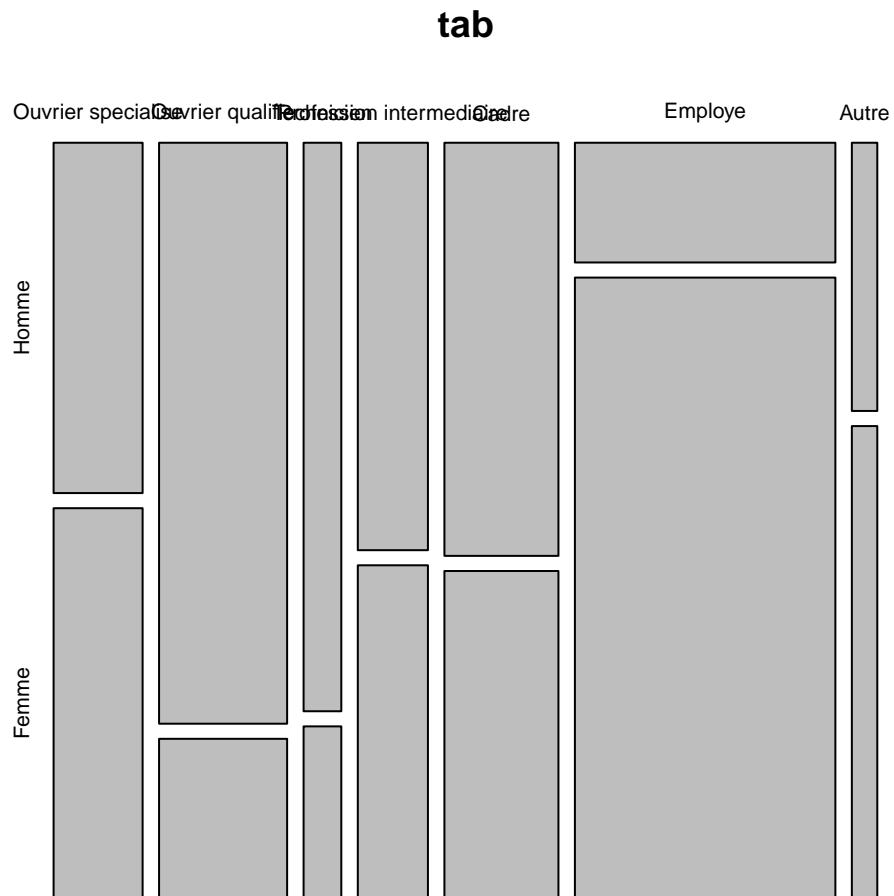
- si la valeur du résidu pour une case est inférieure à -2, alors il y a une sous-représentation de cette case dans le tableau : les effectifs sont significativement plus faibles que ceux attendus sous l’hypothèse d’indépendance
- à l’inverse, si le résidu est supérieur à 2, il y a sur-représentation de cette case
- si le résidu est compris entre -2 et 2, il n’y a pas d’écart à l’indépendance significatif

Les résidus peuvent être une aide utile à l’interprétation, notamment pour des tableaux de grande dimension.

4.1.3 Représentation graphique

Il est possible de faire une représentation graphique d’un tableau croisé, par exemple avec la fonction `mosaicplot` :

```
mosaicplot(tab)
```



On peut améliorer ce graphique en colorant les cases selon les résidus du test du χ^2 (argument `shade = TRUE`) et en orientant verticalement les labels de colonnes (argument `las = 3`) :

```
mosaicplot(tab, las = 3, shade = TRUE)
```



Chaque rectangle de ce graphique représente une case de tableau. Sa largeur correspond au pourcentage des modalités en colonnes (il y'a beaucoup d'employés et d'ouvriers et très peu d'"autres"). Sa hauteur correspond aux pourcentages colonnes : la proportion d'hommes chez les cadres est plus élevée que chez les employés. Enfin, la couleur de la case correspond au résidu du test du χ^2 correspondant : les cases en rouge sont sous-représentées, les cases en bleu sur-représentées, et les cases blanches sont proches des effectifs attendus sous l'hypothèse d'indépendance.

4.2 Croisement d'une variable quantitative et d'une variable qualitative

4.2.1 Représentation graphique

Croiser une variable quantitative et une variable qualitative, c'est essayer de voir si les valeurs de la variable quantitative se répartissent différemment selon la catégorie d'appartenance de la variable qualitative.

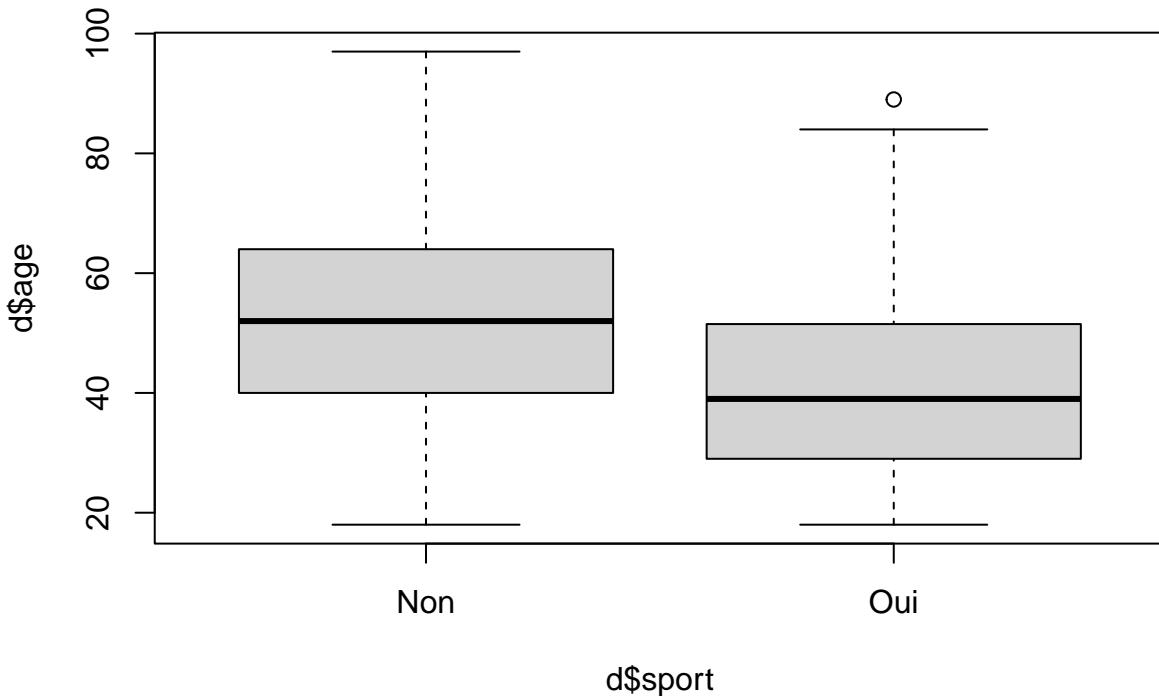
Pour cela, l'idéal est de commencer par une représentation graphique de type "boîte à moustache" à l'aide de la fonction `boxplot`. Par exemple, si on veut visualiser la répartition des âges selon la pratique ou non d'un sport, on va utiliser la syntaxe suivante :

```
boxplot(d$age ~ d$sport)
```



Cette syntaxe de `boxplot` utilise une nouvelle notation de type “formule”. Celle-ci est utilisée notamment pour la spécification des modèles de régression. Ici le `~` peut se lire comme “en fonction de” : on veut représenter le boxplot de l’âge en fonction du sport.

Ce qui va nous donner le résultat suivant :



L’interprétation d’un boxplot est la suivante : Les bords inférieurs et supérieurs du carré central représentent le premier et le troisième quartile de la variable représentée sur l’axe vertical. On a donc 50% de nos observations dans cet intervalle. Le trait horizontal dans le carré représente la médiane. Enfin, des “moustaches” s’étendent de chaque côté du carré, jusqu’aux valeurs minimales et maximales, avec une exception : si des valeurs sont éloignées du carré de plus de 1,5 fois l’écart interquartile (la hauteur du carré), alors on les représente sous forme de points (symbolisant des valeurs considérées comme “extrêmes”).

Dans le graphique ci-dessus, on voit que ceux qui ont pratiqué un sport au cours des douze derniers mois ont l’air d’être sensiblement plus jeunes que les autres.

4.2.2 Calculs d’indicateurs

On peut aussi vouloir comparer certains indicateurs (moyenne, médiane) d’une variable quantitative selon les modalités d’une variable qualitative. Si on reprend l’exemple précédent, on peut calculer la moyenne d’âge pour ceux qui pratiquent un sport et pour ceux qui n’en pratiquent pas.

Une première méthode pour cela est d’extraire de notre population autant de sous-populations qu’il y a de modalités dans la variable qualitative. On peut le faire notamment avec la fonction `filter` du package `dplyr`¹.

On commence par charger `dplyr` (en l’ayant préalablement installé) :

¹Le package en question est présenté en détail dans la partie 10.

```
library(dplyr)
```

Puis on applique `filter` pour créer deux sous-populations, stockées dans deux nouveaux tableaux de données :

```
d_sport <- filter(d, sport == "Oui")
d_nonsport <- filter(d, sport == "Non")
```

On peut ensuite utiliser ces deux nouveaux tableaux de données comme on en a l'habitude, et calculer les deux moyennes d'âge :

```
mean(d_sport$age)
#> [1] 40.92531
```

```
mean(d_nonsport$age)
#> [1] 52.25137
```

Une autre possibilité est d'utiliser la fonction `tapply`, qui prend en paramètre une variable quantitative, une variable qualitative et une fonction, puis applique automatiquement la fonction aux valeurs de la variables quantitative pour chaque niveau de la variable qualitative :

```
tapply(d$age, d$sport, mean)
#>      Non      Oui
#> 52.25137 40.92531
```

On verra dans la partie 10 d'autres méthodes basées sur `dplyr` pour effectuer ce genre d'opérations.

4.2.3 Tests statistiques

Un des tests les plus connus est le test du *t* de Student, qui permet de tester si les moyennes de deux sous-populations peuvent être considérées comme différentes (compte tenu des fluctuations aléatoires provenant du biais d'échantillonnage).

Un test *t* s'effectue à l'aide de la fonction `t.test`. Ainsi, on peut tester l'hypothèse d'égalité des âges moyens selon la pratique ou non d'un sport avec la commande suivante :

```
t.test(d$age ~ d$sport)
#>
#> Welch Two Sample t-test
#>
#> data: d$age by d$sport
#> t = 15.503, df = 1600.4, p-value < 2.2e-16
#> alternative hypothesis: true difference in means between group Non and group Oui is not equal to 0
#> 95 percent confidence interval:
#> 9.893117 12.759002
#> sample estimates:
#> mean in group Non mean in group Oui
#> 52.25137 40.92531
```

Le résultat du test est significatif, avec un p extrêmement petit, et on peut rejeter l'hypothèse nulle d'égalité des moyennes des deux groupes. Le test nous donne même un intervalle de confiance à 95% pour la valeur de la différence entre les deux moyennes.

Nous sommes cependant allés un peu vite, et avons négligé le fait que le test t s'applique normalement à des distributions normales. On peut se faire un premier aperçu visuel de cette normalité en traçant les histogrammes des deux répartitions :

```
hist(d_sport$age)
```



```
hist(d_nonsport$age)
```



Si l'âge dans le groupe des non sportifs se rapproche d'une distribution normale, celui des sportifs en semble assez éloigné, notamment du fait de la limite d'âge à 18 ans imposée par construction de l'enquête.

On peut tester cette normalité à l'aide du test de Shapiro-Wilk et de la fonction `shapiro.test` :

```
shapiro.test(d_sport$age)
#>
#> Shapiro-Wilk normality test
#>
#> data: d_sport$age
#> W = 0.96203, p-value = 9.734e-13
```

```
shapiro.test(d_nonsport$age)
#>
#> Shapiro-Wilk normality test
#>
#> data: d_nonsport$age
#> W = 0.98844, p-value = 1.654e-08
```

Le test est significatif dans les deux cas et rejette l'hypothèse d'une normalité des deux distributions.

Dans ce cas on peut faire appel à un test non-paramétrique, qui ne fait donc pas d'hypothèses sur les lois de distribution des variables testées, en l'occurrence le test des rangs de Wilcoxon, à l'aide de la fonction `wilcox.test` :

```
wilcox.test(d$age ~ d$sport)
#>
```

```
#> Wilcoxon rank sum test with continuity correction
#>
#> data: d$age by d$sport
#> W = 640577, p-value < 2.2e-16
#> alternative hypothesis: true location shift is not equal to 0
```

La valeur p étant à nouveau extrêmement petite, on peut rejeter l'hypothèse d'indépendance et considérer que les distributions des âges dans les deux sous-populations sont différentes.

4.3 Croisement de deux variables quantitatives

Le jeu de données `hdv2003` comportant assez peu de variables quantitatives, on va s'intéresser maintenant à un autre jeu de données comportant des informations du recensement de la population de 2018. On le charge avec :

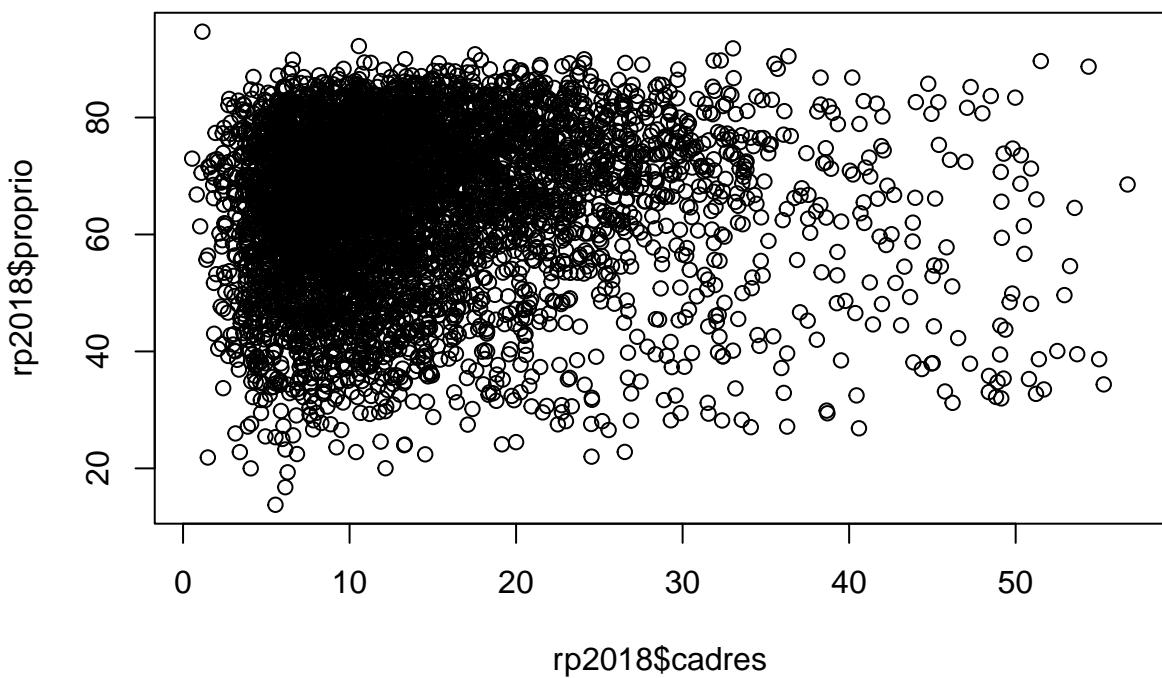
```
data(rp2018)
```

Un nouveau tableau de données `rp2018` devrait apparaître dans votre environnement. Celui-ci comprend les 5417 communes françaises de plus de 2000 habitants, et une soixantaine de variables telles que le département, la population, le taux de chômage, etc. Pour une description plus complète et une liste des variables, voir section [A.3.2.3](#).

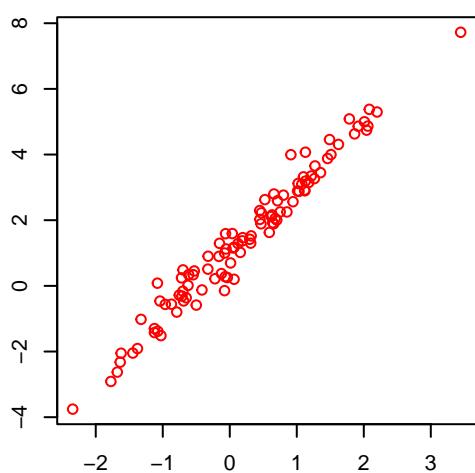
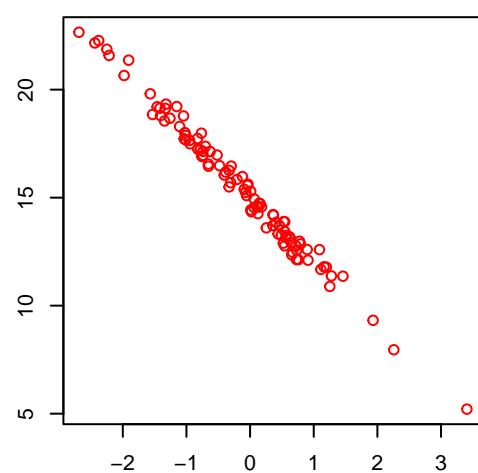
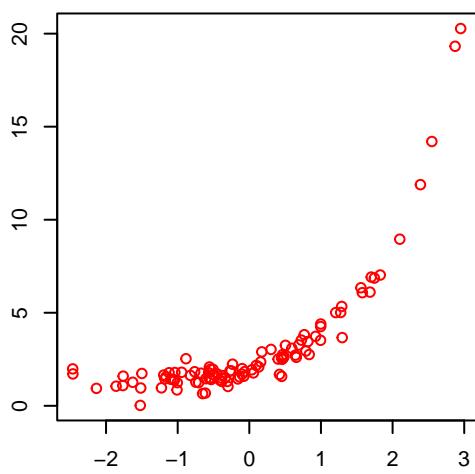
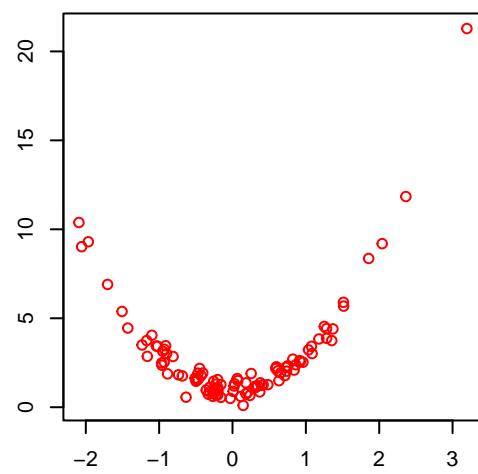
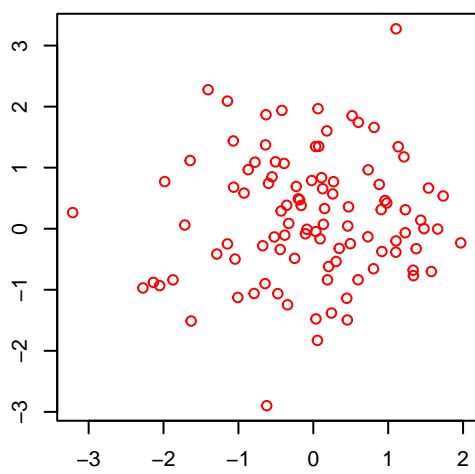
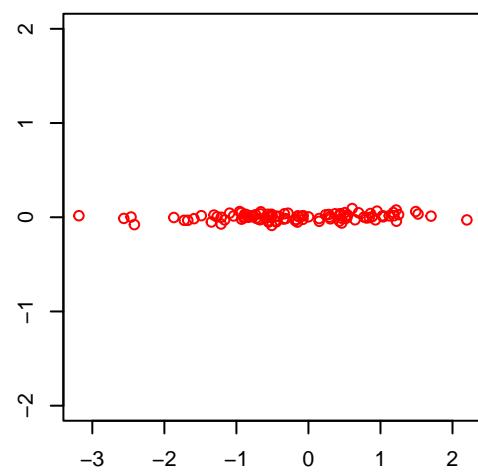
4.3.1 Représentation graphique

Quand on croise deux variables quantitatives, l'idéal est de faire une représentation graphique sous forme de nuage de points à l'aide de la fonction `plot`. On va représenter le croisement entre le pourcentage de cadres et le pourcentage de propriétaires dans la commune :

```
plot(rp2018$cadres, rp2018$proprio)
```

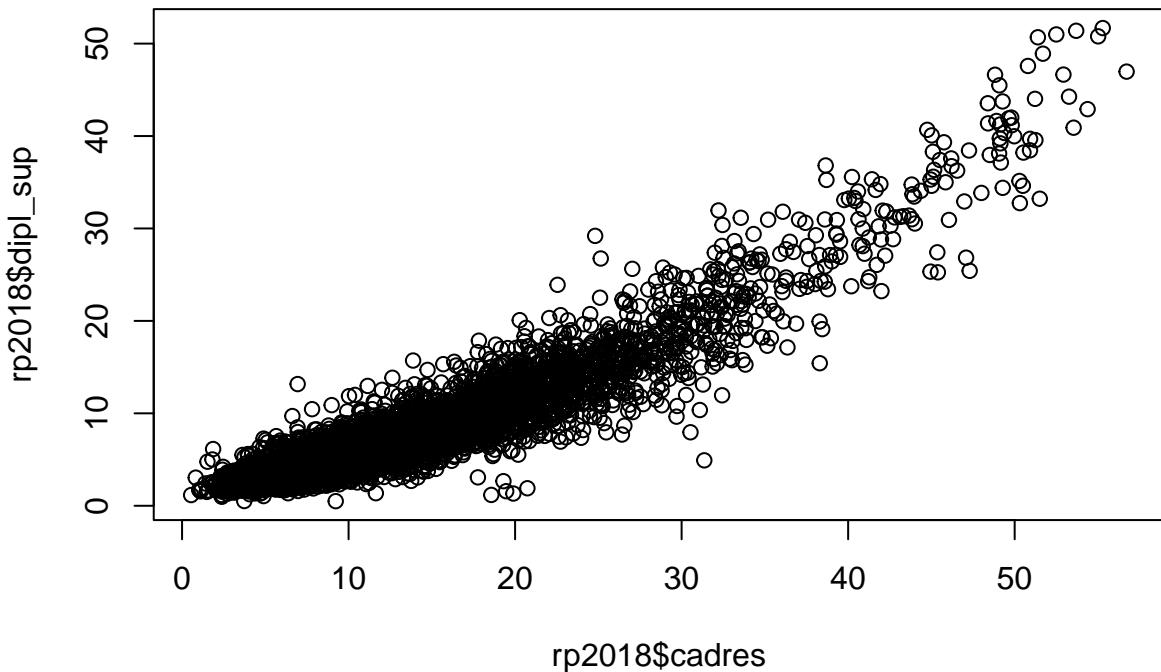


Une représentation graphique est l'idéal pour visualiser l'existence d'un lien entre les deux variables. Voici quelques exemples d'interprétation :

Dépendance linéaire positive**Dépendance linéaire négative****Dépendance non-linéaire monotone****Dépendance non-linéaire non monotone****Indépendance****Indépendance**

Dans ce premier graphique générée sur nos données, il semble difficile de mettre en évidence une relation de dépendance. Si par contre on croise le pourcentage de cadres et celui de diplômés de niveau Bac+5 ou plus, on obtient une belle relation de dépendance linéaire.

```
plot(rp2018$cadres, rp2018$dipl_sup)
```



4.3.2 Calcul d'indicateurs

En plus d'une représentation graphique, on peut calculer certains indicateurs permettant de mesurer le degré d'association de deux variables quantitatives.

4.3.2.1 Corrélation linéaire (Pearson)

La corrélation est une mesure du lien d'association *linéaire* entre deux variables quantitatives. Sa valeur varie entre -1 et 1. Si la corrélation vaut -1, il s'agit d'une association linéaire négative parfaite. Si elle vaut 1, il s'agit d'une association linéaire positive parfaite. Si elle vaut 0, il n'y a aucune association linéaire entre les variables.

On la calcule dans R à l'aide de la fonction `cor`.

Ainsi la corrélation entre le pourcentage de cadres et celui de diplômés du supérieur vaut :

```
cor(rp2018$cadres, rp2018$dipl_sup)
#> [1] 0.9291504
```

Ce qui est extrêmement fort. Il y a donc un lien linéaire et positif entre les deux variables (quand la valeur de l'une augmente, la valeur de l'autre augmente également).

À l'inverse, la corrélation entre le pourcentage de cadres et le pourcentage de propriétaires vaut :

```
cor(rp2018$cadres, rp2018$proprio)
#> [1] 0.06425958
```

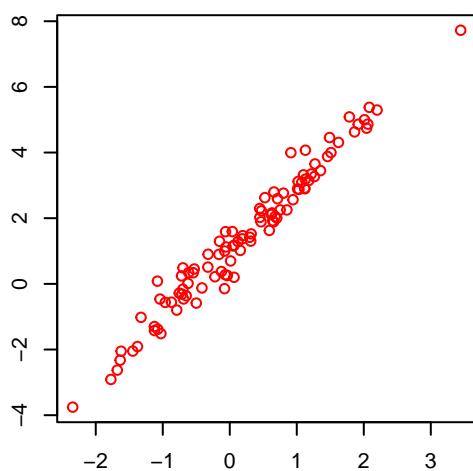
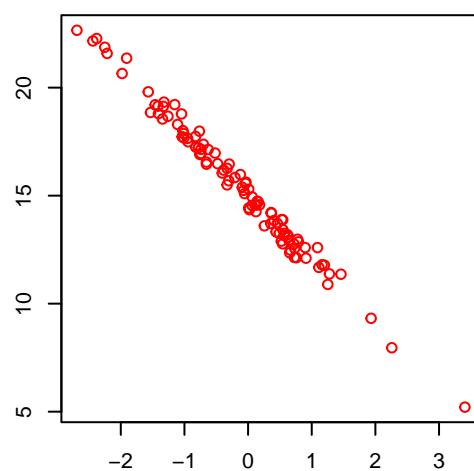
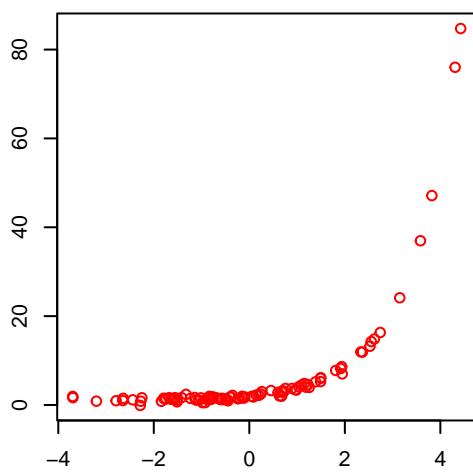
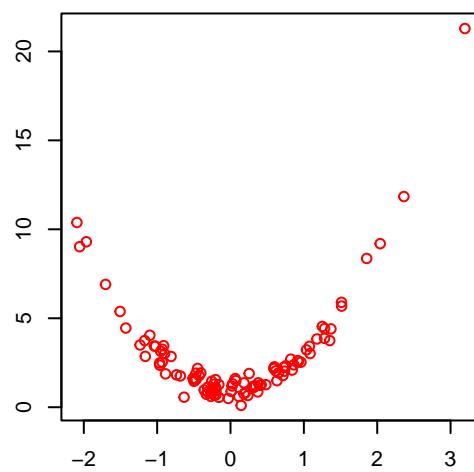
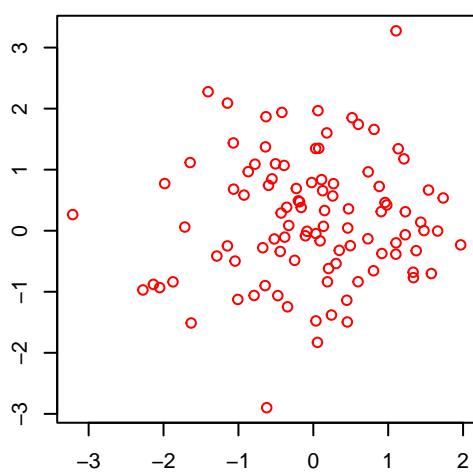
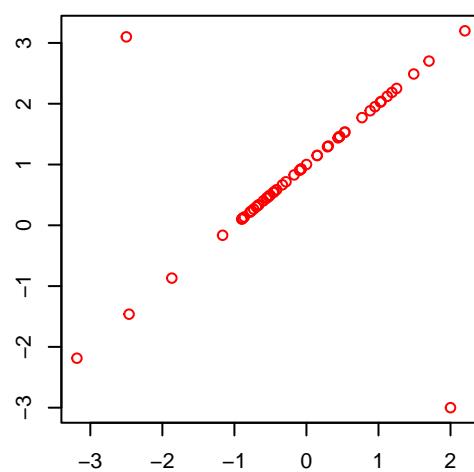
Ce qui indique, pour nos données, une absence de liaison linéaire entre les deux variables.

4.3.2.2 Corrélation des rangs (Spearman)

Le coefficient de corrélation de Pearson ci-dessus fait une hypothèse forte sur les données : elles doivent être liées par une association linéaire. Quand ça n'est pas le cas mais qu'on est en présence d'une association monotone, on peut utiliser un autre coefficient, le coefficient de corrélation des rangs de Spearman.

Plutôt que de se baser sur les valeurs des variables, cette corrélation va se baser sur leurs rangs, c'est-à-dire sur leur position parmi les différentes valeurs prises par les variables.

Ainsi, si la valeur la plus basse de la première variable est associée à la valeur la plus basse de la deuxième, et ainsi de suite jusqu'à la valeur la plus haute, on obtiendra une corrélation de 1. Si la valeur la plus forte de la première variable est associée à la valeur la plus faible de la seconde, et ainsi de suite, et que la valeur la plus faible de la première est associée à la plus forte de la deuxième, on obtiendra une corrélation de -1. Si les rangs sont "mélangés", sans rapports entre eux, on obtiendra une corrélation autour de 0.

Pearson : 0.98 – Spearman : 0.98**Pearson : -0.99 – Spearman : -0.99****Pearson : 0.66 – Spearman : 0.88****Pearson : 0.24 – Spearman : 0.06****Pearson : 0.06 – Spearman : 0.02****Pearson : 0.56 – Spearman : 0.79**

La corrélation des rangs a aussi pour avantage d'être moins sensibles aux valeurs extrêmes ou aux points isolés. On dit qu'elle est plus "robuste".

Pour calculer une corrélation de Spearman, on utilise la fonction `cor` mais avec l'argument `method = "spearman"`.

```
cor(rp2018$cadres, rp2018$dipl_sup, method = "spearman")
#> [1] 0.8986656
```

4.3.3 Régression linéaire

Quand on est en présence d'une association linéaire entre deux variables, on peut vouloir faire la régression linéaire d'une des variables sur l'autres.

Une régression linéaire simple se fait à l'aide de la fonction `lm` :

```
lm(rp2018$cadres ~ rp2018$dipl_sup)
#>
#> Call:
#> lm(formula = rp2018$cadres ~ rp2018$dipl_sup)
#>
#> Coefficients:
#>   (Intercept)  rp2018$dipl_sup
#>           3.578          1.256
```

 On retrouve avec `lm` la syntaxe "formule" déjà rencontrée avec `boxplot`. Elle permet ici de spécifier des modèles de régression : la variable dépendante se place à gauche du `~`, et la variable indépendante à droite. Si on souhaite faire une régression multiple avec plusieurs variables indépendantes, on aura une formule du type `dep ~ indep1 + indep2`. Il est également possible de spécifier des termes plus complexes, des interactions, etc.

`lm` nous renvoie par défaut les coefficients de la droite de régression :

- l'ordonnée à l'origine (`Intercept`) vaut 3.578
- le coefficient associé à `dipl_sup` vaut 1.256

Pour des résultats plus détaillés, on peut stocker le résultat de la régression dans un objet et lui appliquer la fonction `summary`.

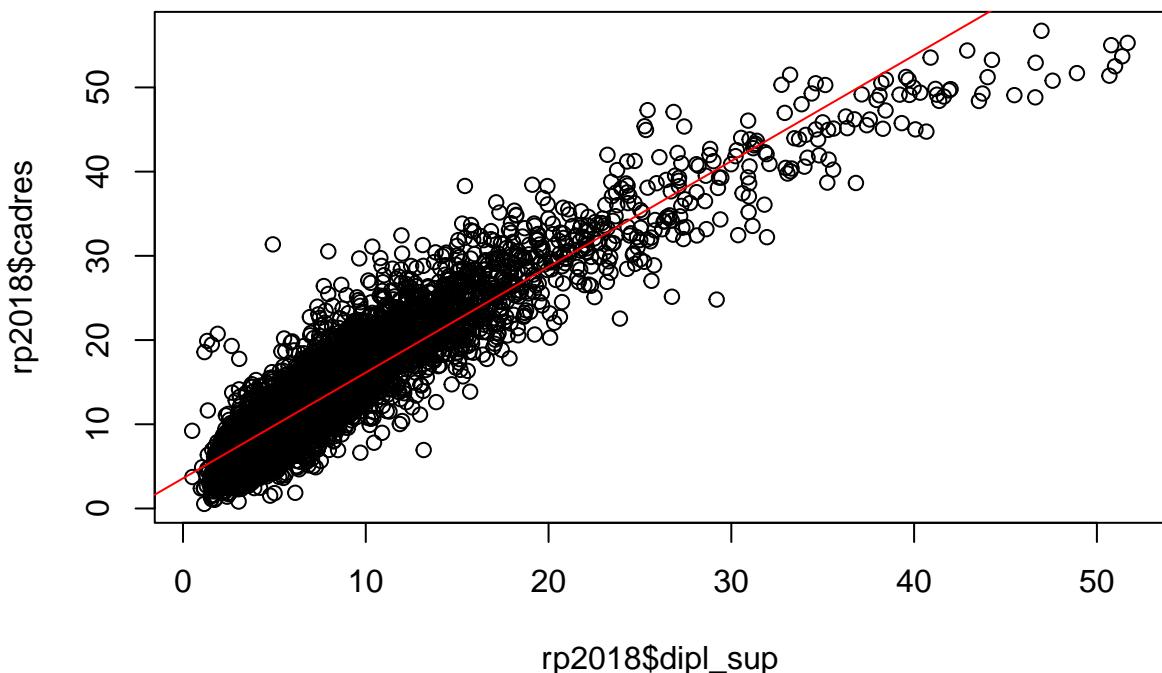
```
reg <- lm(rp2018$cadres ~ rp2018$dipl_sup)
summary(reg)
#>
#> Call:
#> lm(formula = rp2018$cadres ~ rp2018$dipl_sup)
#>
#> Residuals:
#>   Min     1Q   Median     3Q    Max 
#> -15.8329 -1.9320 -0.2953  1.7184 21.6096 
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)    
#> (Intercept) 3.57796   0.06925 51.67 <2e-16 ***
#>
```

```
#> rp2018$dipl_sup  1.25580    0.00679  184.94   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.116 on 5415 degrees of freedom
#> Multiple R-squared:  0.8633, Adjusted R-squared:  0.8633
#> F-statistic: 3.42e+04 on 1 and 5415 DF,  p-value: < 2.2e-16
```

Ces résultats montrent notamment que les coefficients sont significativement différents de 0. La part de cadres augmente donc bien avec celle de diplômés du supérieur.

On peut enfin représenter la droite de régression sur notre nuage de points à l'aide de la fonction `abline`.

```
plot(rp2018$dipl_sup, rp2018$cadres)
abline(reg, col = "red")
```



4.4 Exercices

Exercice 1

Dans le jeu de données `hdv2003`, faire le tableau croisé entre la catégorie socio-professionnelle (variable `qualif`) et le fait de croire ou non en l'existence des classes sociales (variable `c1so`). Identifier la variable indépendante et la variable dépendante, et calculer les pourcentages ligne ou colonne. Interpréter le résultat.

Faire un test du χ^2 . Peut-on rejeter l'hypothèse d'indépendance ?

Représenter ce tableau croisé sous la forme d'un `mosaicplot` en colorant les cases selon les résidus du test du χ^2 .

Exercice 2

Toujours sur le jeu de données `hdv2003`, faire le boxplot qui croise le nombre d'heures passées devant la télévision (variable `heures.tv`) avec le statut d'occupation (variable `occup`).

Calculer la durée moyenne devant la télévision en fonction du statut d'occupation à l'aide de `tapply`.

Exercice 3

Sur le jeu de données `rp2018`, représenter le nuage de points croisant le pourcentage de personnes sans diplôme (variable `dipl_aucun`) et le pourcentage de propriétaires (variable `proprio`).

Calculer le coefficient de corrélation linéaire correspondant.

Chapitre 5

Organiser ses scripts

On l'a vu, le script est l'élément central de toute analyse dans R. C'est lui qui contient l'ensemble des opérations constitutives d'une analyse, dans leur ordre d'exécution : chargement des données, recodages, manipulations, analyses, exports de résultats, etc.

Une conséquence est qu'un script peut rapidement devenir très long, et on peut finir par s'y perdre. Il est donc nécessaire d'organiser son travail pour pouvoir se retrouver facilement parmi les différentes étapes d'un projet d'analyse.

5.1 Les projets dans RStudio

La notion de projet est une fonctionnalité très pratique de RStudio, qui permet d'organiser son travail et de faciliter l'accès à l'ensemble des fichiers constitutifs d'une analyse (données, scripts, documentation, etc.).

En pratique, un projet est un dossier que vous avez créé où bon vous semble sur votre disque dur, et dans lequel vous regrouperez tous les fichiers en question. Utiliser des projets procure plusieurs avantages :

- RStudio lance automatiquement R dans le dossier du projet et facilite ainsi grandement l'accès aux fichiers de données à importer (plus besoin de taper le chemin d'accès complet). De même, si vous déplacez votre dossier sur votre disque, le projet continuera à fonctionner.
- L'onglet *Files* de la zone en bas à droite de l'interface de RStudio vous permet de naviguer facilement dans les fichiers de votre projet.
- Vous pouvez très facilement passer d'un projet à l'autre si vous travaillez sur plusieurs jeux de données en parallèle.

Pour créer un projet, il faut aller dans le menu *File* puis sélectionner *New project*.

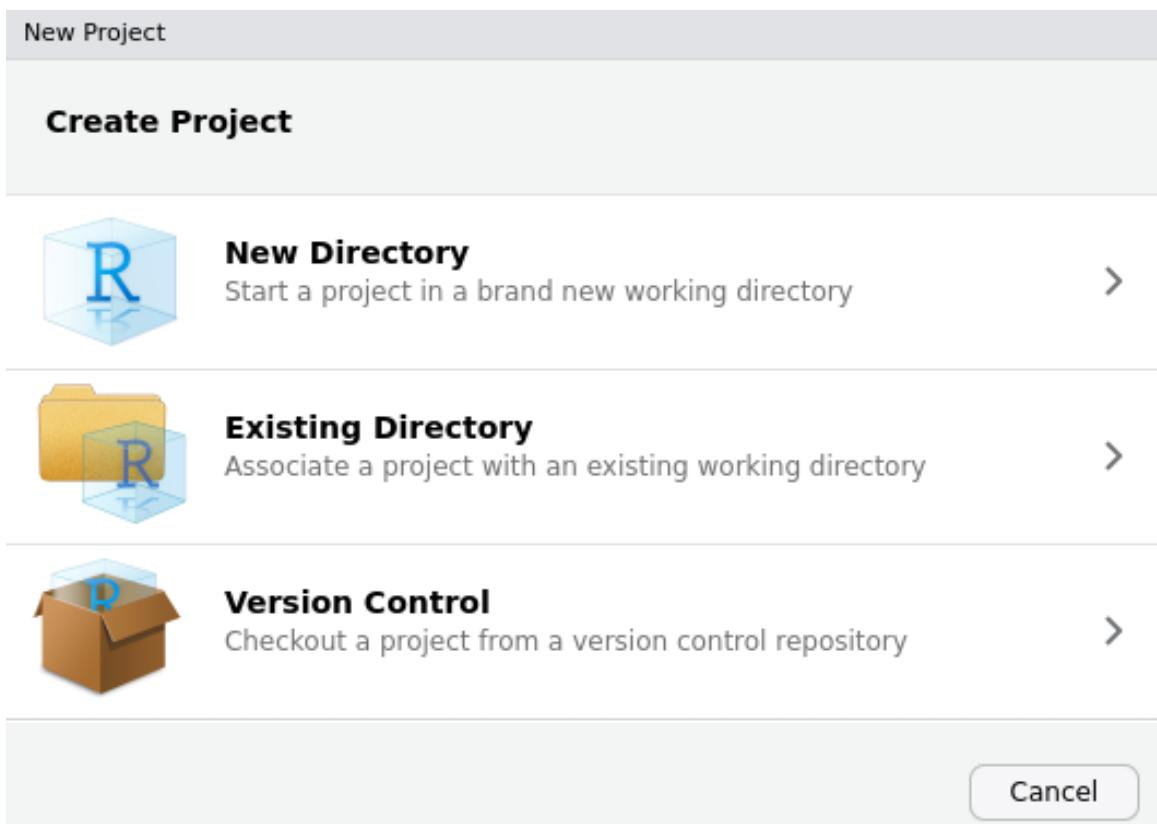


Figure 5.1: Création d'un nouveau projet

Selon que le dossier du projet existe déjà ou pas, on choisira *Existing directory* ou *New directory*. L'étape d'après consiste à créer ou sélectionner le dossier, puis on n'a plus qu'à cliquer sur *Create project*.

À la création du projet, et chaque fois que vous l'ouvrirez, une nouvelle session R est lancée dans la fenêtre *Console* avec le dossier du projet comme répertoire de travail, et l'onglet *Files* affiche les fichiers contenus dans ce dossier.

Une fois le projet créé, son nom est affiché dans un petit menu déroulant en haut à droite de l'interface de RStudio (menu qui permet de passer facilement d'un projet à un autre).

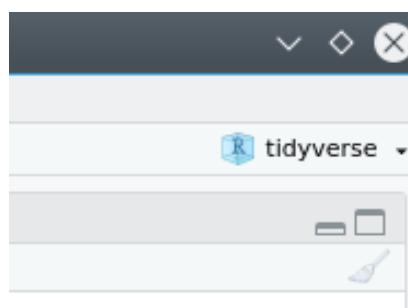


Figure 5.2: Menu projets



Si vous ne retrouvez pas le nom du projet dans ce menu, vous pouvez l'ouvrir en sélectionnant *File* puis *Open Project...* et en allant sélectionner le fichier *.Rproj* qui se trouve dans le dossier du projet à ouvrir.

5.2 Créer des sections dans un script

Lorsqu'un script est long, RStudio permet de créer des "sections" facilitant la navigation.

Pour créer une section, il suffit de faire suivre une ligne de commentaires par plusieurs tirets -, comme ceci :

```
## Titre de la section -----
```

Le nombre de tirets n'a pas d'importance, il doit juste y'en avoir plus de quatre. RStudio affiche alors dans la marge de gauche du script un petit triangle noir qui permet de replier ou déplier le contenu de la section :

```
5
6 #> ## Chargement des données -----
7
8 data(hdv2003)
9
10 #> ## Analyse -----
11
12 hist(hdv2003$age)
```

Figure 5.3: Section de script dépliée

```
5
6 #> ## Chargement des données ⏴
7
8 data(hdv2003)
9
10 #> ## Analyse -----
11
12 hist(hdv2003$age)
```

Figure 5.4: Section de script repliée

De plus, en cliquant sur l'icône *Show document outline* (la plus à droite de la barre d'outils de la fenêtre du script), ou en utilisant le raccourci clavier **Ctrl+Maj+0**, RStudio affiche une “table des matières” automatiquement mise à jour qui liste les sections existantes et permet de naviguer facilement dans le script :



Figure 5.5: Liste dynamique des sections

5.3 Répartir son travail entre plusieurs scripts

Si le script devient très long, les sections peuvent ne plus être suffisantes. De plus, il est souvent intéressant d'isoler certaines parties d'un script, par exemple pour pouvoir les mutualiser. On peut alors répartir les étapes d'une analyse entre plusieurs scripts.

Un exemple courant concerne les recodages et la manipulation des données. Il est fréquent, au cours d'une analyse, de calculer de nouvelles variables, recoder des variables qualitatives existantes, etc. Il peut alors être intéressant de regrouper tous ces recodages dans un script à part (nommé, par exemple, `recodages.R`). Ce fichier contient alors l'ensemble des recodages “validés”, ceux qu'on a testé et qu'on sait vouloir conserver.

Pour exécuter ces recodages, on peut évidemment ouvrir le script `recodages.R` dans RStudio et lancer l'ensemble du code qu'il contient. Mais une méthode plus pratique est d'utiliser la fonction `source` : celle-ci prend en paramètre un nom de fichier `.R`, et quand on l'exécute elle va exécuter l'ensemble du code contenu dans ce fichier.

Ainsi, un début de script `analyse.R` pourra ressembler à ceci :

```
# Analyse des données Histoire de vie 2003

# Chargement des extensions et des données -----
library(questionr)

data(hdv2003)
source("recodages.R")

# Analyse de l'âge ----

hist(hdv2003$age)

(...)
```

L'avantage principal est qu'on peut à tout moment revenir à nos données d'origine et aux recodages “validés” simplement en exécutant les deux lignes :

```
data(hdv2003)
source("recodages.R")
```

L'autre avantage est qu'on peut répartir nos analyses entre différents scripts, et conserver ces deux lignes en haut de chaque script, ce qui permet de “mutualiser” les recodages validés. On pourrait ainsi créer un deuxième script `analyse_qualif.R` qui pourrait ressembler à ceci :

```
# Analyse des données Histoire de vie 2003 - Qualifications

# Chargement des extensions et des données -----
library(questionr)

data(hdv2003)
source("recodages.R")

# Analyse des qualifications ----

freq(hdv2003$qualif)

(...)
```

On peut évidemment répartir les recodages entre plusieurs fichiers et faire appel à autant de `source` que l'on souhaite.



Cette organisation calcule l'ensemble des recodages à chaque début de script. C'est intéressant et pratique pour des données de taille raisonnable, mais pour des fichiers plus volumineux les calculs peuvent être trop longs. Dans ce cas il est préférable de créer des scripts dédiés qui chargent les données source,

effectuent calculs et recodages, et enregistrent les données résultantes dans un fichier de données (voir le chapitre sur l'import/export de données). Et c'est ce fichier résultat qui sera chargé par les scripts d'analyse.

5.4 Désactiver la sauvegarde de l'espace de travail

Vous avez sans doute remarqué qu'au moment de quitter RStudio, une boîte de dialogue semblable à celle-ci s'affiche parfois :

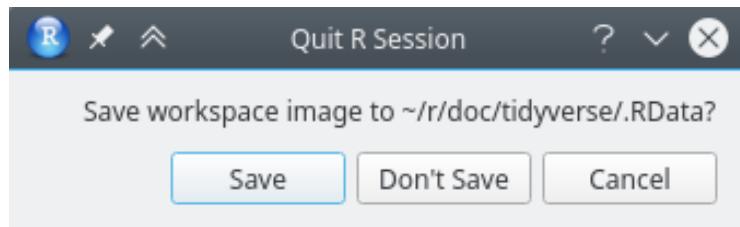


Figure 5.6: Dialogue d'enregistrement de l'espace de travail

Et il est bien difficile de comprendre de quoi cela parle.

Il s'agit en fait d'une fonctionnalité de R lui-même qui propose d'enregistrer notre espace de travail (*workspace*), c'est-à-dire l'ensemble des objets qui existent actuellement dans notre environnement, dans un fichier nommé `.RData`. La prochaine fois que R est lancé dans le même dossier (par exemple à la réouverture du projet), s'il trouve un fichier `.RData` il va le lire automatiquement et restaurer l'ensemble des objets dans l'état où ils étaient.

Ceci peut sembler pratique, mais c'est en fait une mauvaise idée, pour deux raisons :

- on peut se retrouver avec des objets dont on ne sait plus d'où ils viennent et comment ils ont été calculés
- cette manière de faire casse la logique principale de R, qui est que c'est le script qui est central, et que c'est lui qui retrace toutes les étapes de notre analyse et permet de les reproduire

Il est donc *fortement recommandé*, juste après l'installation de RStudio, de désactiver cette fonctionnalité. Pour cela, aller dans le menu *Tools*, puis *Global Options*, et s'assurer que :

- la case *Restore .RData into workspace at startup* est décochée
- le champ *Save workspace to .RData on exit* vaut *Never*



Figure 5.7: Options d'enregistrement de l'espace de travail

Partie II

Introduction au tidyverse

Chapitre 6

Le tidyverse

6.1 Extensions

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 d’extensions conçues pour travailler ensemble et basées sur une philosophie commune.

Elles abordent un très grand nombre d’opérations courantes dans R (la liste n’est pas exhaustive) :

- visualisation
- manipulation des tableaux de données
- import/export de données
- manipulation de variables
- extraction de données du Web
- programmation

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](#).

6.2 Installation

`tidyverse` est également le nom d’une extension qu’on peut installer de manière classique, soit via le bouton *Install* de l’onglet *Packages* de RStudio, soit en utilisant la commande :

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

Cette commande va en fait installer plusieurs extensions qui constituent le “coeur” du *tidyverse*, à savoir :

- `ggplot2` (visualisation)
- `dplyr` (manipulation des données)
- `tidyr` (remise en forme des données)
- `purrr` (programmation)
- `readr` (importation de données)
- `tibble` (tableaux de données)
- `forcats` (variables qualitatives)
- `stringr` (chaînes de caractères)



Figure 6.1: Packages de l'extension tidyverse

De la même manière, charger l'extension avec :

```
library(tidyverse)
```

Chargera l'ensemble des extensions précédentes.

Il existe d'autres extensions qui font partie du *tidyverse* mais qui doivent être chargées explicitement, comme par exemple **readxl** (pour l'importation de données depuis des fichiers Excel). La liste complète se trouve sur [le site officiel du tidyverse](#).

Ce document est basé sur les versions d'extension suivantes :

```
#> ggplot2 3.3.5    purrr  0.3.4
#> tibble   3.1.6    dplyr   1.0.7
#> tidyr    1.1.4    stringr 1.4.0
#> readr    2.1.1   forcats 0.5.1
```

6.3 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

On verra plus précisément dans la section 12 comment définir et rendre des données *tidy* avec l'extension **tidyr**.

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

6.4 tibbles

Une autre particularité du *tidyverse* est que ces extensions travaillent avec des tableaux de données au format *tibble*, qui est une évolution plus moderne du classique *data frame* du R de base. Ce format est fourni et géré par l'extension du même nom (**tibble**), qui fait partie du cœur du *tidyverse*. La plupart des fonctions des extensions du *tidyverse* acceptent des *data frames* en entrée, mais retournent un objet de classe **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...)¹
- 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 colonnes²
- 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*. On peut ainsi facilement convertir un *data frame* en *tibble* avec **as_tibble** :

```
as_tibble(mtcars)
#> # A tibble: 32 x 11
#>   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <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
```

Si le *data frame* d'origine a des *rownames*, on peut d'abord les convertir en colonnes avec **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>
#> 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
```

¹Quand on veut utiliser des noms de ce type, on doit les entourer avec des *backticks* (`)

²Dans R de base, si une table d contient une colonne **qualif**, d\$qual retournera cette colonne.

À l'inverse, on peut à tout moment convertir un tibble en *data frame* avec `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
#> [ reached 'max' / getOption("max.print") -- omitted 16 rows ]
```

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

```
column_to_rownames(as.data.frame(d))
#> #>      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
#> [ reached 'max' / getOption("max.print") -- omitted 14 rows ]
```



Les deux fonctions `column_to_rownames` et `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.

Chapitre 7

Importer et exporter des données

R n'est pas prévu pour la saisie de données, mais il bénéficie de nombreuses fonctions et packages permettant l'import de données depuis un grand nombre de formats. Seuls les plus courants seront abordés ici.

Il est très vivement conseillé de travailler avec les projets de RStudio pour faciliter l'accès aux fichiers et pouvoir regrouper l'ensemble des éléments d'une analyse dans un dossier (voir partie 5.1).



Les projets permettent notamment de ne pas avoir à spécifier un chemin complet vers un fichier (sous Windows, quelque chose du genre `C:\\\\Users\\\\toto\\\\Documents\\\\quanti\\\\projet\\\\data\\\\donnees.xls`) mais un chemin relatif au dossier du projet (juste `donnees.xls` si le fichier se trouve à la racine du projet, `data/donnees.xls` s'il se trouve dans un sous-dossier `data`, etc.)

7.1 Import de fichiers textes

L'extension `readr`, qui fait partie du *tidyverse*, permet l'importation de fichiers texte, notamment au format CSV (*Comma separated values*), format standard pour l'échange de données tabulaires entre logiciels.

Cette extension fait partie du "coeur" du *tidyverse*, elle est donc automatiquement chargée avec :

```
library(tidyverse)
```

Si votre fichier CSV suit un format CSV standard (c'est le cas s'il a été exporté depuis LibreOffice par exemple), avec des champs séparés par des virgules, vous pouvez utiliser la fonction `read_csv` en lui passant en argument le nom du fichier :

```
d <- read_csv("fichier.csv")
```

Si votre fichier vient d'Excel, avec des valeurs séparées par des points virgule, utilisez la fonction `read_csv2` :

```
d <- read_csv2("fichier.csv")
```

Dans la même famille de fonction, `read_tsv` permet d'importer des fichiers dont les valeurs sont séparées par des tabulations, et `read_delim` des fichiers délimités par un séparateur indiqué en argument.

Chaque fonction dispose de plusieurs arguments, parmi lesquels :

- `col_names` indique si la première ligne contient le nom des colonnes (TRUE par défaut)

- `col_types` permet de spécifier manuellement le type des colonnes si `readr` ne les identifie pas correctement
- `na` est un vecteur de chaînes de caractères indiquant les valeurs devant être considérées comme manquantes. Ce vecteur vaut `c("", "NA")` par défaut

Il peut arriver, notamment sous Windows, que l'encodage des caractères accentués ne soit pas correct au moment de l'importation. On peut alors spécifier manuellement l'encodage du fichier importé à l'aide de l'option `locale`. Par exemple, si l'on est sous Mac ou Linux et que le fichier a été créé sous Windows, il est possible qu'il soit encodé au format iso-8859-1. On peut alors l'importer avec :

```
d <- read_csv("fichier.csv", locale = locale(encoding = "ISO-8859-1"))
```

À l'inverse, si vous importez un fichier sous Windows et que les accents ne sont pas affichés correctement, il est sans doute encodé en UTF-8 :

```
d <- read_csv("fichier.csv", locale = locale(encoding = "UTF-8"))
```

Pour plus d'informations sur ces fonctions, voir [le site de l'extension readr](#).



À noter que si vous souhaitez importer des fichiers textes très volumineux le plus rapidement possible, la fonction `fread` de l'extension `data.table` est plus rapide que `read_csv`.

7.1.1 Interface interactive d'import de fichiers

RStudio propose une interface permettant d'importer un fichier de données de manière interactive. Pour y accéder, dans l'onglet *Environment*, cliquez sur le bouton *Import Dataset* :

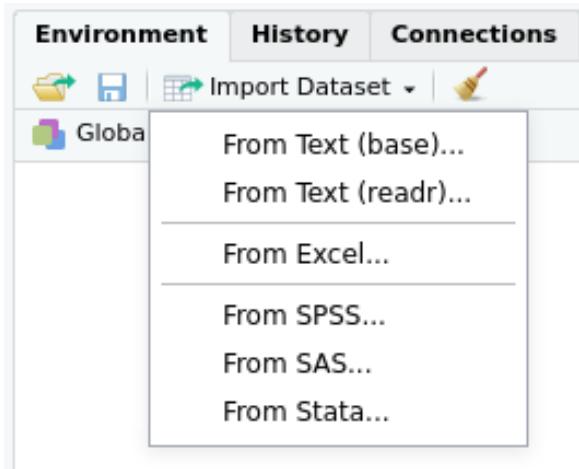


Figure 7.1: Menu *Import Dataset*

Sélectionnez *From Text (readr)...*. Une nouvelle fenêtre s'affiche :

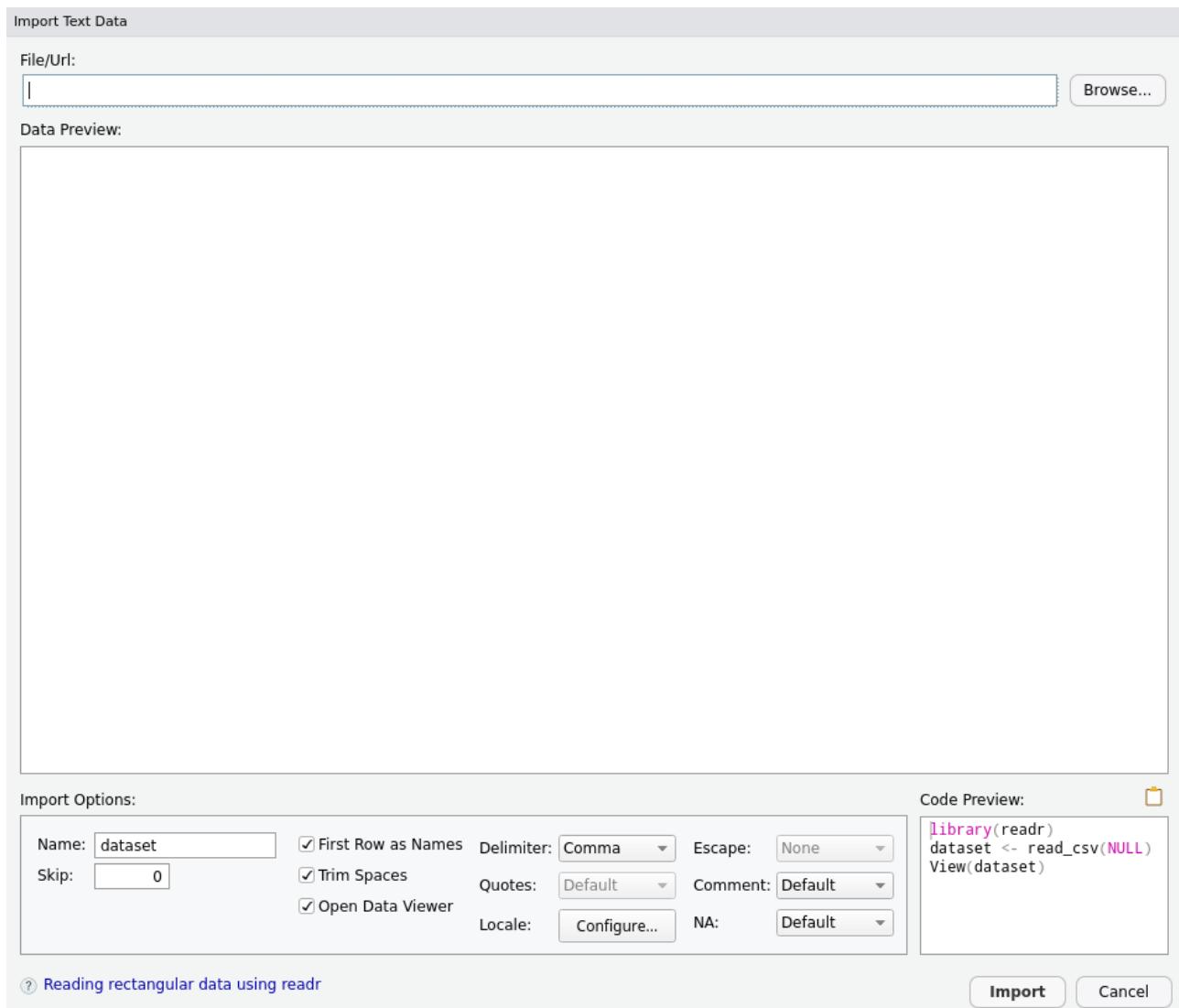


Figure 7.2: Dialogue d'importation

Il vous suffit d'indiquer le fichier à importer dans le champ *File/URL* tout en haut (vous pouvez même indiquer un lien vers un fichier distant via HTTP). Un aperçu s'ouvre dans la partie *Data Preview* et vous permet de vérifier si l'import est correct :

Import Text Data

File/Url:

Browse...

Data Preview:

id (integer)	age (integer)	sexe (character)	nivetud (character)	poids (double)	occup (character)	qualif (character)	freres
1	28	Femme	Enseignement supérieur y compris technique superi...	2634.3982	Exerce une profession	Employe	
2	23	Femme	NA	9738.3958	Etudiant, eleve	NA	
3	59	Homme	Derniere annee d'etudes primaires	3994.1025	Exerce une profession	Technicien	
4	34	Homme	Enseignement supérieur y compris technique superi...	5731.6615	Exerce une profession	Technicien	
5	71	Femme	Derniere annee d'etudes primaires	4329.0940	Retraite	Employe	
6	35	Femme	Enseignement technique ou professionnel court	8674.6994	Exerce une profession	Employe	
7	60	Femme	Derniere annee d'etudes primaires	6165.8035	Au foyer	Ouvrier qualifie	
8	47	Homme	Enseignement technique ou professionnel court	12891.6408	Exerce une profession	Ouvrier qualifie	
9	20	Femme	NA	7808.8721	Etudiant, eleve	NA	
10	28	Homme	Enseignement technique ou professionnel long	2277.1605	Exerce une profession	Autre	
11	65	Femme	Enseignement supérieur y compris technique superi...	704.3227	Retraite	Employe	
12	47	Homme	Zeme cycle	6697.8682	Exerce une profession	Ouvrier qualifie	
13	63	Femme	Derniere annee d'etudes primaires	7118.4659	Retraite	Employe	
14	67	Femme	Enseignement technique ou professionnel court	586.7714	Exerce une profession	NA	
15	76	Femme	A arrete ses etudes, avant la dernière annee d'étud...	11042.0774	Retraite	NA	
16	49	Femme	Enseignement technique ou professionnel court	9958.2287	Exerce une profession	Employe	
17	62	Homme	Enseignement supérieur y compris technique superi...	4836.1393	Retraite	Cadre	
18	20	Femme	NA	1551.4846	Etudiant, eleve	NA	
19	70	Homme	Derniere annee d'etudes primaires	3141.1572	Retraite	Ouvrier specialise	
20	39	Femme	Enseignement technique ou professionnel court	27195.8378	Exerce une profession	Ouvrier qualifie	

Previewing first 50 entries.

Import Options:

Name: <input type="text" value="hdv2003"/>	<input checked="" type="checkbox"/> First Row as Names	Delimiter: <input type="button" value="Comma"/>	Escape: <input type="button" value="None"/>
Skip: <input type="text" value="0"/>	<input checked="" type="checkbox"/> Trim Spaces	Quotes: <input type="button" value="Default"/>	Comment: <input type="button" value="Default"/>
	<input checked="" type="checkbox"/> Open Data Viewer	Locale: <input type="button" value="Configure..."/>	NA: <input type="button" value="Default"/>

Code Preview:

```
library(readr)
hdv2003 <- read_csv("hdv2003.csv")
View(hdv2003)
```

[? Reading rectangular data using readr](#) Import Cancel

Figure 7.3: Exemple de dialogue d’importation

Vous pouvez modifier les options d’importation, changer le type des colonnes, etc. et l’aperçu se met à jour. De même, le code correspondant à l’importation du fichier avec les options sélectionnées est affiché dans la partie *Code Preview*.



Important : une fois que l’import semble correct, ne cliquez pas sur le bouton *Import*. À la place, sélectionnez le code généré et copiez-le (ou cliquez sur l’icône en forme de presse papier) et choisissez *Cancel*. Ensuite, collez le code dans votre script et exécutez-le (vous pouvez supprimer la ligne commençant par *View*).

Cette manière de faire permet “d’automatiser” l’importation des données, puisqu’à la prochaine ouverture du script vous aurez juste à exécuter le code en question, sans repasser par l’interface d’import.

7.2 Import depuis un fichier Excel

L’extension `readxl`, qui fait également partie du *tidyverse*, permet d’importer des données directement depuis un fichier au format `xls` ou `xlsx`.

Elle ne fait pas partie du “coeur” du *tidyverse*, il faut donc la charger explicitement avec :

```
library(readxl)
```

On peut alors utiliser la fonction `read_excel` en lui spécifiant le nom du fichier :

```
d <- read_excel("fichier.xls")
```

Il est possible de spécifier la feuille et la plage de cellules que l'on souhaite importer avec les arguments `sheet` et `range` :

```
d <- read_excel("fichier.xls", sheet = "Feuille2", range = "C1:F124")
```

Comme pour l'import de fichiers texte, une interface interactive d'import de fichiers Excel est disponible dans RStudio dans l'onglet *Environment*. Pour y accéder, cliquez sur *Import Dataset* puis *From Excel....*.

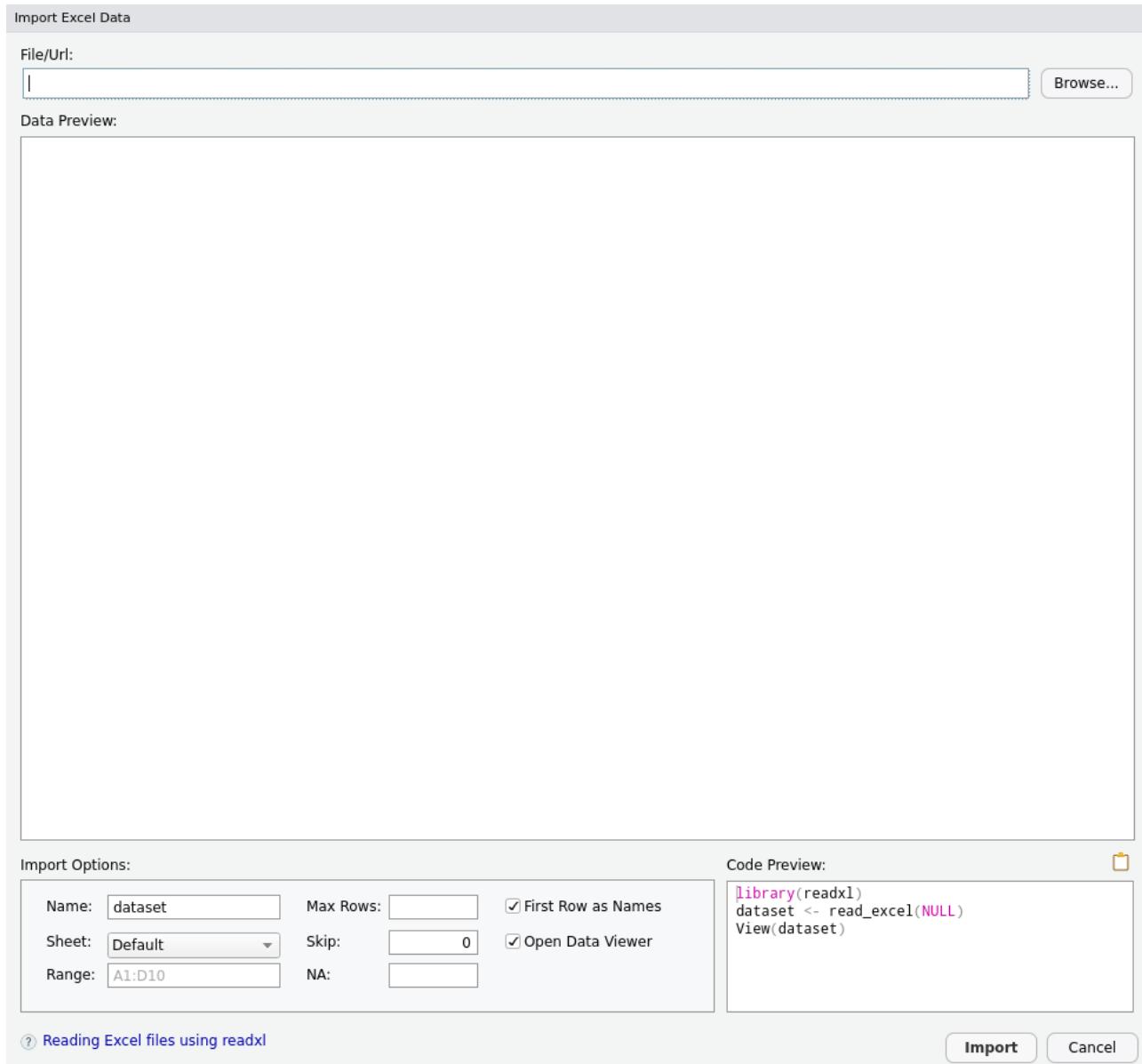


Figure 7.4: Dialogue d'importation d'un fichier Excel

Spécifiez le chemin ou l'URL du fichier dans le premier champ, vérifiez l'import dans la partie *Data Preview*, modifiez si besoin les options d'importation, copiez le code d'importation généré dans la partie *Code Preview* et collez le dans votre script.

Pour plus d'informations, voir [le site de l'extension `readxl`](#).

7.3 Import de fichiers SAS, SPSS et Stata

L'import de fichiers de données au format SAS, SPSS ou Stata se fait via les fonctions de l'extension `haven`.

Celle-ci fait partie du *tidyverse*, mais doit être chargée explicitement avec :

```
library(haven)
```

- Pour les fichiers provenant de SAS, vous pouvez utiliser les fonctions `read_sas` ou `read_xpt`
- Pour les fichiers provenant de SPSS, vous pouvez utiliser `read_sav` ou `read_por`
- Pour les fichiers provenant de Stata, utilisez `read_dta`

Chaque fonction dispose de plusieurs options. Le plus simple est d'utiliser, là aussi l'interface interactive d'importation de données de RStudio : dans l'onglet *Environment*, sélectionnez *Import Dataset* puis *From SPSS*, *From SAS* ou *From Stata*. Indiquez le chemin ou l'url du fichier, réglez les options d'importation, puis copiez le code d'importation généré et collez le dans votre script.

Pour plus d'informations, voir [le site de l'extension `haven`](#)

7.4 Import de fichiers dBase

Le format dBase est encore utilisé, notamment par l'INSEE, pour la diffusion de données volumineuses.

Les fichiers au format `dbf` peuvent être importées à l'aide de la fonction `read.dbf` de l'extension `foreign`¹ :

```
library(foreign)
d <- read.dbf("fichier.dbf")
```

La fonction `read.dbf` n'admet qu'un seul argument, `as.is`. Si `as.is = FALSE` (valeur par défaut), les chaînes de caractères sont automatiquement converties en `factor` à l'importation. Si `as.is = TRUE`, elles sont conservées telles quelles.

7.5 Connexion à des bases de données

7.5.1 Interfaçage via l'extension DBI

R est capable de s'interfacer avec différents systèmes de bases de données relationnelles, dont SQLite, MS SQL Server, PostgreSQL, MariaDB, etc.

Pour illustrer rapidement l'utilisation de bases de données, on va créer une base SQLite d'exemple à l'aide du code R suivant, qui copie la table du jeu de données `mtcars` dans une base de données `bdd.sqlite` :

¹`foreign` est une extension installée de base avec R, vous n'avez pas besoin de l'installer, il vous suffit de la charger avec `library`.

```
library(DBI)
library(RSQLite)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "resources/bdd.sqlite")
data(mtcars)
mtcars$name <- rownames(mtcars)
dbWriteTable(con, "mtcars", mtcars)
dbDisconnect(con)
```

Si on souhaite se connecter à cette base de données par la suite, on peut utiliser l'extension `DBI`, qui propose une interface générique entre R et différents systèmes de bases de données. On doit aussi avoir installé et chargé l'extension spécifique à notre base, ici `RSQLite`. On commence par ouvrir une connexion à l'aide de la fonction `dbConnect` de `DBI` :

```
library(DBI)
library(RSQLite)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "resources/bdd.sqlite")
```

La connexion est stockée dans un objet `con`, qu'on va utiliser à chaque fois qu'on voudra interroger la base.

On peut vérifier la liste des tables présentes et les champs de ces tables avec `dbListTables` et `dbListFields` :

```
dbListTables(con)
#> [1] "mtcars"
```

```
dbListFields(con, "mtcars")
#> [1] "mpg"    "cyl"   "disp"   "hp"    "drat"   "wt"    "qsec"   "vs"    "am"    "gear"
#> [11] "carb"  "name"
```

On peut également lire le contenu d'une table dans un objet de notre environnement avec `dbReadTable` :

```
cars <- dbReadTable(con, "mtcars")
```

On peut également envoyer une requête SQL directement à la base et récupérer le résultat :

```
dbGetQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
#>      mpg cyl  disp  hp drat   wt  qsec vs am gear carb          name
#> 1 22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1 Datsun 710
#> 2 24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2 Merc 240D
#> 3 22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2 Merc 230
#> 4 32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1 Fiat 128
#> 5 30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2 Honda Civic
#> 6 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1 Toyota Corolla
#> 7 21.5   4 120.1 97 3.70 2.465 20.01  1  0    3    1 Toyota Corona
#> 8 27.3   4  79.0 66 4.08 1.935 18.90  1  1    4    1 Fiat X1-9
#> 9 26.0   4 120.3 91 4.43 2.140 16.70  0  1    5    2 Porsche 914-2
#> 10 30.4  4  95.1 113 3.77 1.513 16.90  1  1    5    2 Lotus Europa
#> 11 21.4  4 121.0 109 4.11 2.780 18.60  1  1    4    2 Volvo 142E
```

Enfin, quand on a terminé, on peut se déconnecter à l'aide de `dbDisconnect` :

```
dbDisconnect(con)
```

Ceci n'est évidemment qu'un tout petit aperçu des fonctionnalités de DBI.

7.5.2 Utilisation de `dplyr` et `dbplyr`

L'extension `dplyr` est dédiée à la manipulation de données, elle est présentée chapitre 10. En installant l'extension complémentaire `dbplyr`, on peut utiliser `dplyr` directement sur une connection à une base de données générée par DBI :

```
library(DBI)
library(RSQLite)
library(dplyr)
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "resources/bdd.sqlite")
```

La fonction `tbl` notamment permet de créer un nouvel objet qui représente une table de la base de données :

```
cars_tbl <- tbl(con, "mtcars")
```



Ici l'objet `cars_tbl` n'est *pas* un tableau de données, c'est juste un objet permettant d'interroger la table de notre base de données.

On peut utiliser cet objet avec les verbes de `dplyr` :

```
cars_tbl %>%
  filter(cyl == 4) %>%
  select(name, mpg, cyl)
#> # Source:   lazy query [?? x 3]
#> # Database: sqlite 3.37.0
#> #   [/home/runner/work/tidyverse/tidyverse/resources/bdd.sqlite]
#>   name          mpg   cyl
#>   <chr>        <dbl> <dbl>
#> 1 Datsun 710    22.8     4
#> 2 Merc 240D    24.4     4
#> 3 Merc 230     22.8     4
#> 4 Fiat 128     32.4     4
#> 5 Honda Civic   30.4     4
#> 6 Toyota Corolla 33.9     4
#> 7 Toyota Corona 21.5     4
#> 8 Fiat X1-9     27.3     4
#> 9 Porsche 914-2   26.0     4
#> 10 Lotus Europa   30.4     4
#> # ... with more rows
```

`dbplyr` s'occupe, de manière transparente, de transformer les instructions `dplyr` en requête SQL, d'interroger la base de données et de renvoyer le résultat. De plus, tout est fait pour qu'un minimum d'opérations sur la base, parfois coûteuses en temps de calcul, ne soient effectuées.



Il est possible de modifier des objets de type `tbl`, par exemple avec `mutate` :

```
cars_tbl <- cars_tbl %>% mutate(type = "voiture")
```

Dans ce cas la nouvelle colonne `type` est bien créée et on peut y accéder par la suite. Mais **cette création se fait dans une table temporaire** : elle n'existe que le temps de la connexion à la base de données. À la prochaine connexion, cette nouvelle colonne n'apparaîtra pas dans la table.

Bien souvent on utilisera une base de données quand les données sont trop volumineuses pour être gérées par un ordinateur de bureau. Mais si les données ne sont pas trop importantes, il sera en général plus rapide de récupérer l'intégralité de la table dans notre session R pour pouvoir la manipuler comme les tableaux de données habituels. Ceci se fait grâce à la fonction `collect` de `dplyr` :

```
cars <- cars_tbl %>% collect
```

Ici, `cars` est bien un tableau de données classique, copie de la table de la base au moment du `collect`.

Et dans tous les cas, on n'oubliera pas de se déconnecter avec :

```
dbDisconnect(con)
```

7.5.3 Ressources

Pour plus d'informations, voir la [documentation très complète](#) (en anglais) proposée par RStudio.

Par ailleurs, depuis la version 1.1, RStudio facilite la connexion à certaines bases de données grâce à l'onglet *Connections*. Pour plus d'informations on pourra se référer à l'article (en anglais) [Using RStudio Connections](#).

7.6 Export de données

7.6.1 Export de tableaux de données

On peut avoir besoin d'exporter un tableau de données dans R vers un fichier dans différents formats. La plupart des fonctions d'import disposent d'un équivalent permettant l'export de données. On citera notamment :

- `write_csv`, `write_csv2`, `read_tsv` permettent d'enregistrer un *data frame* ou un tibble dans un fichier au format texte délimité
- `write_sas` permet d'exporter au format SAS
- `write_sav` permet d'exporter au format SPSS
- `write_dta` permet d'exporter au format Stata

Il n'existe par contre pas de fonctions permettant d'enregistrer directement au format `xls` ou `xlsx`. On peut dans ce cas passer par un fichier CSV.

Ces fonctions sont utiles si on souhaite diffuser des données à quelqu'un d'autre, ou entre deux logiciels.

Si vous travaillez sur des données de grandes dimensions, les formats texte peuvent être lents à exporter et importer. Dans ce cas, d'autres extensions comme `arrow` ou `fst` peuvent être utiles : elles permettent d'enregistrer des *data frames* dans des formats plus rapides. Les formats proposés par `arrow` permettent en outre l'échange de données tabulaires avec d'autres langages de programmation comme Python ou JavaScript.

7.6.2 Sauvegarder des objets

Une autre manière de sauvegarder des données est de les enregistrer au format **RData**. Ce format propre à R est compact, rapide, et permet d'enregistrer plusieurs objets R, quel que soit leur type, dans un même fichier.

Pour enregistrer des objets, il suffit d'utiliser la fonction **save** et de lui fournir la liste des objets à sauvegarder et le nom du fichier :

```
save(d, rp2018, tab, file = "fichier.RData")
```

Pour charger des objets préalablement enregistrés, utiliser **load** :

```
load("fichier.RData")
```

Les objets **d**, **rp2018** et **tab** devraient alors apparaître dans votre environnement.



Attention, quand on utilise **load**, les objets chargés sont importés directement dans l'environnement en cours avec leur nom d'origine. Si d'autres objets du même nom existent déjà, ils sont écrasés sans avertissement.

Une alternative est d'utiliser les fonctions **saveRDS** et **readRDS**, qui permettent d'enregistrer un unique objet, et de le charger dans notre session avec le nom que l'on souhaite.

```
saveRDS(rp2018, "fichier.rds")
df <- readRDS("fichier.rds")
```

Chapitre 8

Visualiser avec `ggplot2`

`ggplot2` est une extension du *tidyverse* qui permet de générer des graphiques avec une syntaxe cohérente et puissante. Elle nécessite l'apprentissage d'un "mini-langage" supplémentaire, mais permet la construction de graphiques complexes de manière efficace.

Une des particularités de `ggplot2` est qu'elle part du principe que les données relatives à un graphique sont stockées dans un tableau de données (*data frame*, *tibble* ou autre).

8.1 Préparation

`ggplot2` fait partie du cœur du *tidyverse*, elle est donc chargée automatiquement avec :

```
library(tidyverse)
```

On peut également la charger explicitement avec :

```
library(ggplot2)
```

Dans ce qui suit on utilisera le jeu de données issu du recensement de la population de 2018 inclus dans l'extension `questionr` (résultats partiels concernant les communes de plus de 2000 habitants de France métropolitaine). On charge ces données et on en extrait les données de 5 départements (l'utilisation de la fonction `filter` sera expliquée dans la section 10.2.2 de la [partie sur `dplyr`](#) :

```
library(questionr)
data(rp2018)

rp <- filter(
  rp2018,
  département %in% c("Oise", "Rhône", "Hauts-de-Seine", "Lozère", "Bouches-du-Rhône")
)
```

8.2 Initialisation

Un graphique `ggplot2` s'initialise à l'aide de la fonction `ggplot()`. Les données représentées graphiquement sont toujours issues d'un tableau de données (*data frame* ou *tibble*), qu'on passe en argument `data` à la fonction :

```
ggplot(data = rp)
## Ou, équivalent
ggplot(rp)
```

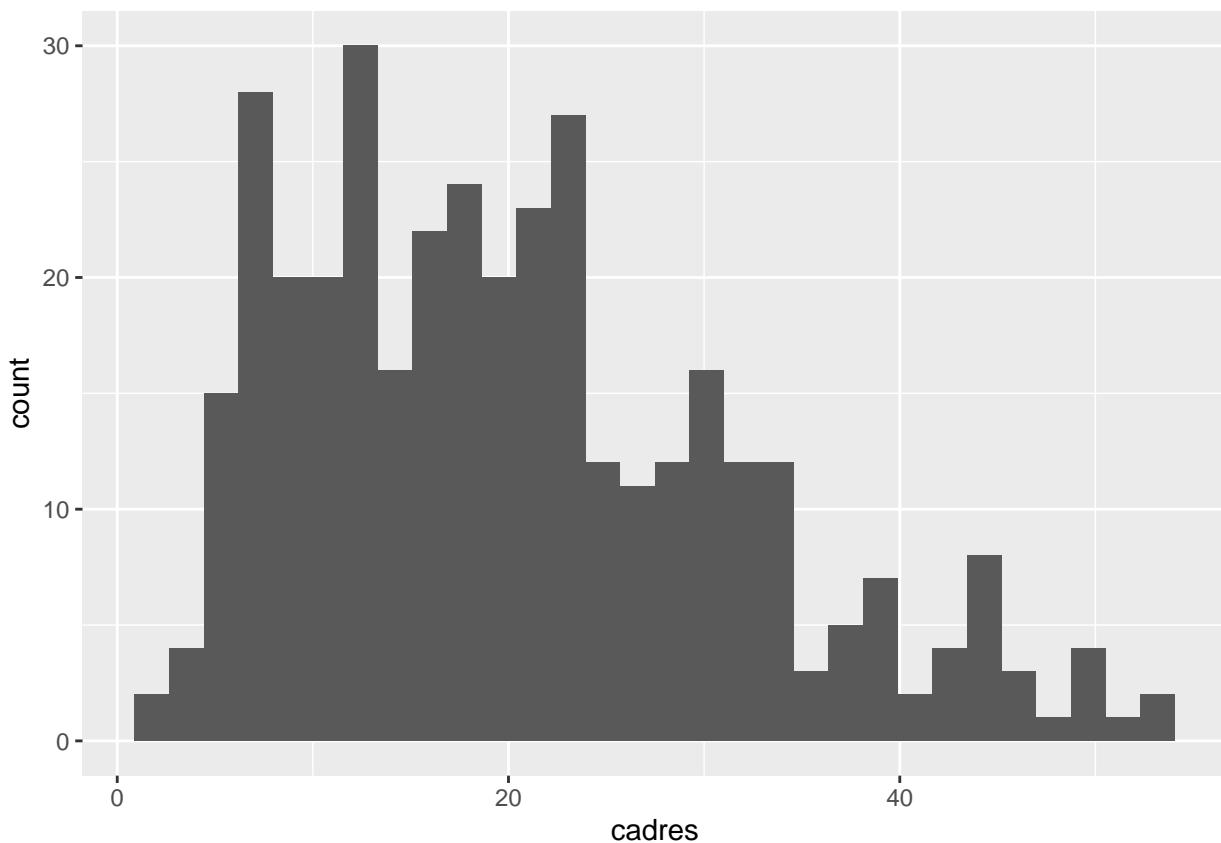
On a défini la source de données, il faut maintenant ajouter des éléments de représentation graphique. Ces éléments sont appelés des `geom`, et on les ajoute à l'objet graphique de base avec l'opérateur `+`.

Un des `geom` les plus simples est `geom_histogram`. On peut l'ajouter de la manière suivante :

```
ggplot(rp) + geom_histogram()
```

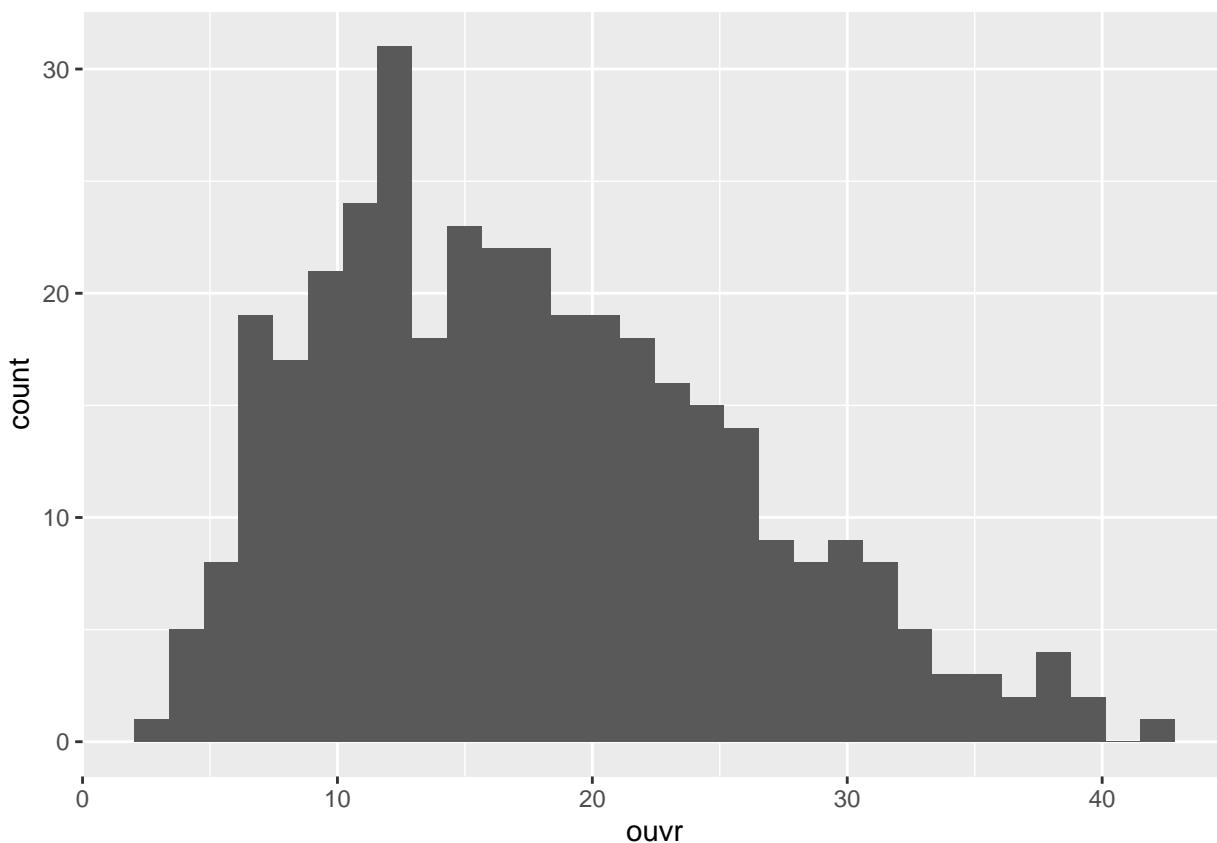
Reste à indiquer quelle donnée nous voulons représenter sous forme d'histogramme. Cela se fait à l'aide d'arguments passés via la fonction `aes()`. Ici nous avons un paramètre à renseigner, `x`, qui indique la variable à représenter sur l'axe des `x` (l'axe horizontal). Ainsi, si on souhaite représenter la distribution des communes du jeu de données selon le pourcentage de cadres dans leur population active (variable `cadres`), on pourra faire :

```
ggplot(rp) + geom_histogram(aes(x = cadres))
```



Si on veut représenter une autre variable, il suffit de changer la valeur de `x` :

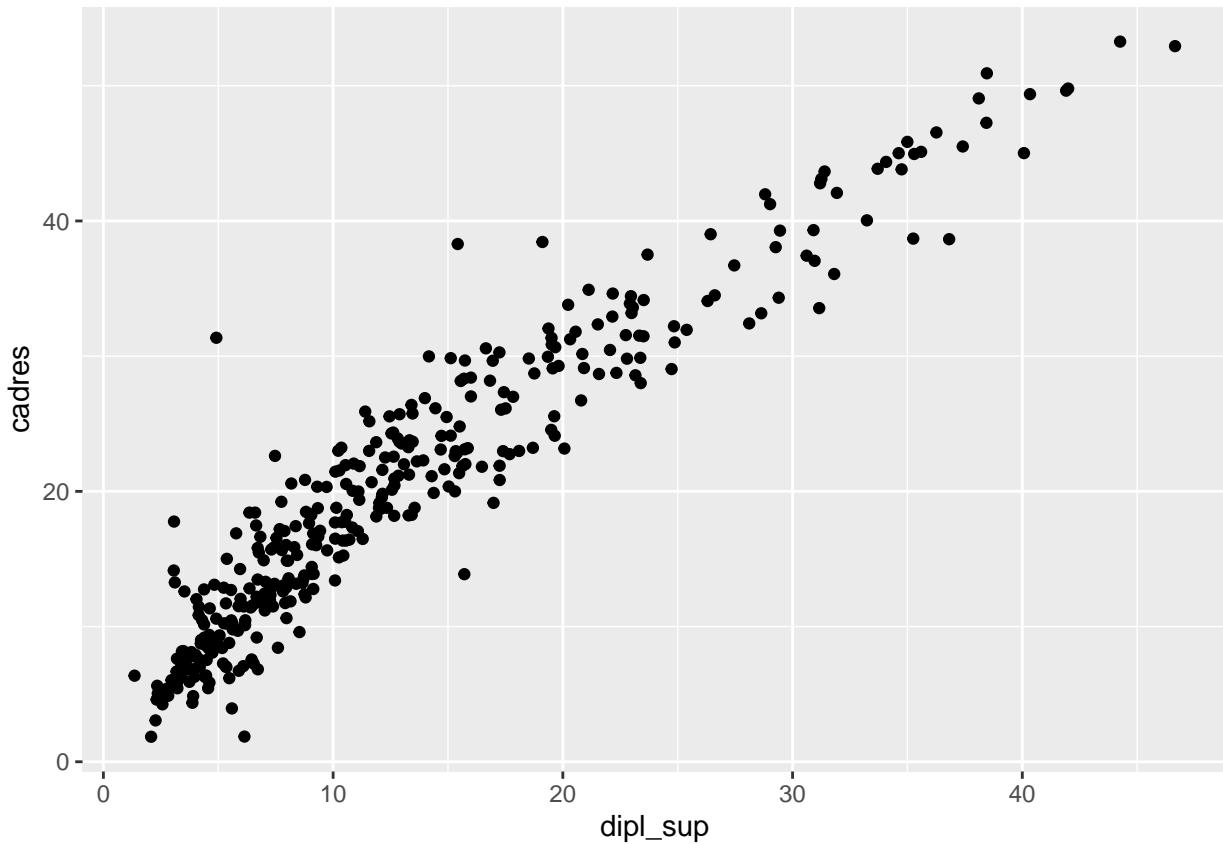
```
ggplot(rp) + geom_histogram(aes(x = ouvr))
```



Quand on spécifie une variable, inutile d'indiquer le nom du tableau de données sous la forme `rp$ouvr`, car `ggplot2` recherche automatiquement la variable dans le tableau de données indiqué avec le paramètre `data`. On peut donc se contenter de `ouvr`.

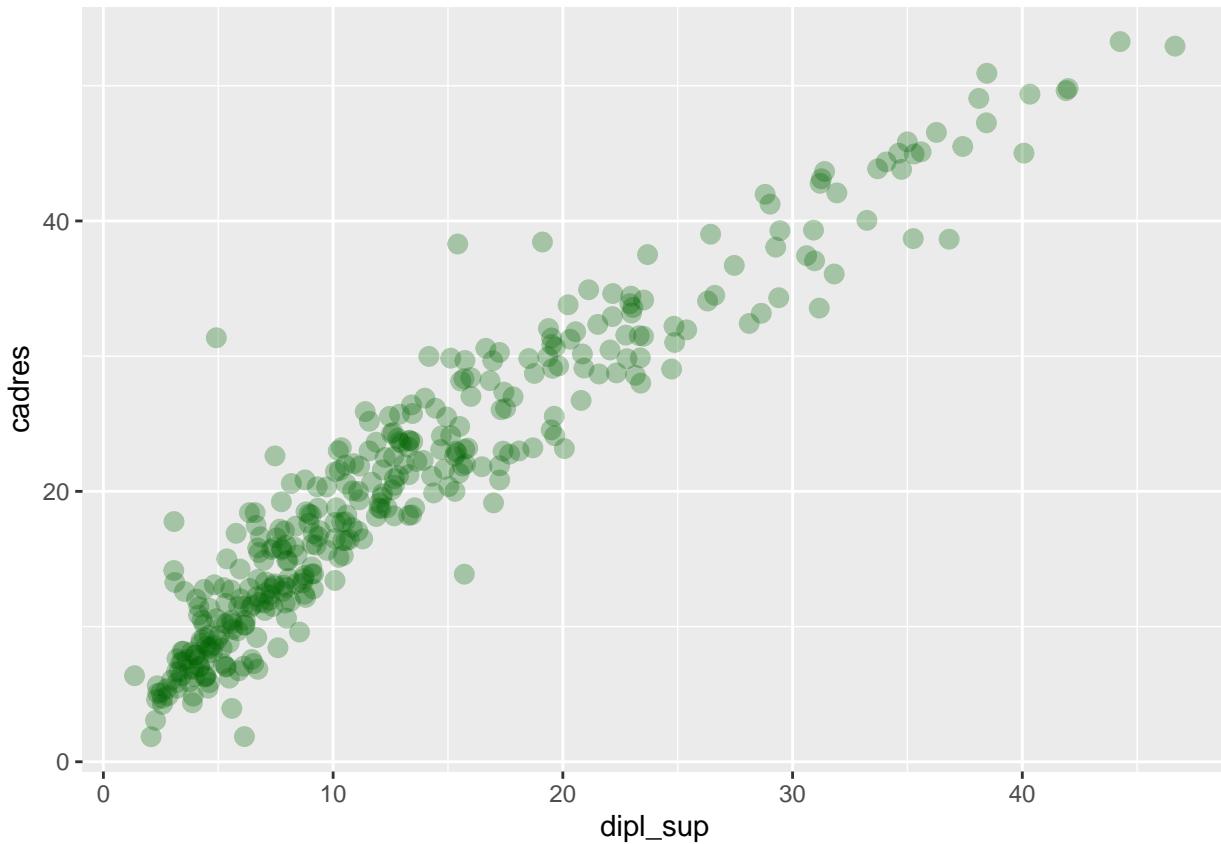
Certains `geom` prennent plusieurs paramètres. Ainsi, si on veut représenter un nuage de points, on peut le faire en ajoutant un `geom_point`. On doit alors indiquer à la fois la position en `x` (la variable sur l'axe horizontal) et en `y` (la variable sur l'axe vertical) de ces points, il faut donc passer ces deux arguments à `aes()` :

```
ggplot(rp) + geom_point(aes(x = dipl_sup, y = cadres))
```



On peut modifier certains attributs graphiques d'un `geom` en lui passant des arguments supplémentaires. Par exemple, pour un nuage de points, on peut modifier la couleur des points avec l'argument `color`, leur taille avec l'argument `size`, et leur transparence avec l'argument `alpha` :

```
ggplot(rp) +
  geom_point(
    aes(x = dipl_sup, y = cadres),
    color = "darkgreen", size = 3, alpha = 0.3
  )
```



On notera que dans ce cas les arguments sont dans la fonction `geom` mais à l'extérieur du `aes()`. Plus d'explications sur ce point dans quelques instants.

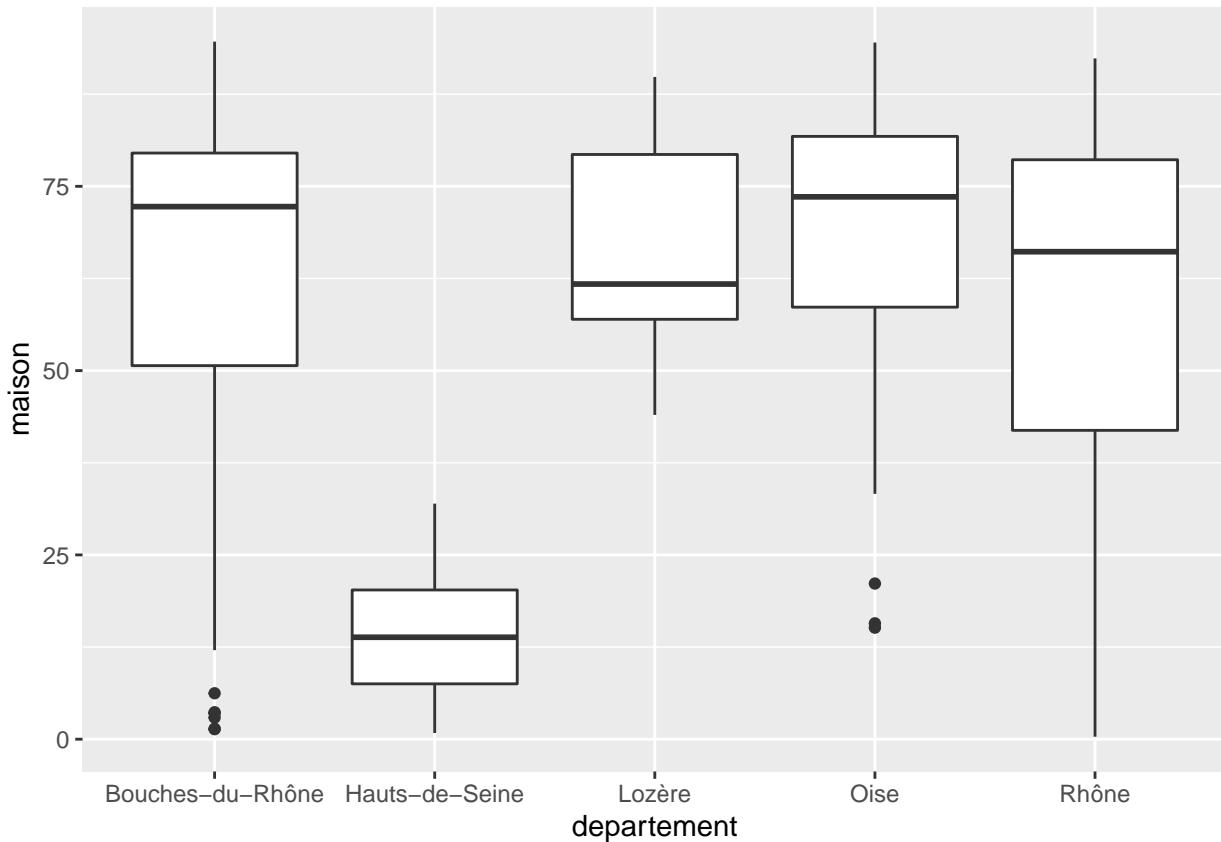
8.3 Exemples de `geom`

Il existe un grand nombre de `geom`, décrits en détail dans la [documentation officielle](#). Outre les `geom_histogram` et `geom_point` que l'on vient de voir, on pourra noter les `geom` suivants.

8.3.1 `geom_boxplot`

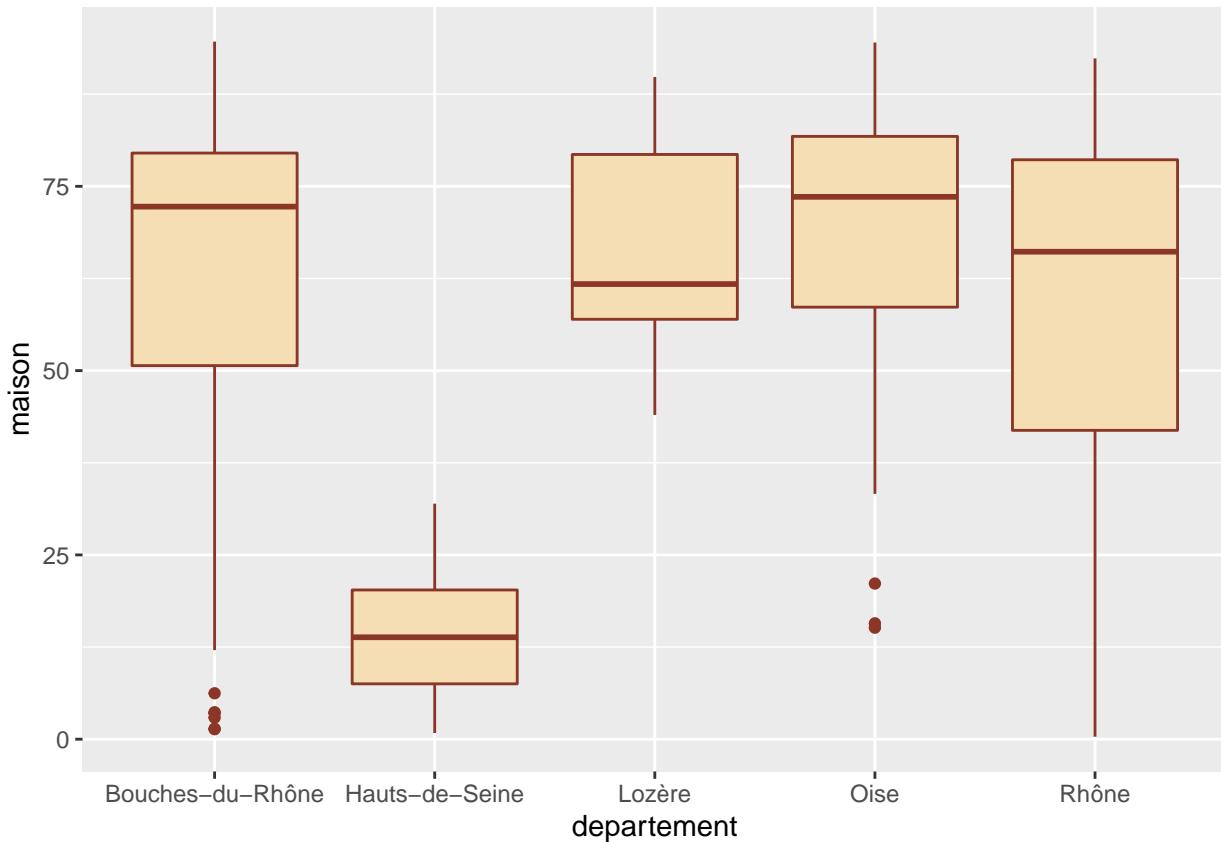
`geom_boxplot` permet de représenter des boîtes à moustaches. On lui passe en `y` la variable numérique dont on veut étudier la répartition, et en `x` la variable qualitative contenant les classes qu'on souhaite comparer. Ainsi, si on veut comparer la répartition du pourcentage de maisons en fonction du département de la commune, on pourra faire :

```
ggplot(rp) + geom_boxplot(aes(x = departement, y = maison))
```



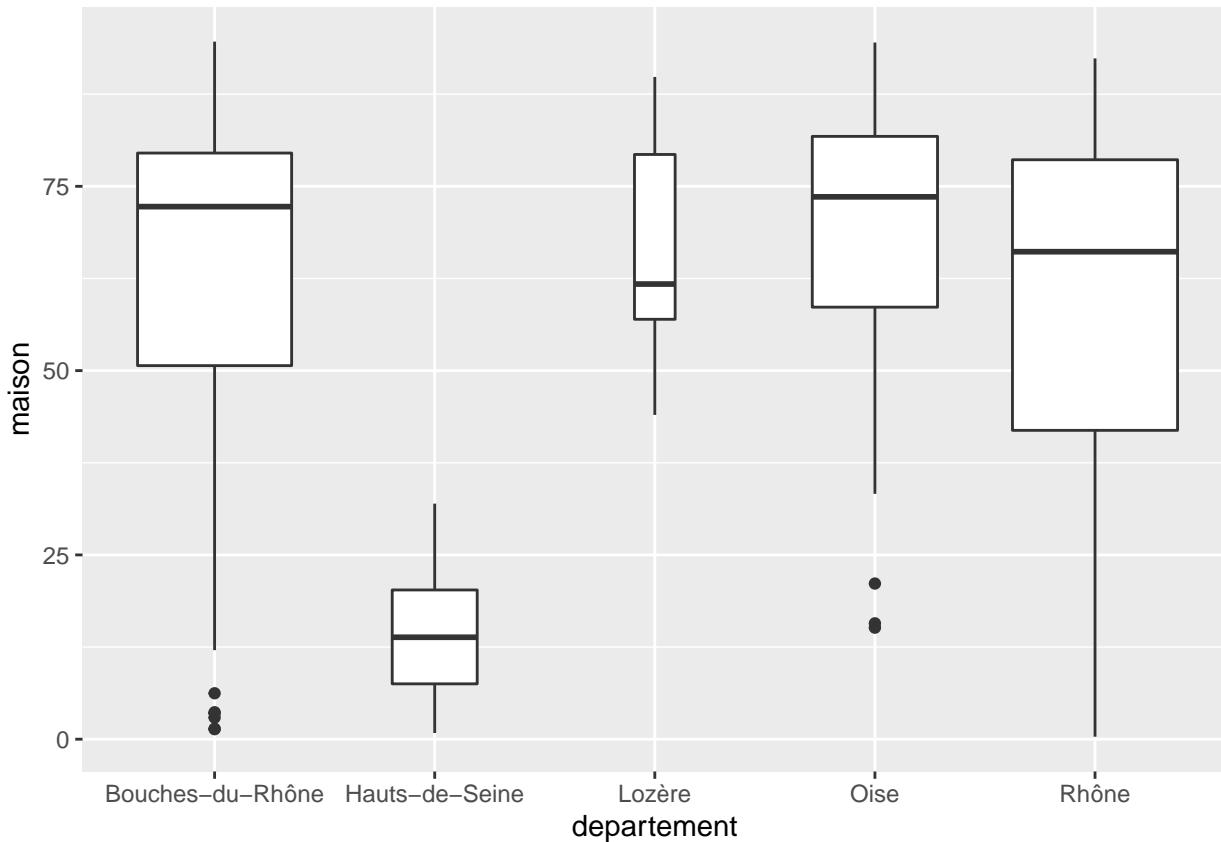
On peut personnaliser la présentation avec différents arguments supplémentaires comme `fill` ou `color` :

```
ggplot(rp) +
  geom_boxplot(
    aes(x = departement, y = maison),
    fill = "wheat", color = "tomato4"
  )
```



Un autre argument utile, `varwidth`, permet de faire varier la largeur des boîtes en fonction des effectifs de la classe (donc, ici, en fonction du nombre de communes de chaque département) :

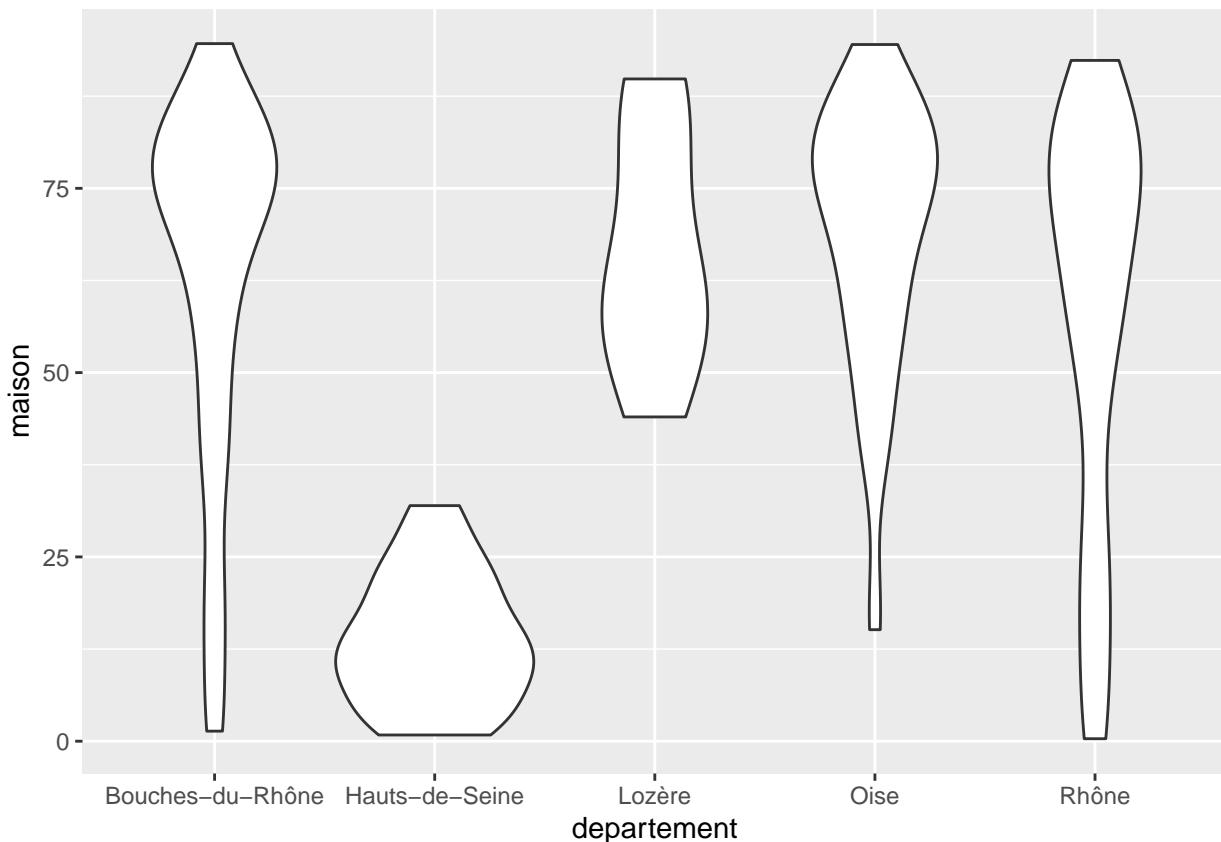
```
ggplot(rp) +
  geom_boxplot(aes(x = departement, y = maison), varwidth = TRUE)
```



8.3.2 geom_violin

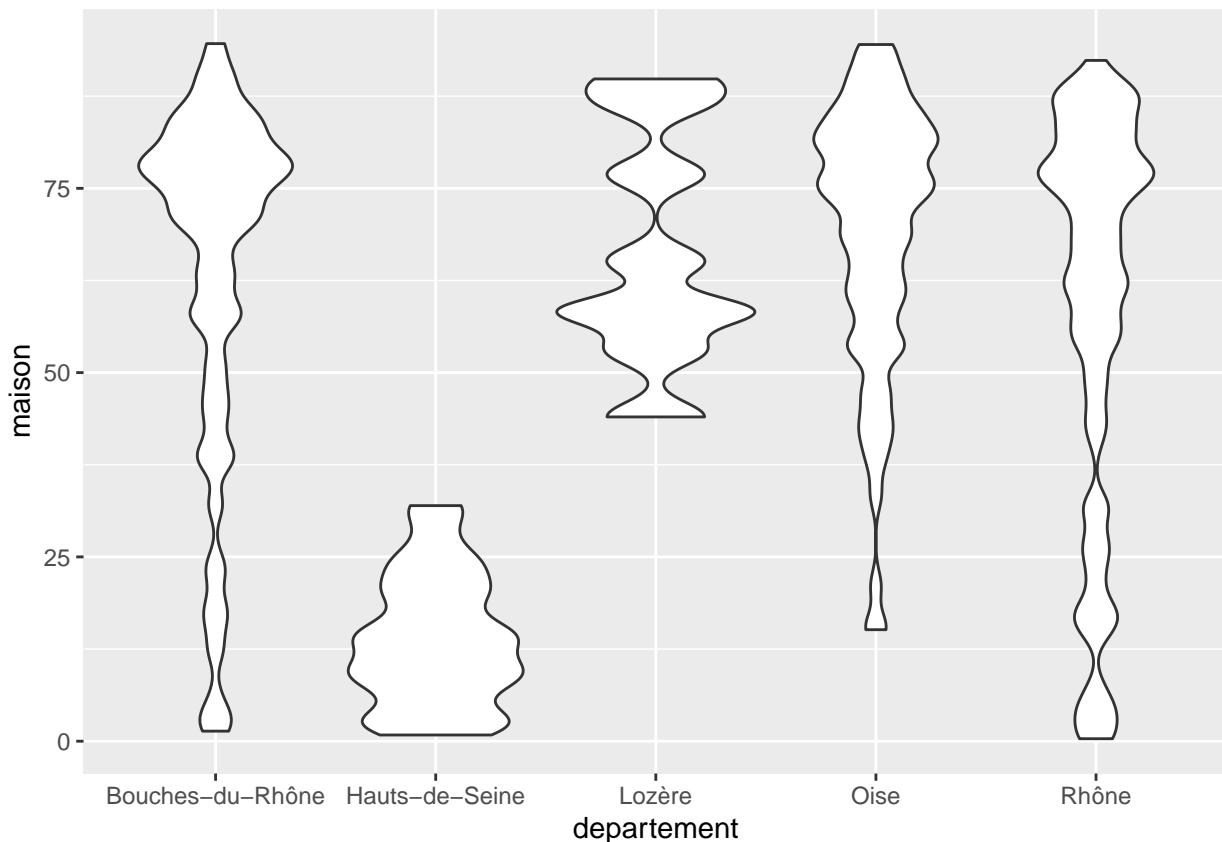
`geom_violin` est très semblable à `geom_boxplot`, mais utilise des graphes en violon à la place des boîtes à moustache.

```
ggplot(rp) + geom_violin(aes(x = departement, y = maison))
```



Les graphes en violon peuvent donner une lecture plus fine des différences de distribution selon les classes. Comme pour les graphiques de densité, on peut faire varier le niveau de “détail” de la représentation en utilisant l’argument `bw` (bande passante).

```
ggplot(rp) +
  geom_violin(
    aes(x = département, y = maison),
    bw = 2
  )
```

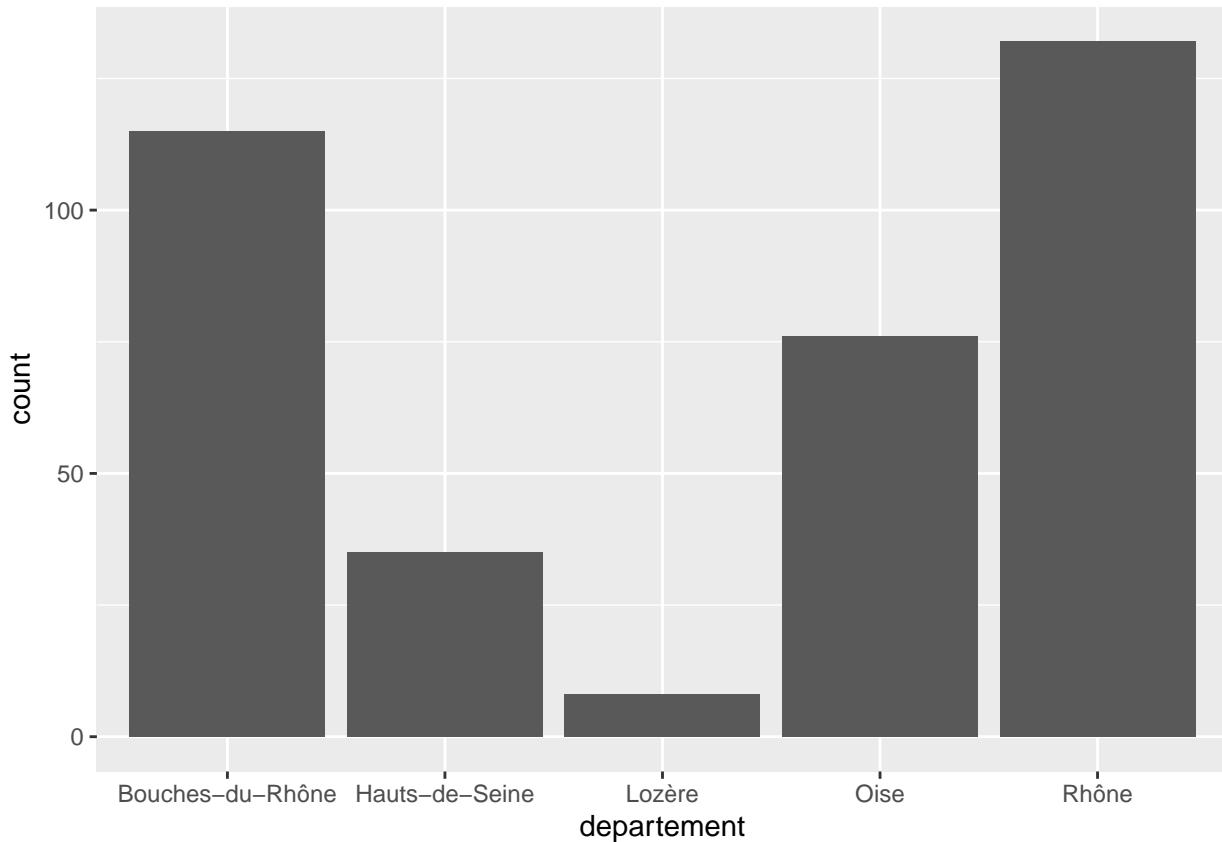


8.3.3 `geom_bar` et `geom_col`

`geom_bar` permet de produire un graphique en bâtons (*barplot*). On lui passe en `x` la variable qualitative dont on souhaite représenter l'effectif de chaque modalité.

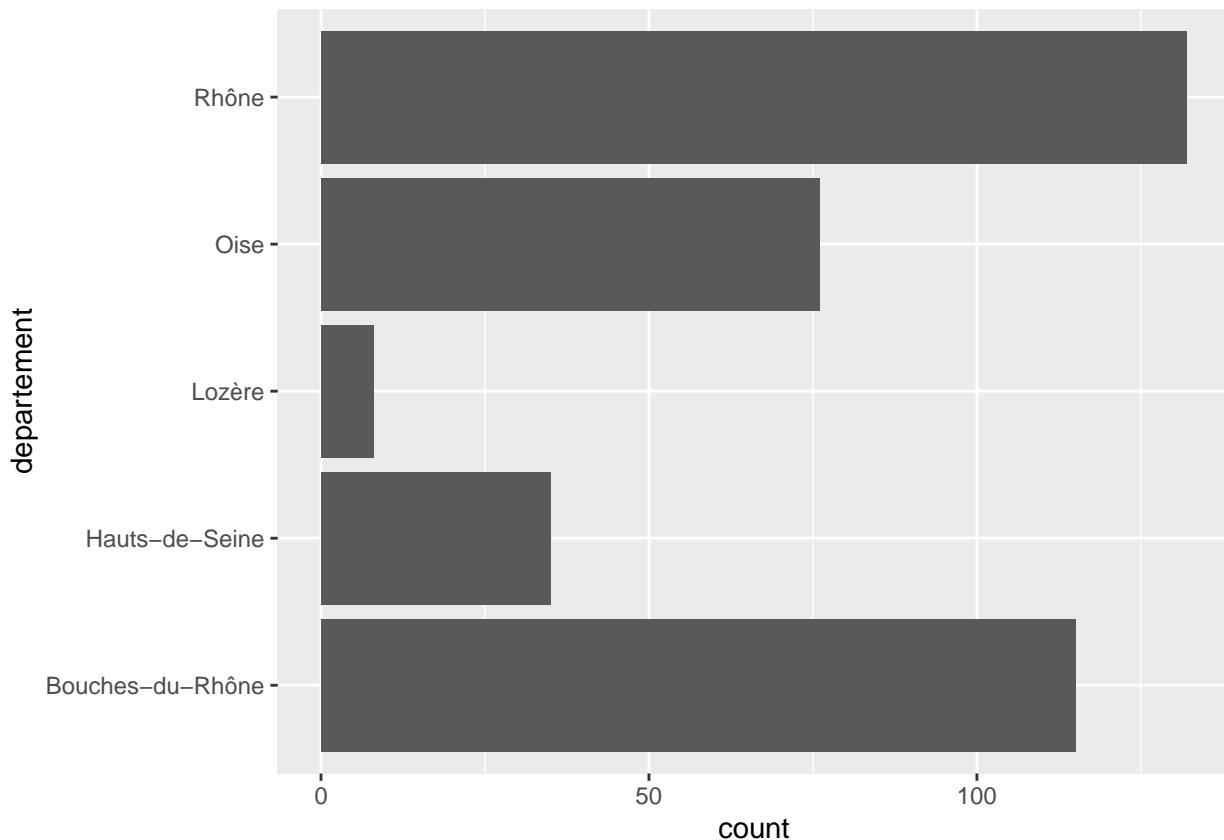
Par exemple, si on veut afficher le nombre de communes de notre jeu de données pour chaque département :

```
ggplot(rp) + geom_bar(aes(x = département))
```



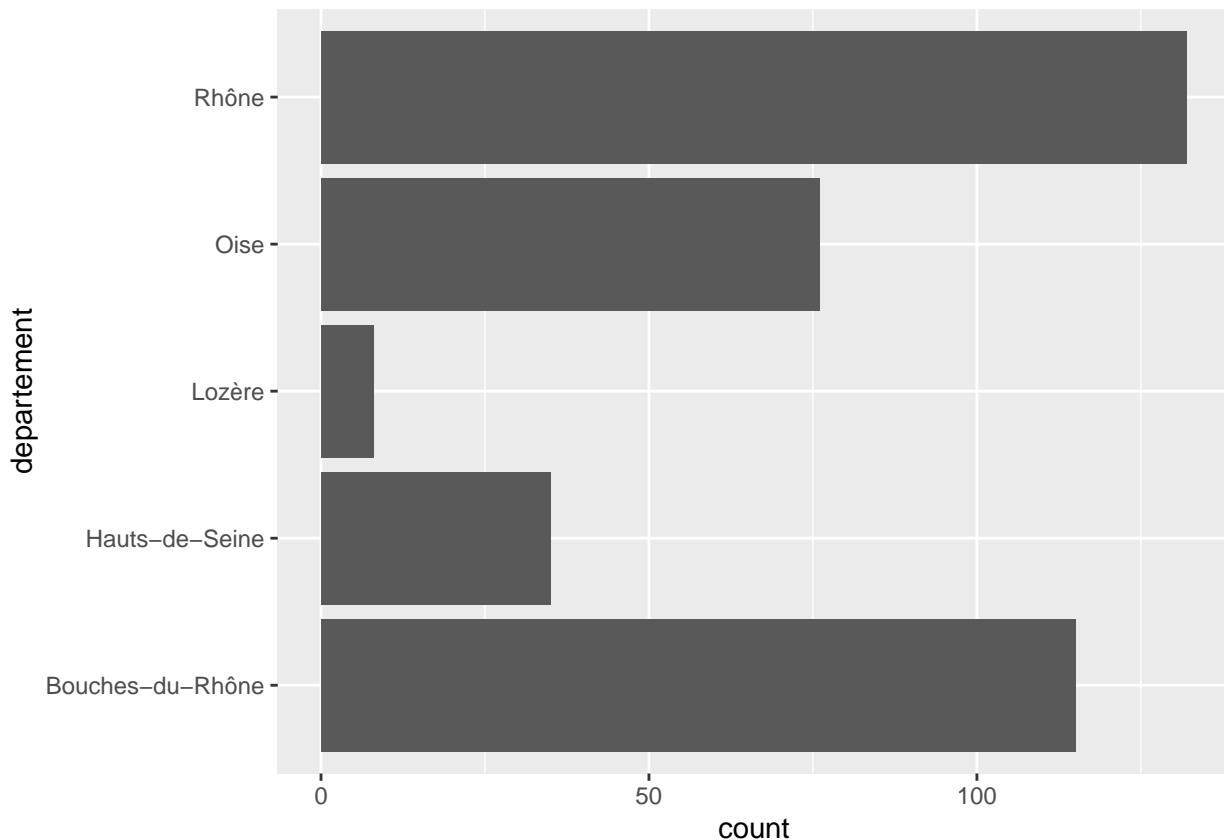
Si on préfère avoir un graphique en barres horizontales, il suffit de passer la variable comme attribut *y* plutôt que *x*.

```
ggplot(rp) + geom_bar(aes(y = departement))
```



Une autre possibilité est d'utiliser `coord_flip()`, qui permet d'intervertir l'axe horizontal et l'axe vertical.

```
ggplot(rp) +
  geom_bar(aes(x = département)) +
  coord_flip()
```



À noter que `coord_flip()` peut s'appliquer à n'importe quel graphique `ggplot2`.

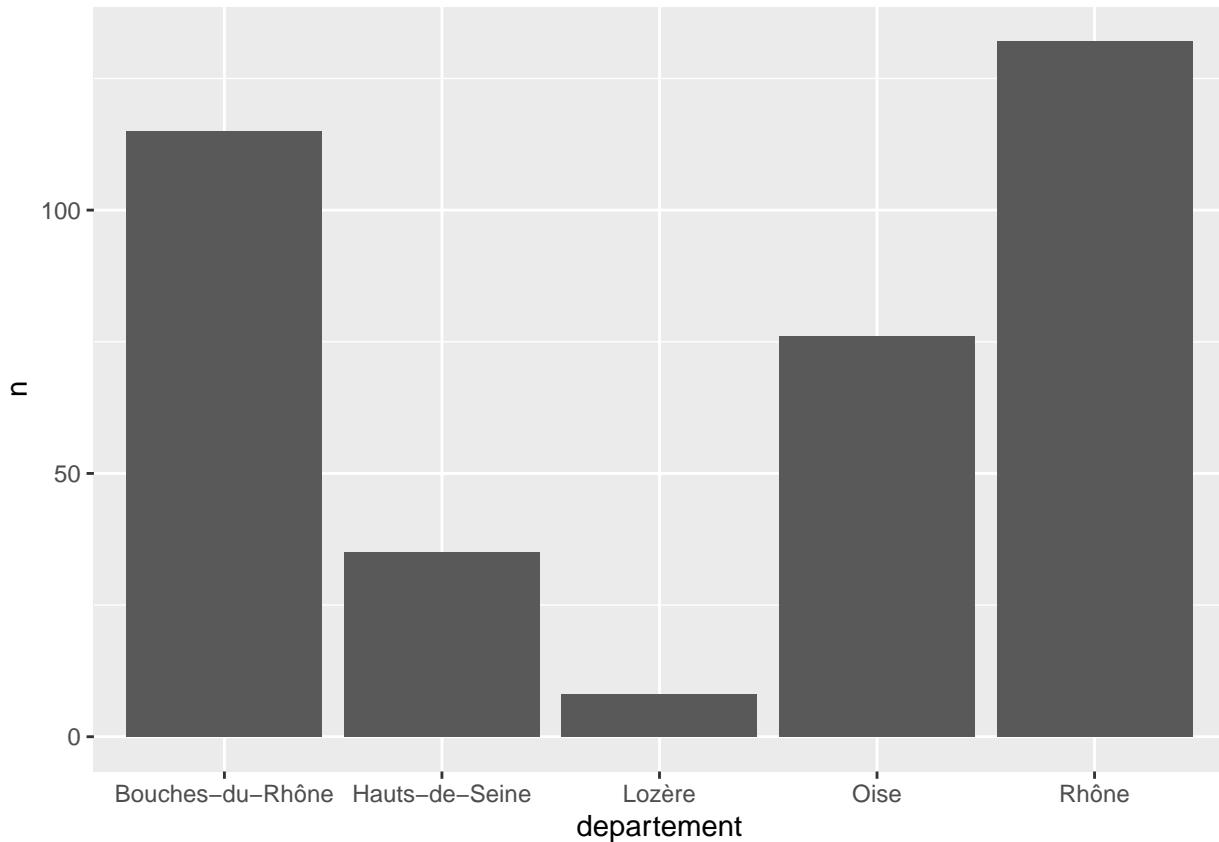
Parfois, on a déjà calculé le tri à plat de la variable à représenter. Dans ce cas on souhaite représenter les effectifs directement, sans les calculer.

C'est le cas par exemple si on a les données sous la forme suivante dans un tableau de données nommé `tab` :

```
tab
#>      departement   n
#> 1 Bouches-du-Rhône 115
#> 2 Hauts-de-Seine  35
#> 3     Lozère     8
#> 4       Oise    76
#> 5     Rhône   132
```

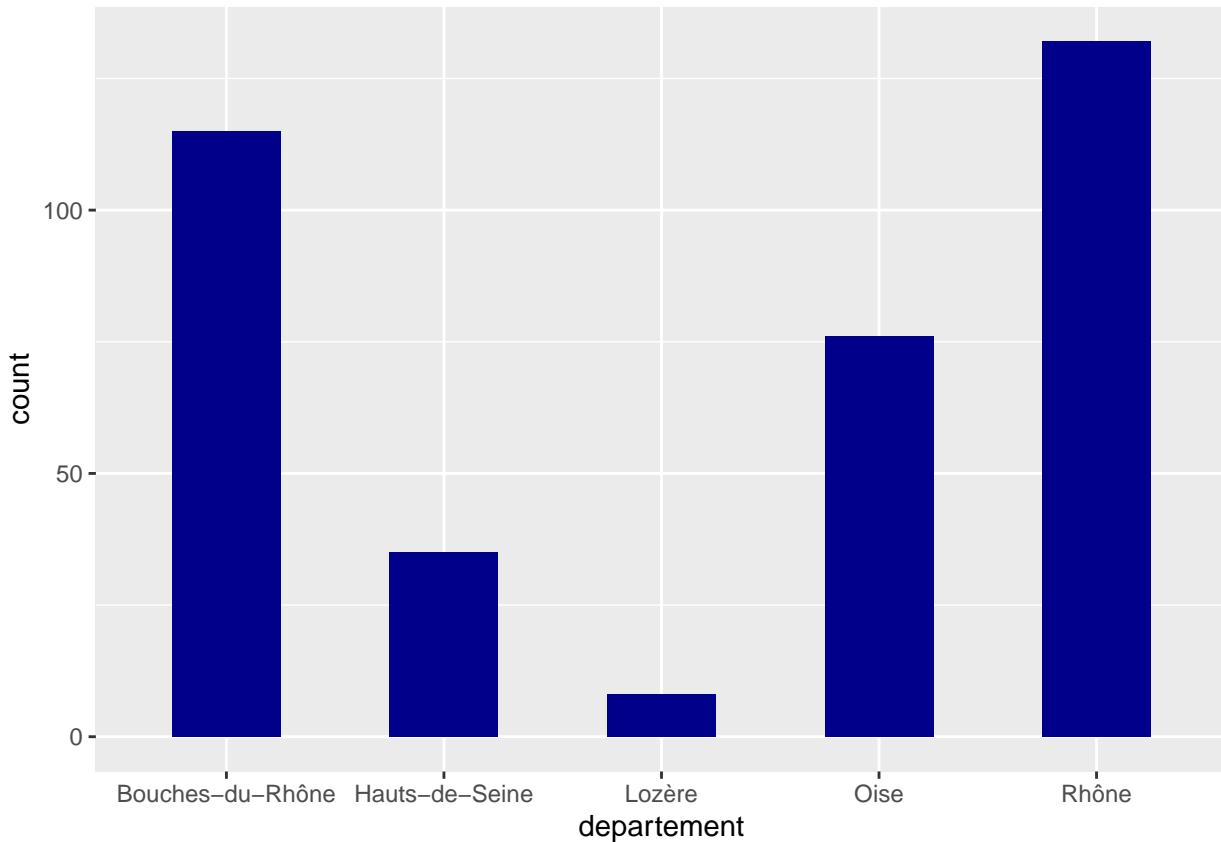
Dans ce cas on peut utiliser `geom_col`. Il faut alors spécifier, en plus de l'argument `x`, un argument `y` indiquant la variable contenant la hauteur des barres. Dans notre exemple, il s'agit de la variable `n`.

```
ggplot(tab) + geom_col(aes(x = departement, y = n))
```



Que ce soit pour `geom_bar` ou `geom_col`, on peut modifier l'apparence du graphique en passant des arguments supplémentaires comme `fill` ou `width`.

```
ggplot(rp) +
  geom_bar(
    aes(x = departement),
    fill = "darkblue", width = .5
  )
```

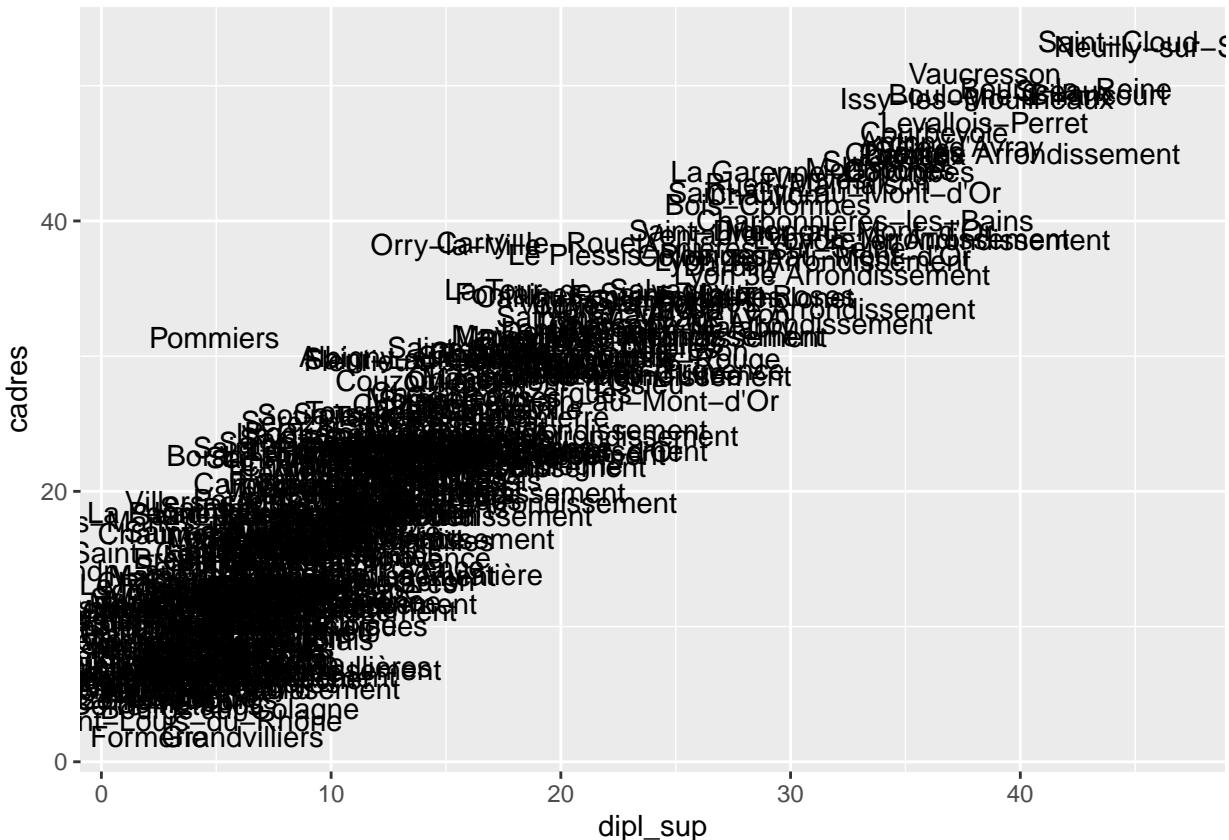


8.3.4 geom_text

`geom_text` permet d'afficher des étiquettes de texte. On doit lui fournir trois paramètres dans `aes` : `x` et `y` pour la position des étiquettes, et `label` pour leur texte.

Par exemple, si on souhaite représenter le nuage croisant la part des diplômés du supérieur et la part de cadres, mais en affichant le nom de la commune (variable `commune`) plutôt qu'un simple point, on peut faire :

```
ggplot(rp) +
  geom_text(
    aes(x = dipl_sup, y = cadres, label = commune)
  )
#> Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <c5>
#> Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <93>
```

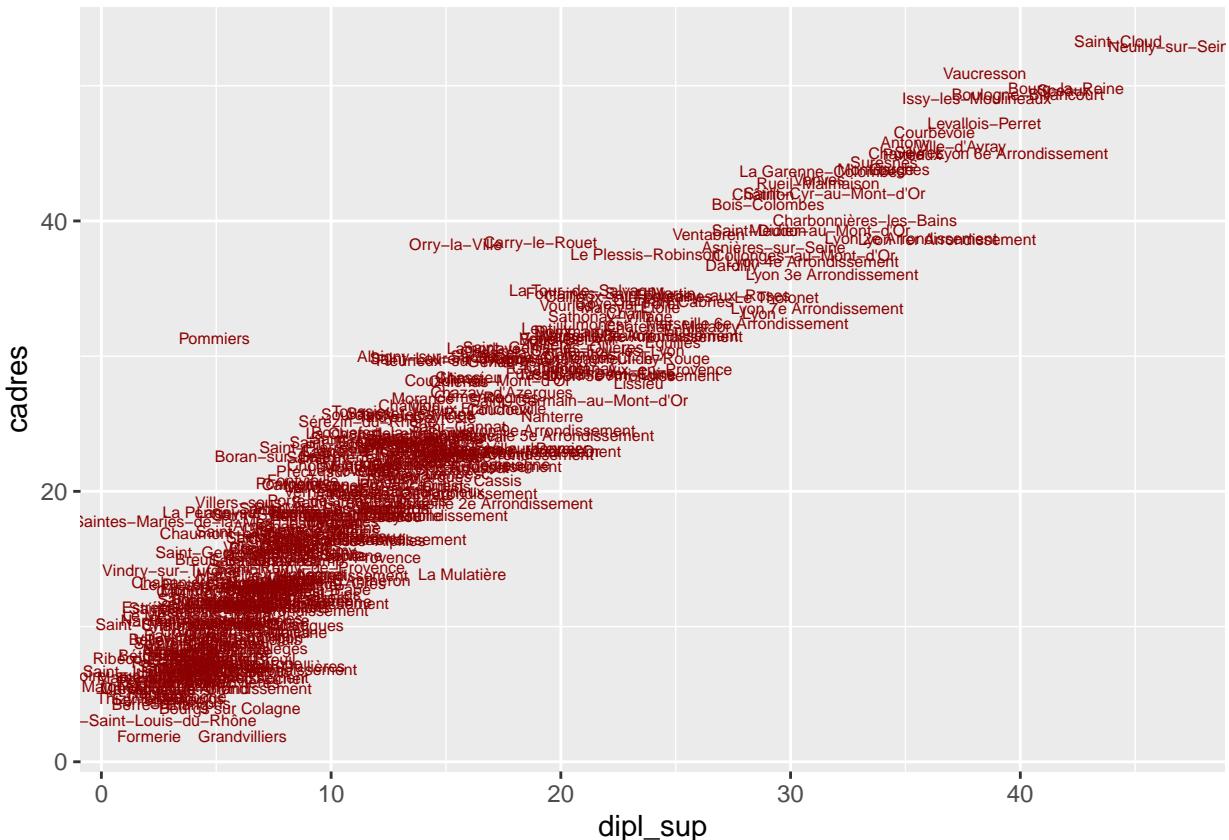


On peut personnaliser l'apparence et la position du texte avec des arguments comme `size`, `color`, etc.

```

ggplot(rp) +
  geom_text(
    aes(x = dipl_sup, y = cadres, label = commune),
    color = "darkred", size = 2
  )
#> Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèveœil-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <c5>
#> Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèveœil-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <93>

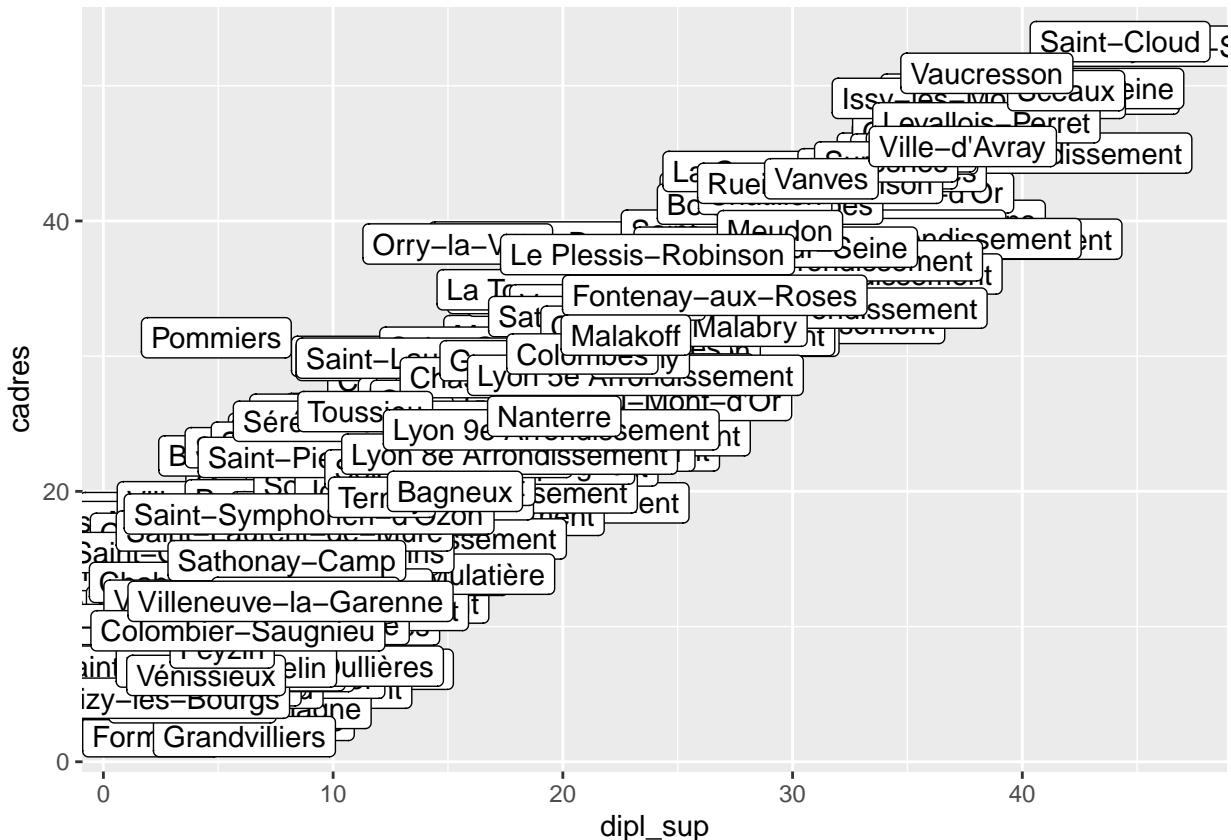
```



8.3.5 geom_label

`geom_label` est identique à `geom_text`, mais avec une présentation un peu différente.

```
ggplot(rp) + geom_label(aes(x = dipl_sup, y = cadres, label = commune))
#> Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, : erreur
#> de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est substitué
#> pour <c5>
#> Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, : erreur
#> de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est substitué
#> pour <93>
#> Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, : erreur
#> de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est substitué
#> pour <c5>
#> Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, : erreur
#> de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est substitué
#> pour <93>
#> Warning in grid.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <c5>
#> Warning in grid.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
#> erreur de conversion de 'Crèvecœur-le-Grand' dans 'mbcsToSbcs' : le point est
#> substitué pour <93>
```

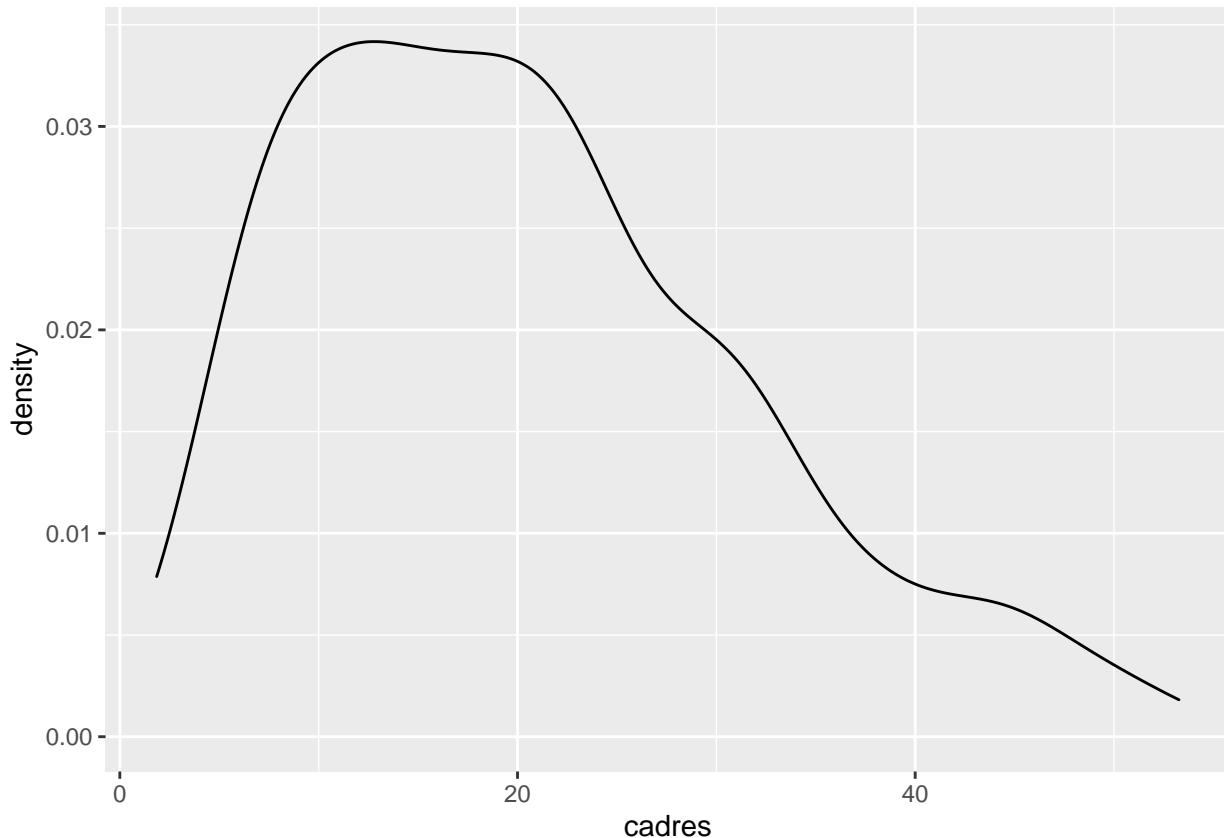


8.3.6 geom_density

`geom_density` permet d'afficher l'estimation de densité d'une variable numérique. Son usage est similaire à celui de `geom_histogram`.

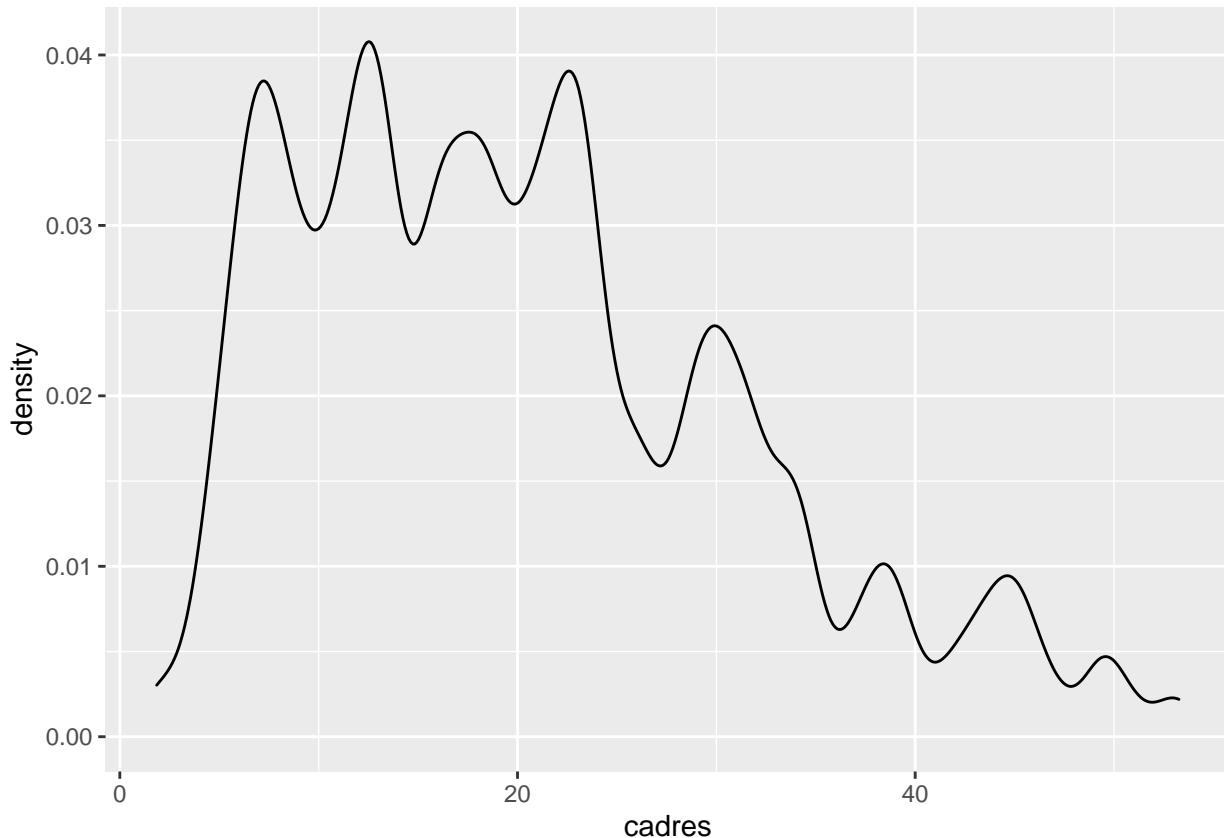
Ainsi, si on veut afficher la densité de la répartition de la part des cadres dans les communes de notre jeu de données :

```
ggplot(rp) + geom_density(aes(x = cadres))
```



On peut utiliser différents arguments pour ajuster le calcul de l'estimation de densité, parmi lesquels `kernel` et `bw` (voir la page d'aide de la fonction `density` pour plus de détails). `bw` (abréviation de *bandwidth*, bande passante) permet de régler la “finesse” de l'estimation de densité, un peu comme le choix du nombre de classes dans un histogramme :

```
ggplot(rp) + geom_density(aes(x = cadres), bw = 1)
```



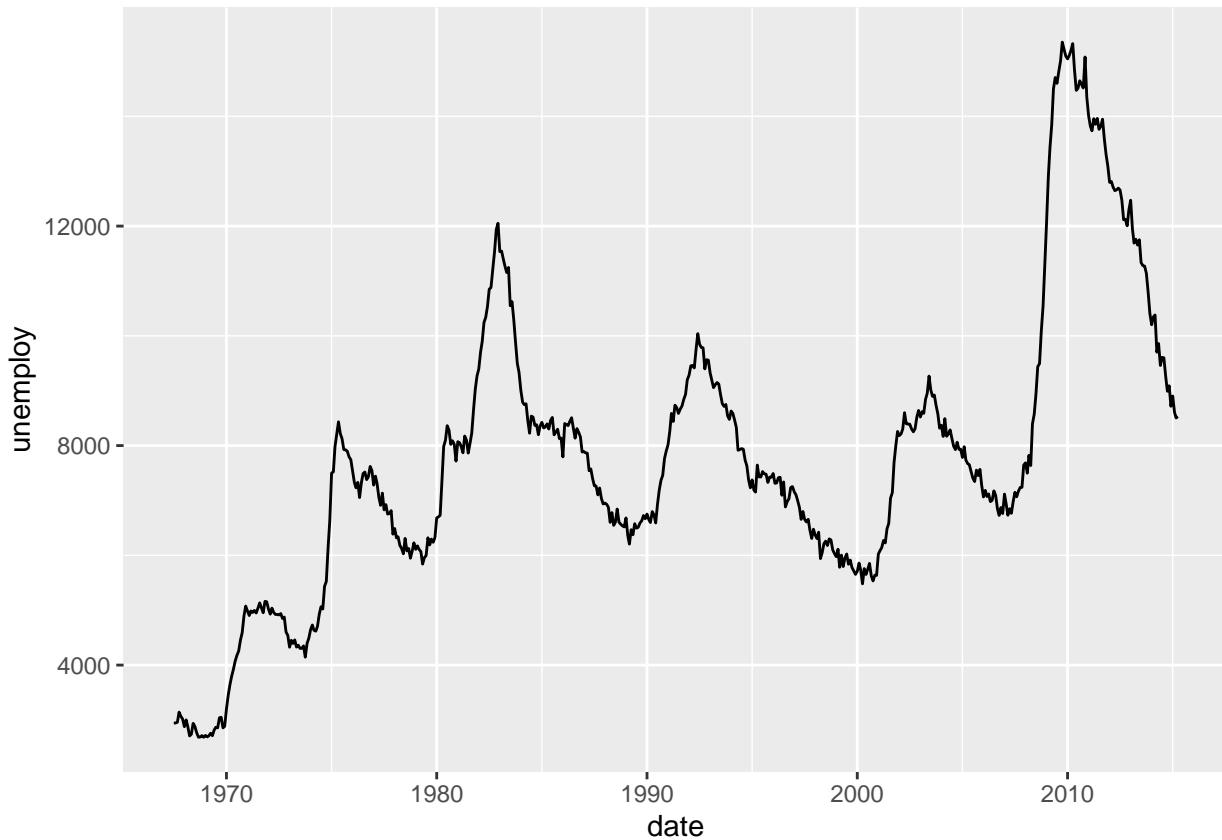
8.3.7 `geom_line`

`geom_line` trace des lignes connectant les différentes observations entre elles. Il est notamment utilisé pour la représentation de séries temporelles. On passe à `geom_line` deux paramètres : `x` et `y`. Les observations sont alors connectées selon l'ordre des valeurs passées en `x`.

Comme il n'y a pas de données adaptées pour ce type de représentation dans notre jeu de données d'exemple, on va utiliser ici le jeu de données `economics` inclus dans `ggplot2` et représenter l'évolution du taux de chômage aux États-Unis (variable `unemploy`) dans le temps (variable `date`) :

```
data("economics")
economics
#> # A tibble: 574 x 6
#>   date      pce    pop psavert uempmed unemploy
#>   <date>    <dbl>  <dbl>    <dbl>    <dbl>    <dbl>
#> 1 1967-07-01 507. 198712    12.6     4.5    2944
#> 2 1967-08-01 510. 198911    12.6     4.7    2945
#> 3 1967-09-01 516. 199113    11.9     4.6    2958
#> 4 1967-10-01 512. 199311    12.9     4.9    3143
#> 5 1967-11-01 517. 199498    12.8     4.7    3066
#> 6 1967-12-01 525. 199657    11.8     4.8    3018
#> 7 1968-01-01 531. 199808    11.7     5.1    2878
#> 8 1968-02-01 534. 199920    12.3     4.5    3001
#> 9 1968-03-01 544. 200056    11.7     4.1    2877
#> 10 1968-04-01 544. 200208    12.3     4.6    2709
#> # ... with 564 more rows
```

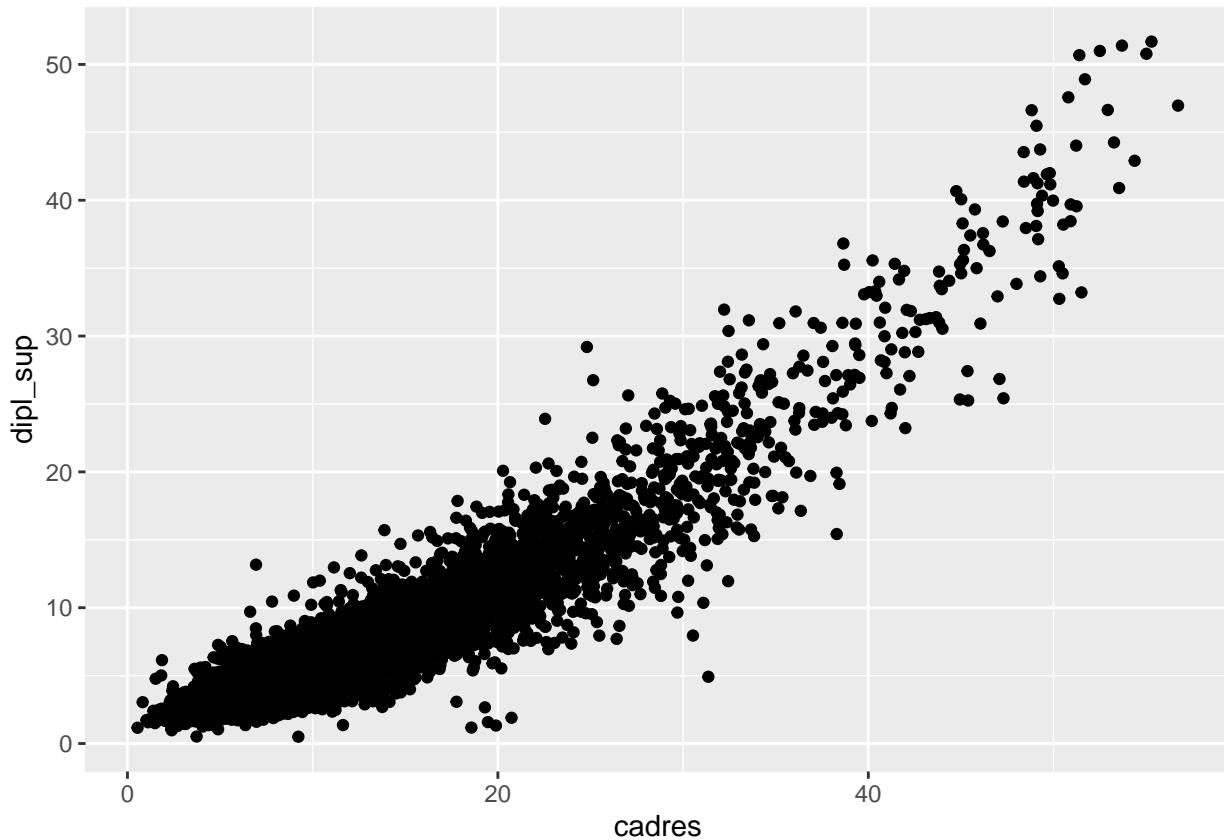
```
ggplot(economics) + geom_line(aes(x = date, y = unemploy))
```



8.3.8 `geom_hex` et `geom_bin2d`

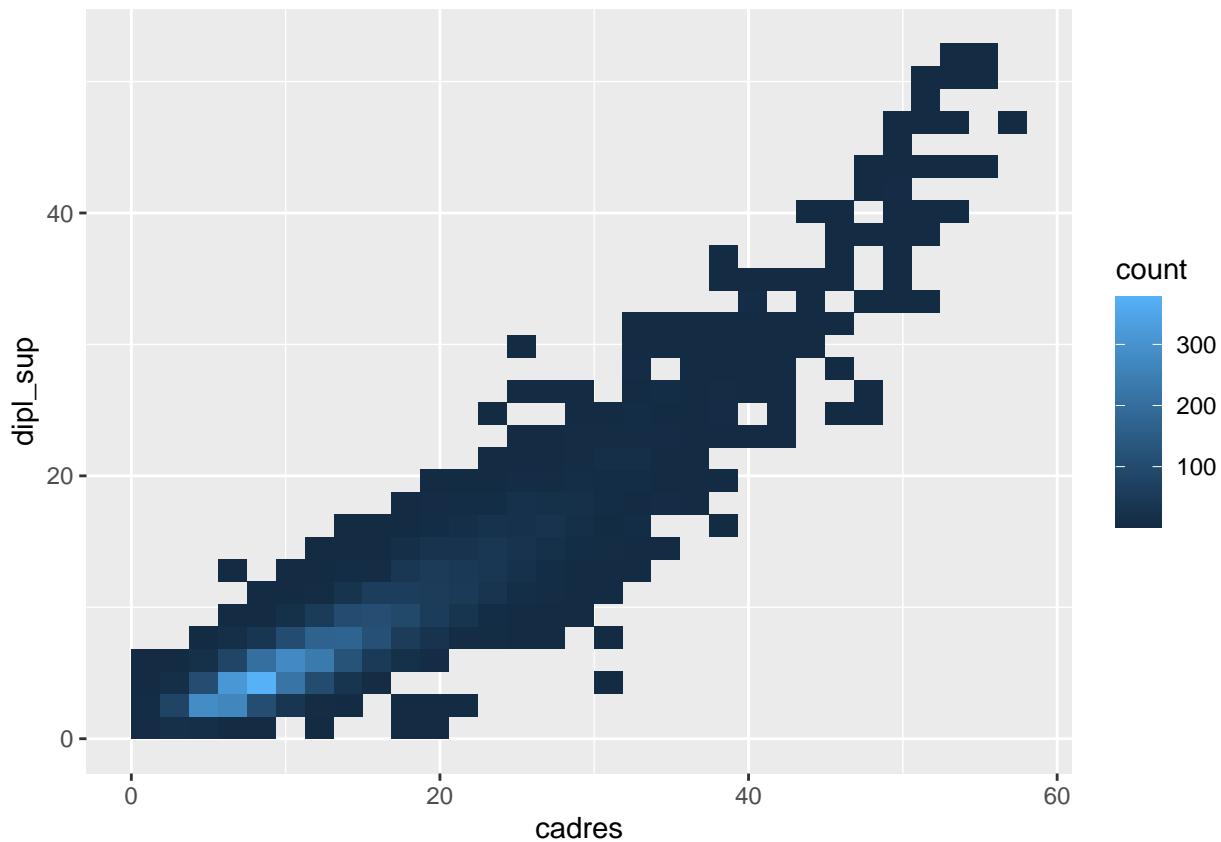
Lorsque le nombre de points est important, la représentation sous forme de nuage peut vite devenir illisible : la superposition des données empêche de voir précisément leur répartition.

```
ggplot(rp2018) + geom_point(aes(x = cadres, y = dipl_sup))
```



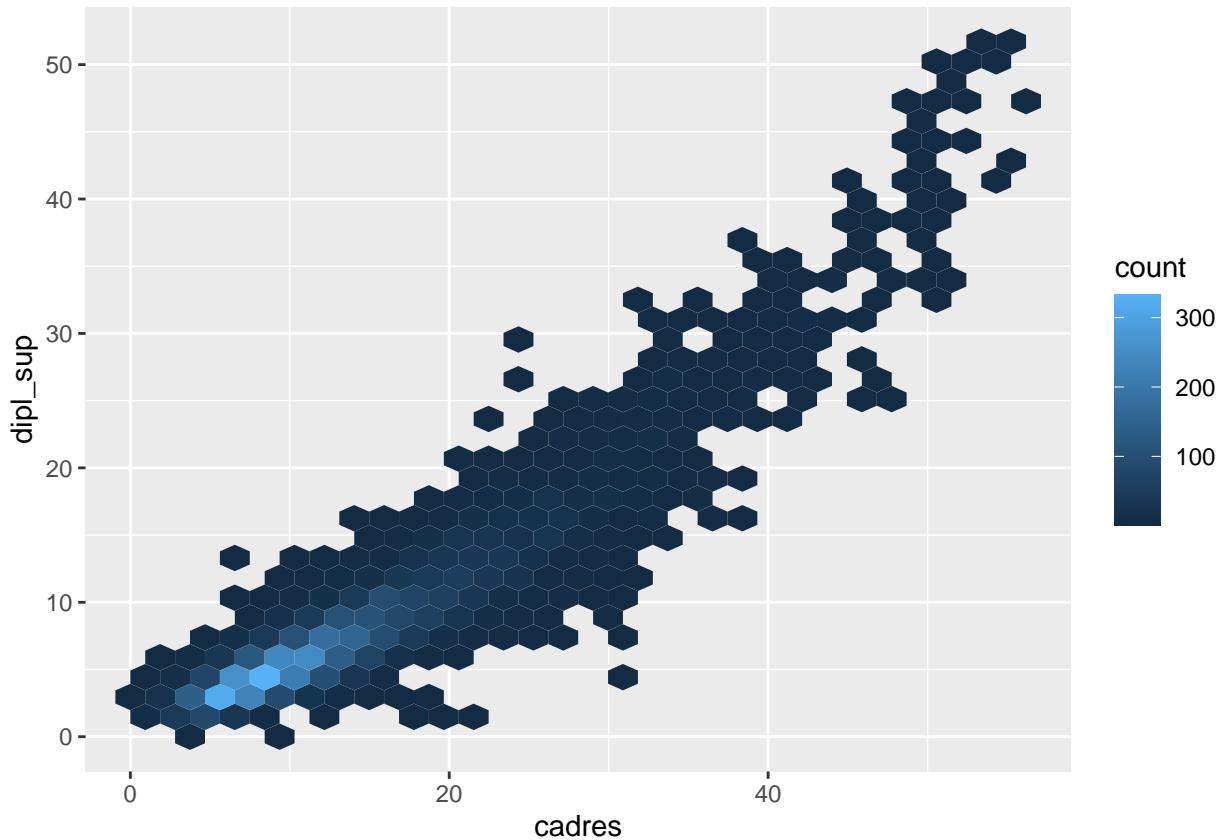
Dans ces cas-là, on peut utiliser `geom_bin2d`, qui va créer une grille sur toute la zone du graphique et colorier chaque carré selon le nombre de points qu'il contient (les carrés n'en contenant aucun restant transparents).

```
ggplot(rp2018) +
  geom_bin2d(aes(x = cadres, y = dipl_sup))
```



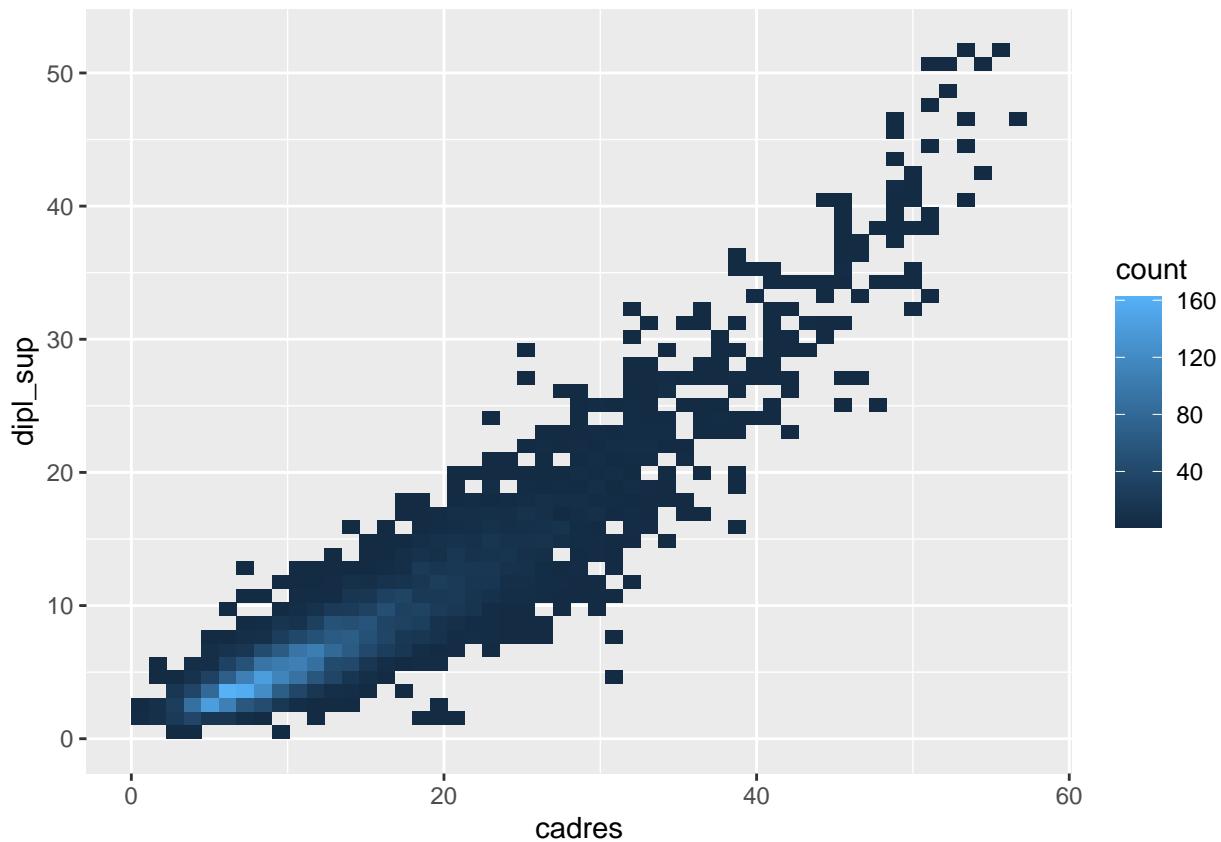
Une variante fonctionnant de manière très semblable est `geom_hex`, qui elle crée une grille constituée d'hexagones.

```
ggplot(rp2018) +  
  geom_hex(aes(x = cadres, y = dipl_sup))
```

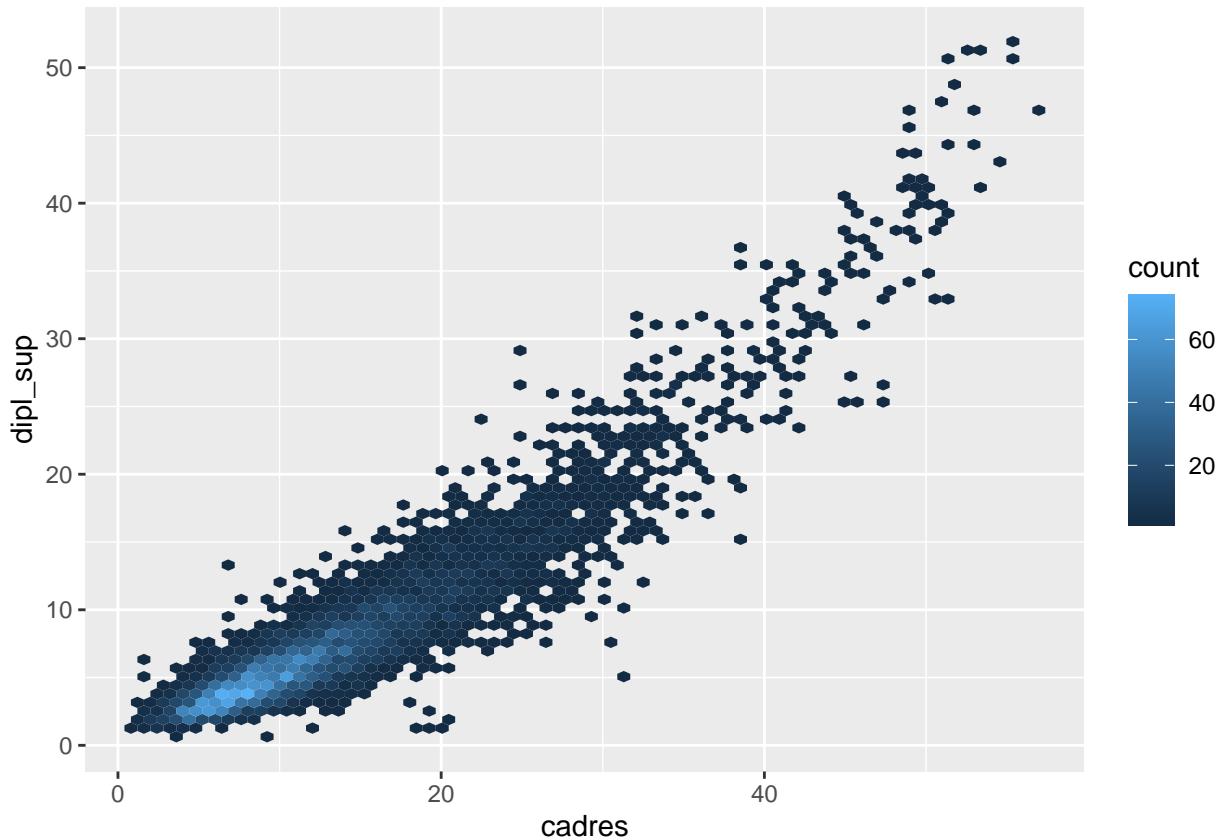


Dans les deux cas, on peut faire varier le nombre de zones, et donc la finesse du “quadrillage”, en utilisant l’argument `bins` (dont la valeur par défaut est 30).

```
ggplot(rp2018) +
  geom_bin2d(
    aes(x = cadres, y = dipl_sup),
    bins = 50
  )
```



```
ggplot(rp2018) +  
  geom_hex(  
    aes(x = cadres, y = dipl_sup),  
    bins = 70  
)
```



8.4 Mappages

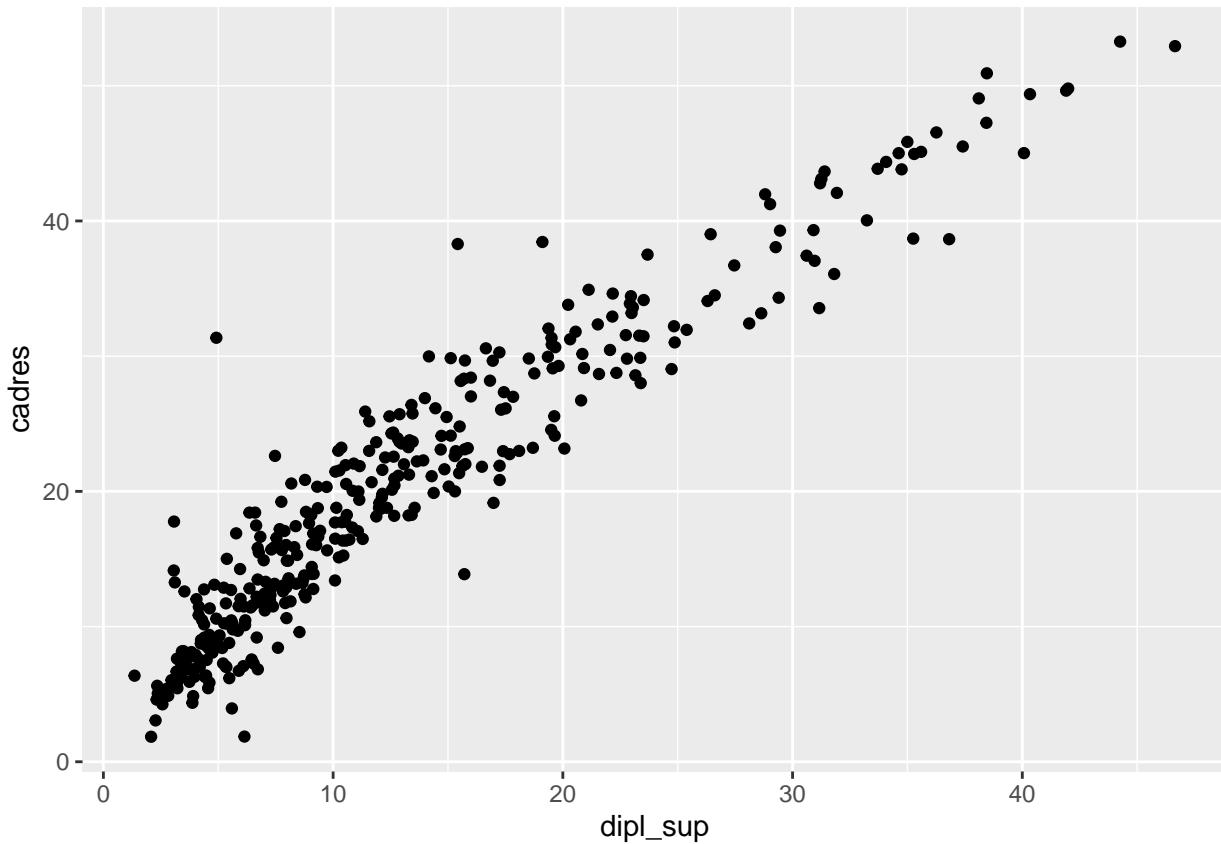
Un *mappage*, dans *ggplot2*, est une mise en relation entre un **attribut graphique** du *geom* (position, couleur, taille...) et une **variable** du tableau de données.

Ces mappages sont passés aux différents *geom* via la fonction *aes()* (abréviation d'*aesthetic*).

8.4.1 Exemples de mappages

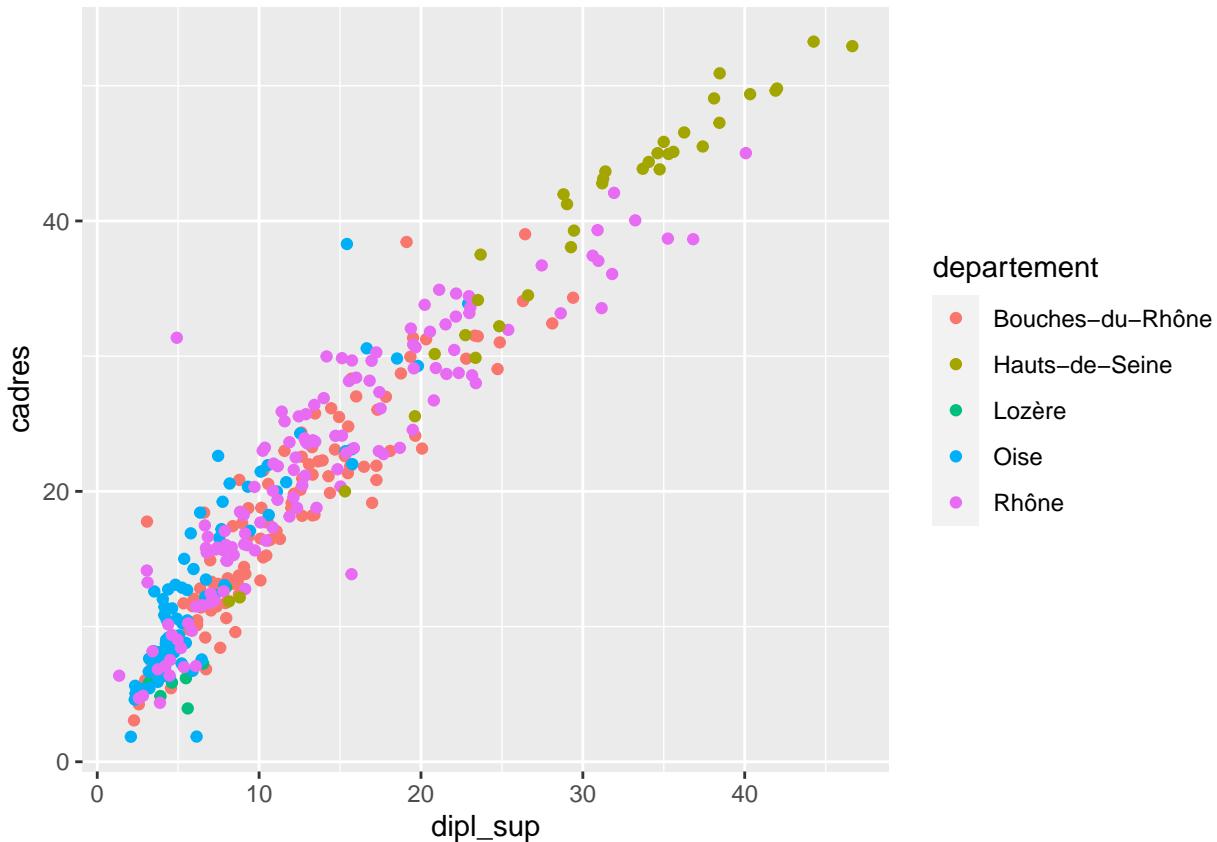
On a déjà vu les mappages *x* et *y* pour un nuage de points. Ceux-ci signifient que la position d'un point donné horizontalement (*x*) et verticalement (*y*) dépend de la valeur des variables passées comme arguments *x* et *y* dans *aes()*.

```
ggplot(rp) +
  geom_point(
    aes(x = dipl_sup, y = cadres)
  )
```



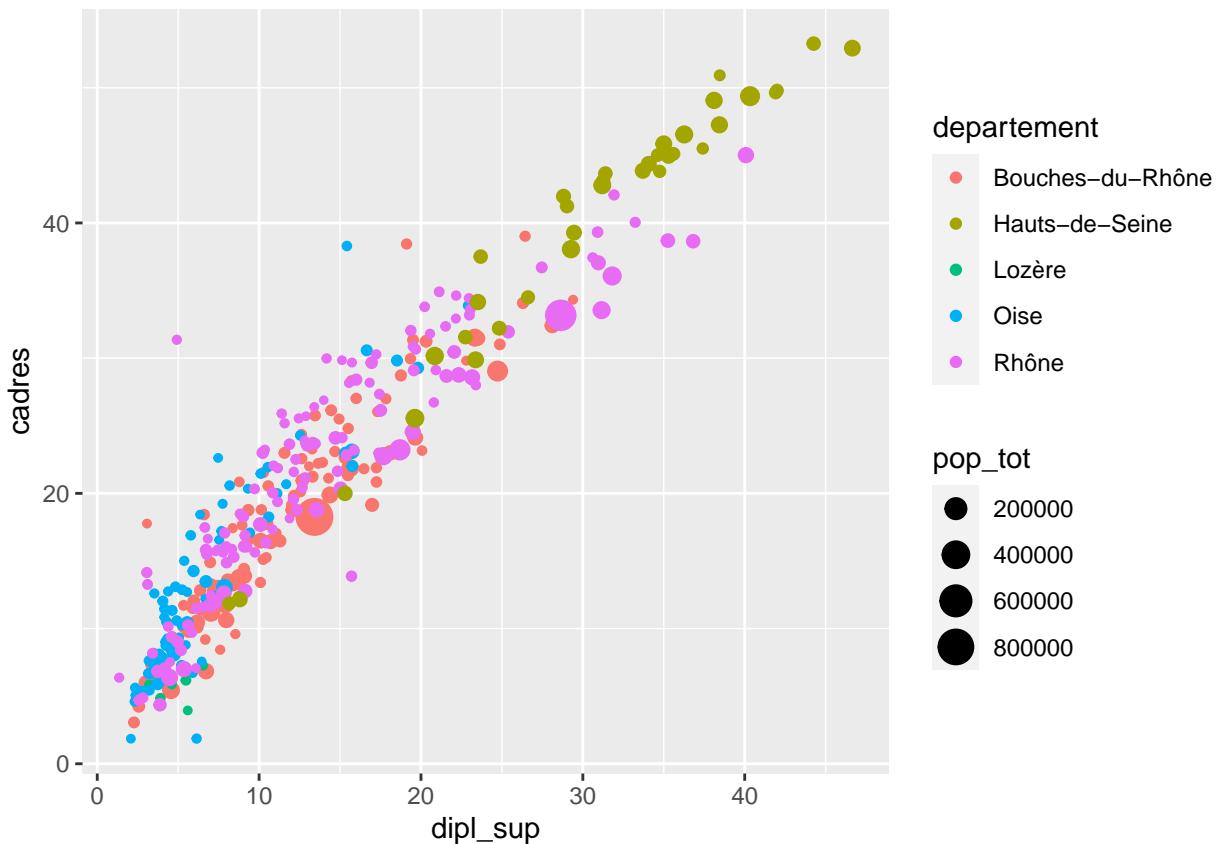
Mais on peut ajouter d'autres mappages. Par exemple, `color` permet de faire varier la couleur des points automatiquement en fonction des valeurs d'une troisième variable. Ainsi, on peut vouloir colorer les points selon le département de la commune correspondante.

```
ggplot(rp) +
  geom_point(
    aes(x = dipl_sup, y = cadres, color = departement)
  )
```



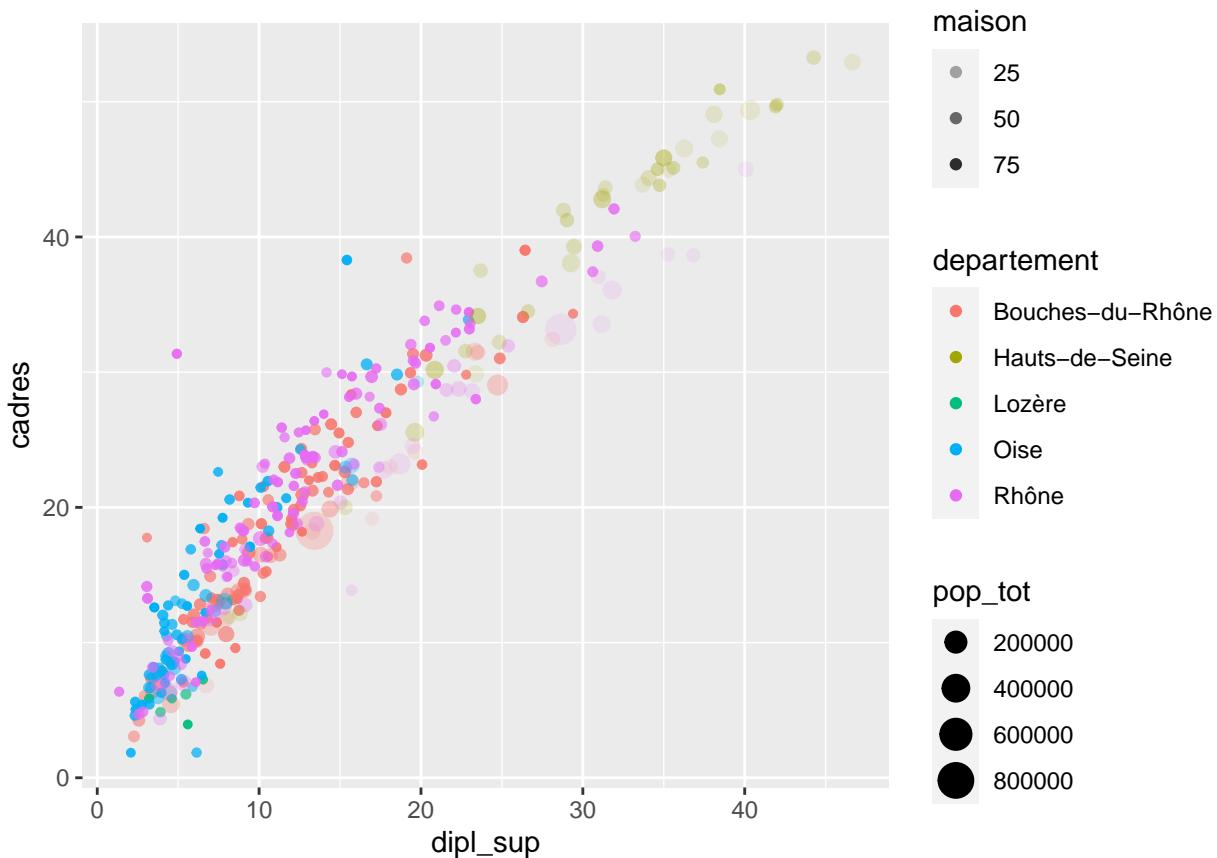
On peut aussi faire varier la taille des points avec `size`. Ici, la taille dépend de la population totale de la commune :

```
ggplot(rp) +  
  geom_point(  
    aes(x = dipl_sup, y = cadres, color = departement, size = pop_tot)  
  )
```



On peut même associer la transparence des points à une variable avec `alpha` :

```
ggplot(rp) +  
  geom_point(  
    aes(x = dipl_sup, y = cadres, color = departement, size = pop_tot, alpha = maison)  
  )
```

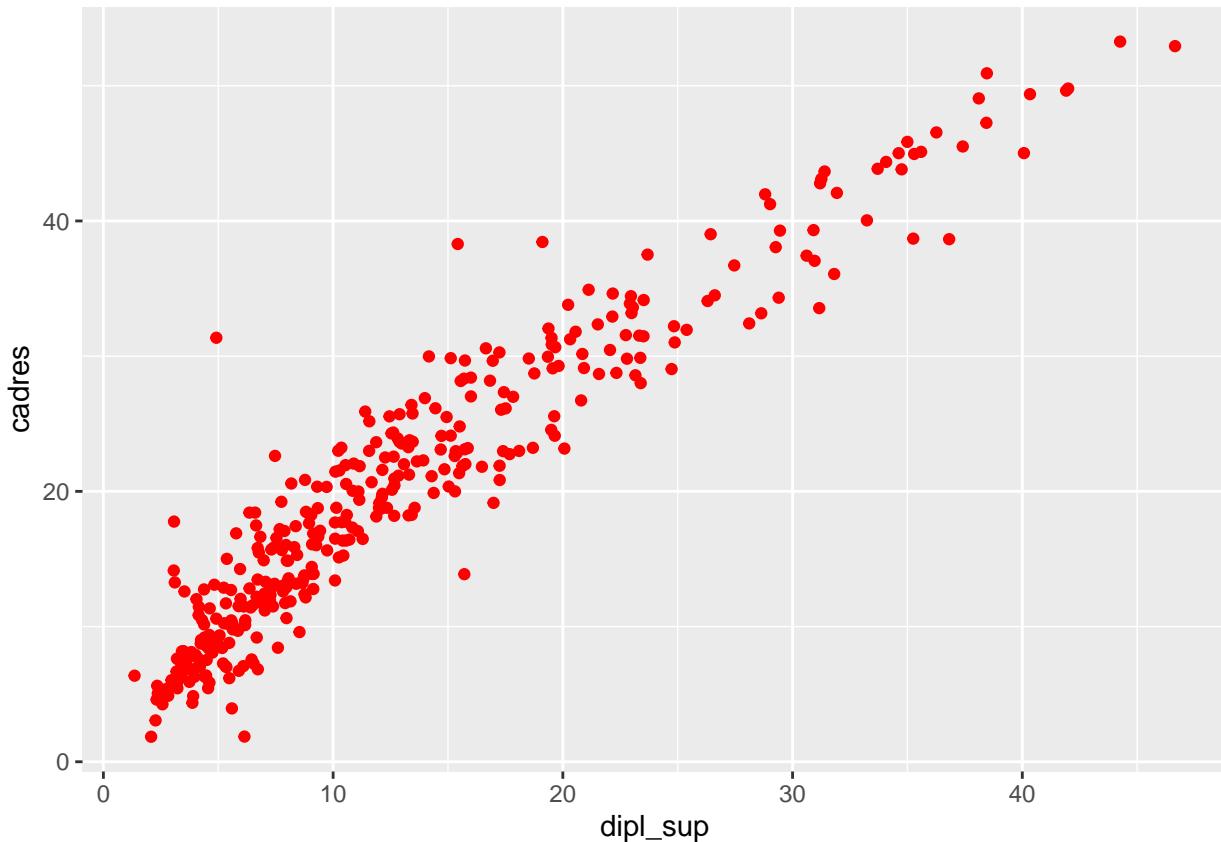


Chaque `geom` possède sa propre liste de mappages.

8.4.2 `aes()` or not `aes()` ?

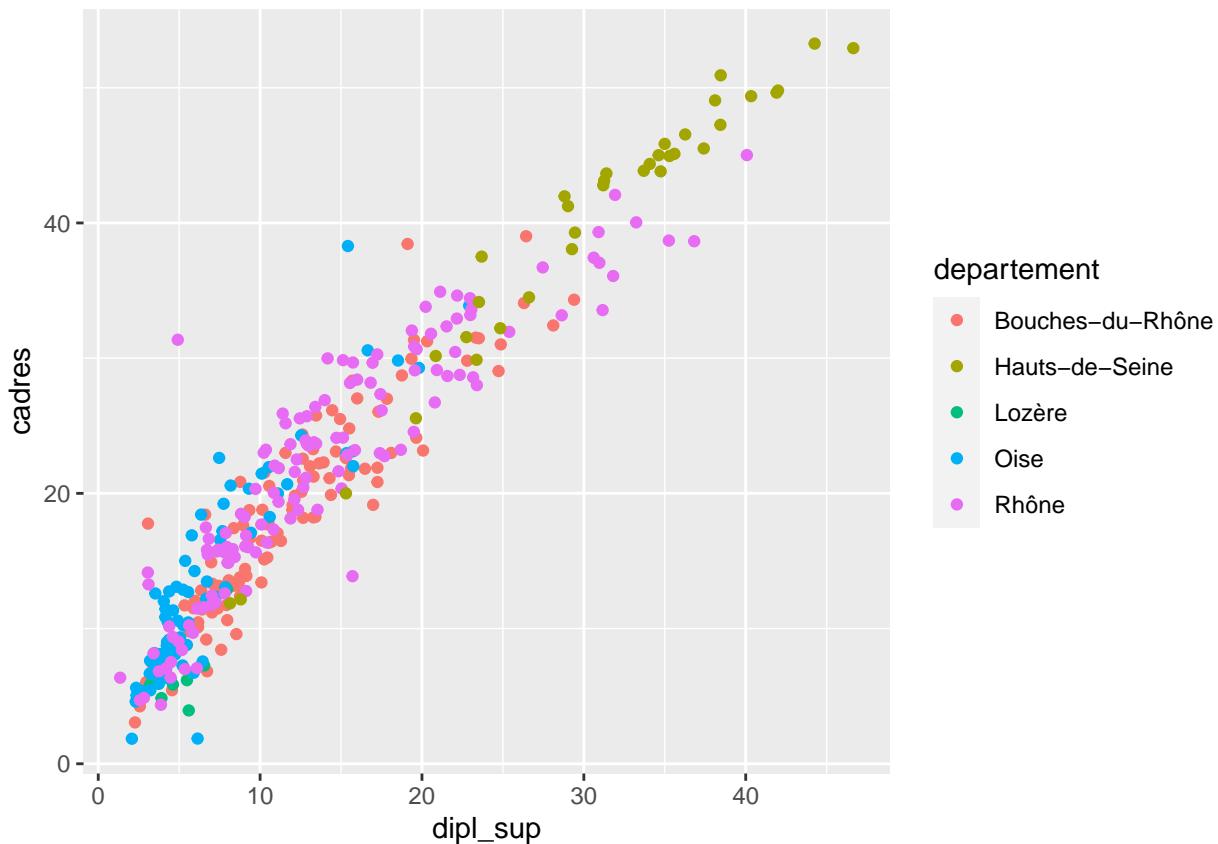
Comme on l'a déjà vu, parfois on souhaite changer un attribut sans le relier à une variable : c'est le cas par exemple si on veut représenter tous les points en rouge. Dans ce cas on utilise toujours l'attribut `color`, mais comme il ne s'agit pas d'un mappage, on le définit à l'**extérieur** de la fonction `aes()`.

```
ggplot(rp) +
  geom_point(
    aes(x = diplo_sup, y = cadres),
    color = "red"
  )
```



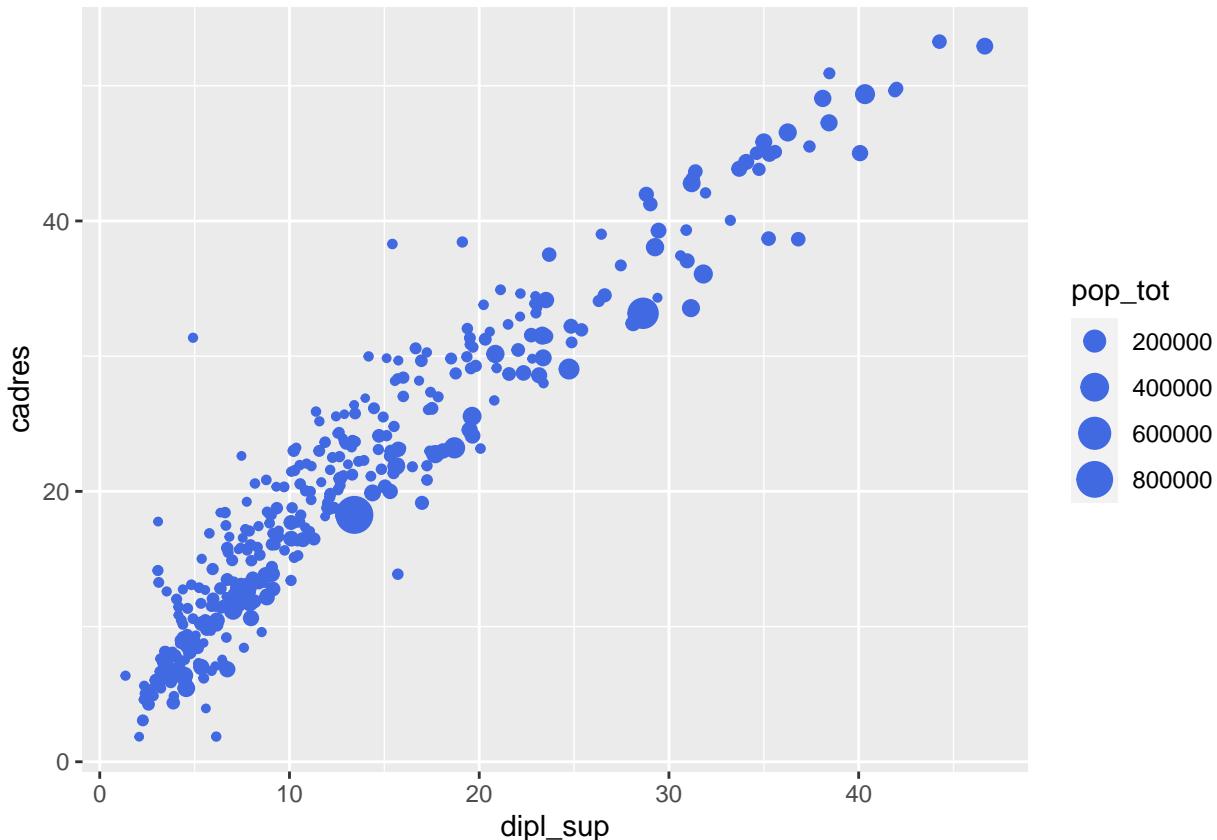
Par contre, si on veut faire varier la couleur en fonction des valeurs prises par une variable, on réalise un mappage, et on doit donc placer l'attribut `color` à l'intérieur de `aes()`.

```
ggplot(rp) +  
  geom_point(  
    aes(x = dipl_sup, y = cadres, color = departement)  
  )
```



On peut mélanger attributs liés à une variable (mappage, donc dans `aes()`) et attributs constants (donc à l'extérieur). Dans l'exemple suivant, la taille varie en fonction de la variable `pop_tot`, mais la couleur est constante pour tous les points.

```
ggplot(rp) +
  geom_point(
    aes(x = dipl_sup, y = cadres, size = pop_tot),
    color = "royalblue"
  )
```



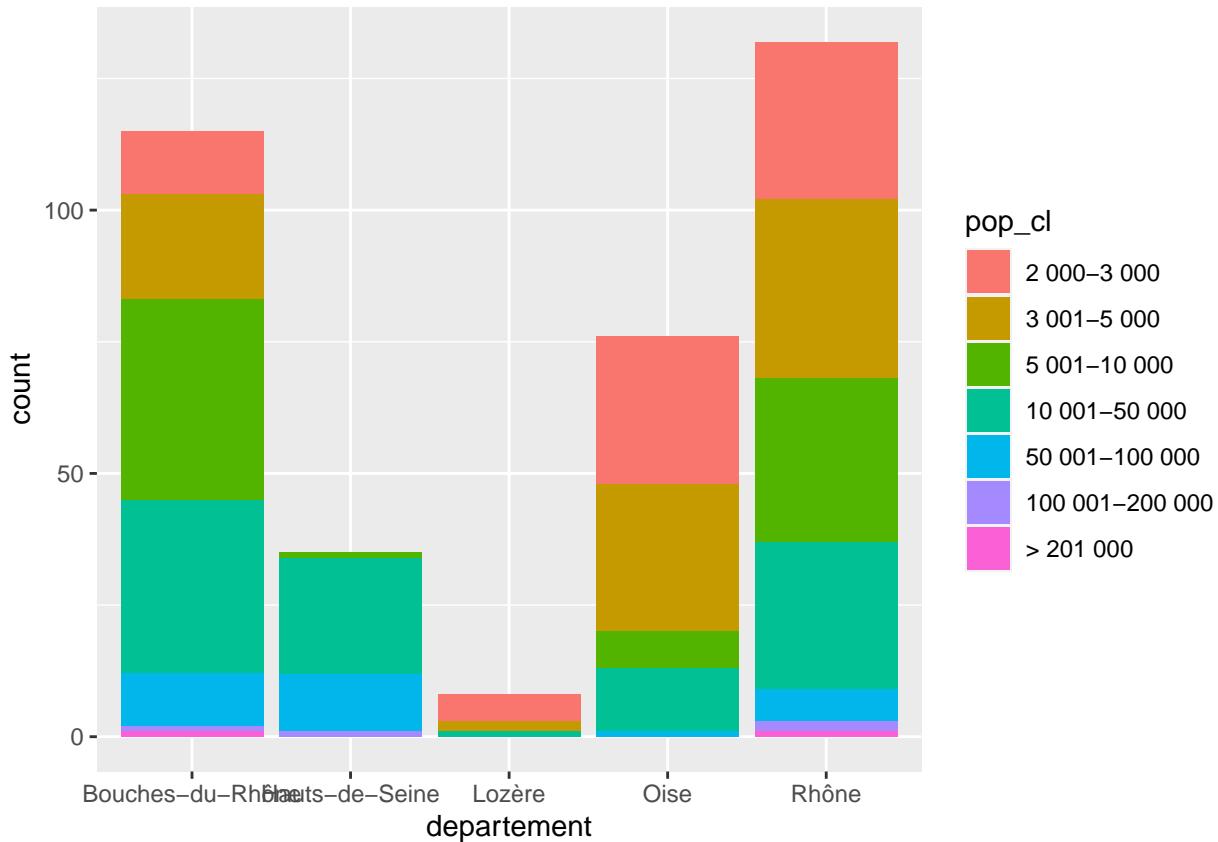
La règle est donc simple mais très importante :

Si on établit un lien entre les valeurs d'une variable et un attribut graphique, on définit un mappage, et on le déclare dans `aes()`. Sinon, on modifie l'attribut de la même manière pour tous les points, et on le définit en-dehors de la fonction `aes()`.

8.4.3 geom_bar et position

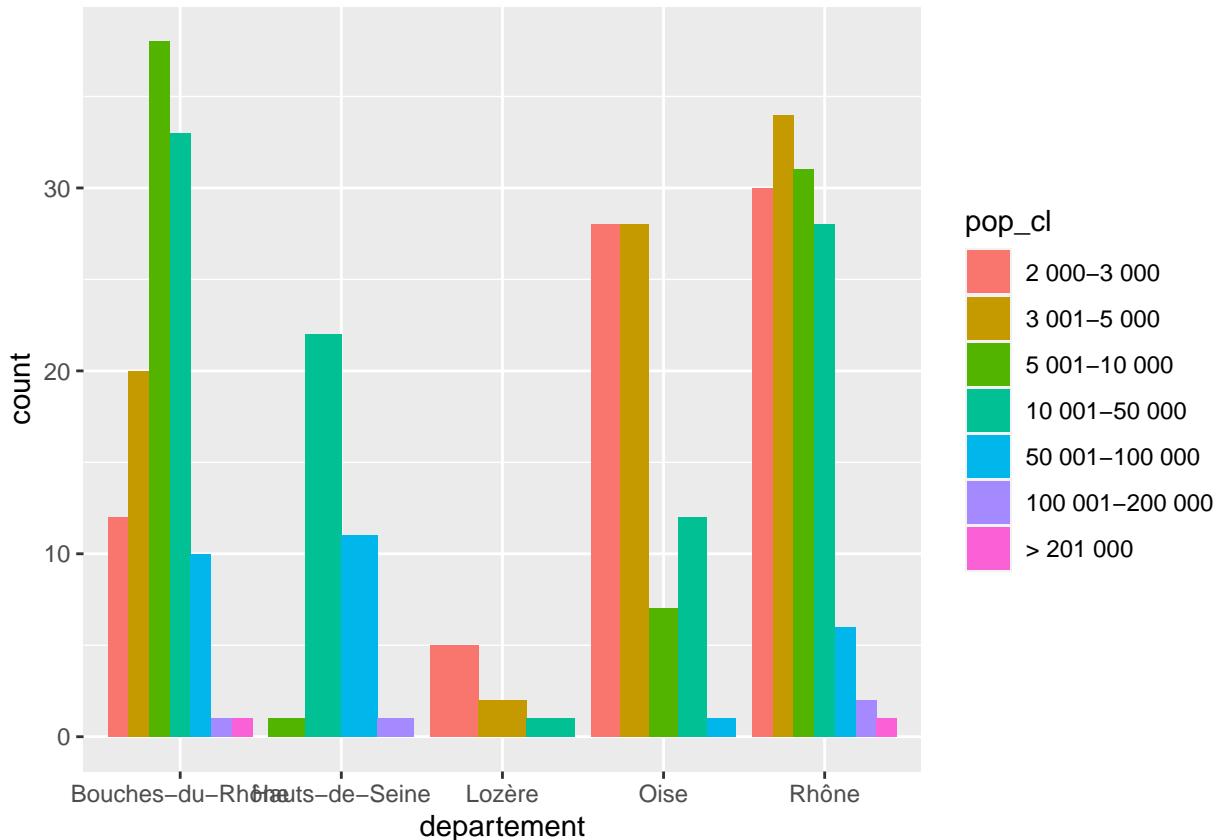
Un des mappages possibles de `geom_bar` est l'attribut `fill`, qui permet de tracer des barres de couleur différentes selon les modalités d'une deuxième variable :

```
ggplot(rp) + geom_bar(aes(x = departement, fill = pop_c1))
```



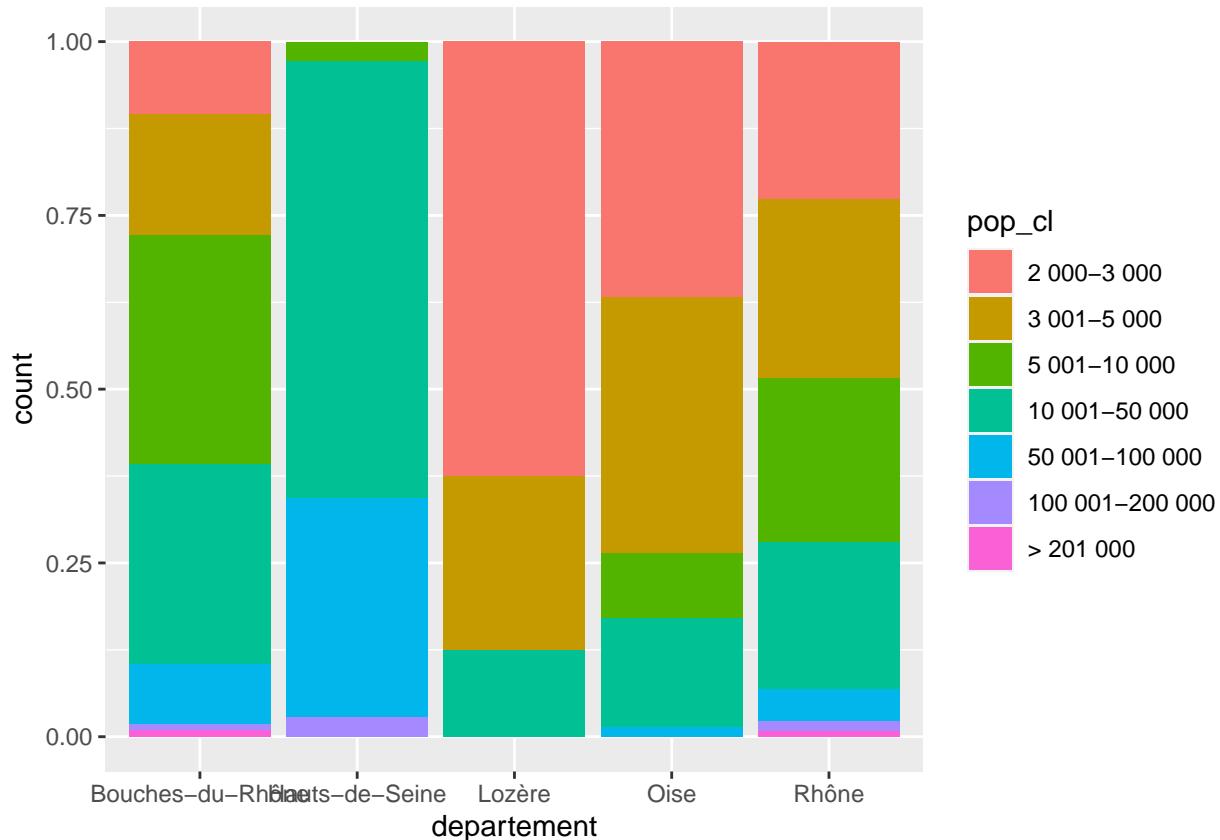
L'attribut `position` de `geom_bar` permet d'indiquer comment les différentes barres doivent être positionnées. Par défaut l'argument vaut `position = "stack"` et elles sont donc "empilées". Mais on peut préciser `position = "dodge"` pour les mettre côté à côté.

```
ggplot(rp) +
  geom_bar(
    aes(x = departement, fill = pop_cl),
    position = "dodge"
  )
```



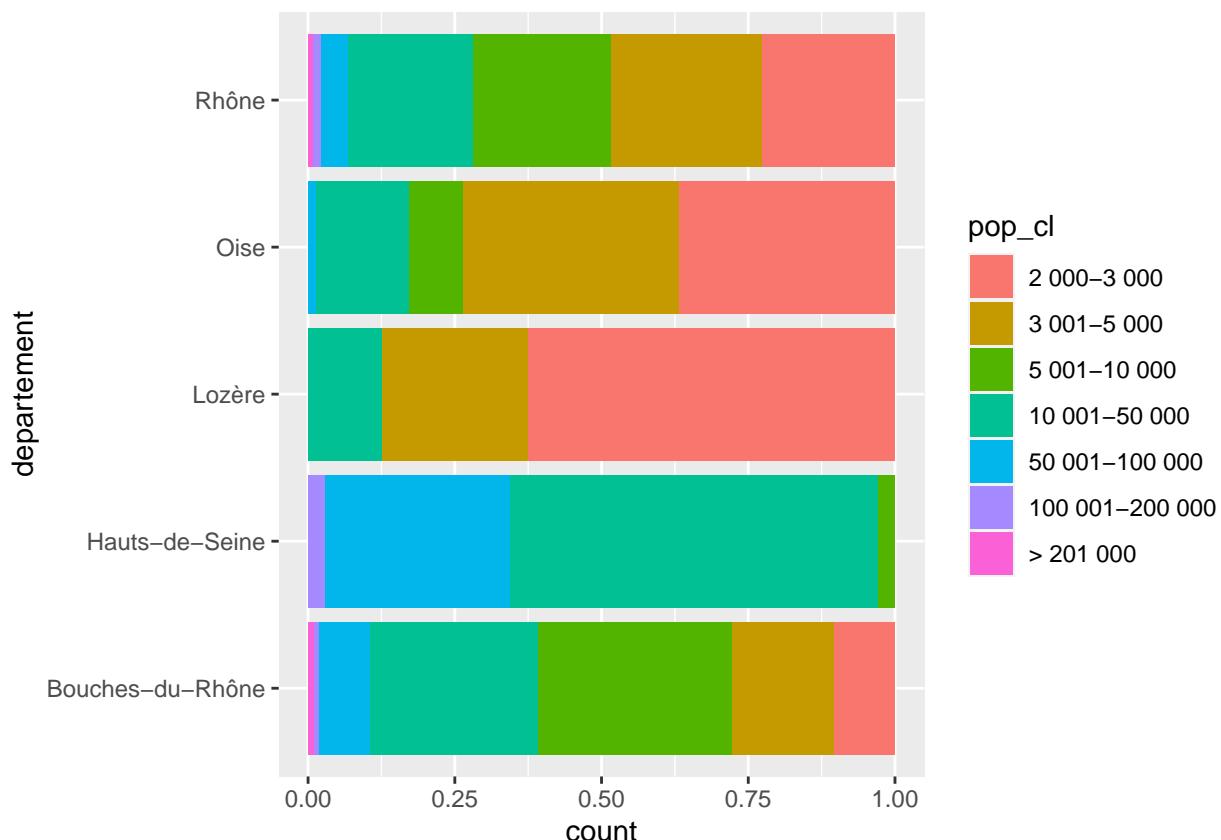
Ou encore `position = "fill"` pour représenter non plus des effectifs, mais des proportions.

```
ggplot(rp) +
  geom_bar(
    aes(x = departement, fill = pop_cl),
    position = "fill"
  )
```



Là encore, on peut utiliser `coord_flip()` si on souhaite une visualisation avec des barres horizontales.

```
ggplot(rp) +
  geom_bar(
    aes(x = departement, fill = pop_cl),
    position = "fill"
  ) +
  coord_flip()
```

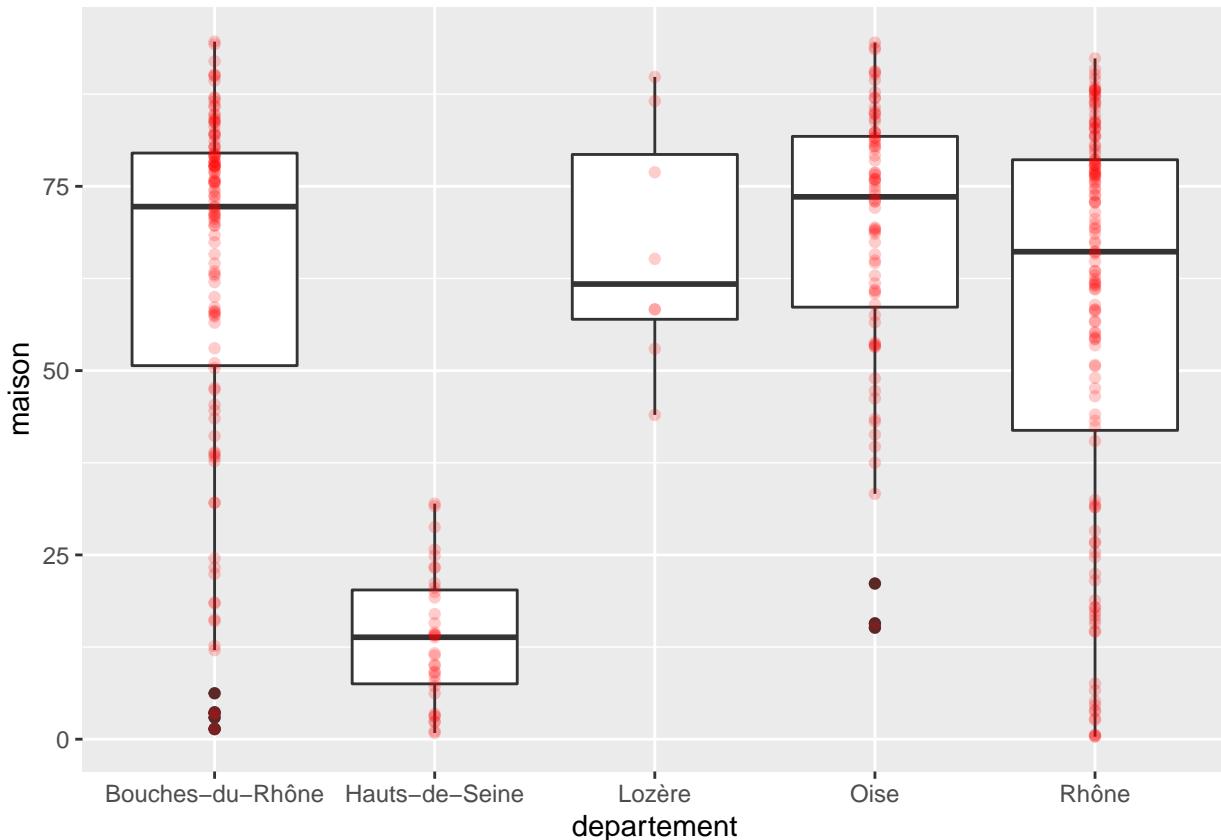


8.5 Représentation de plusieurs geom

On peut représenter plusieurs `geom` simultanément sur un même graphique, il suffit de les ajouter à tour de rôle avec l'opérateur `+`.

Par exemple, on peut superposer la position des points au-dessus d'un boxplot. On va pour cela ajouter un `geom_point` après avoir ajouté notre `geom_boxplot`.

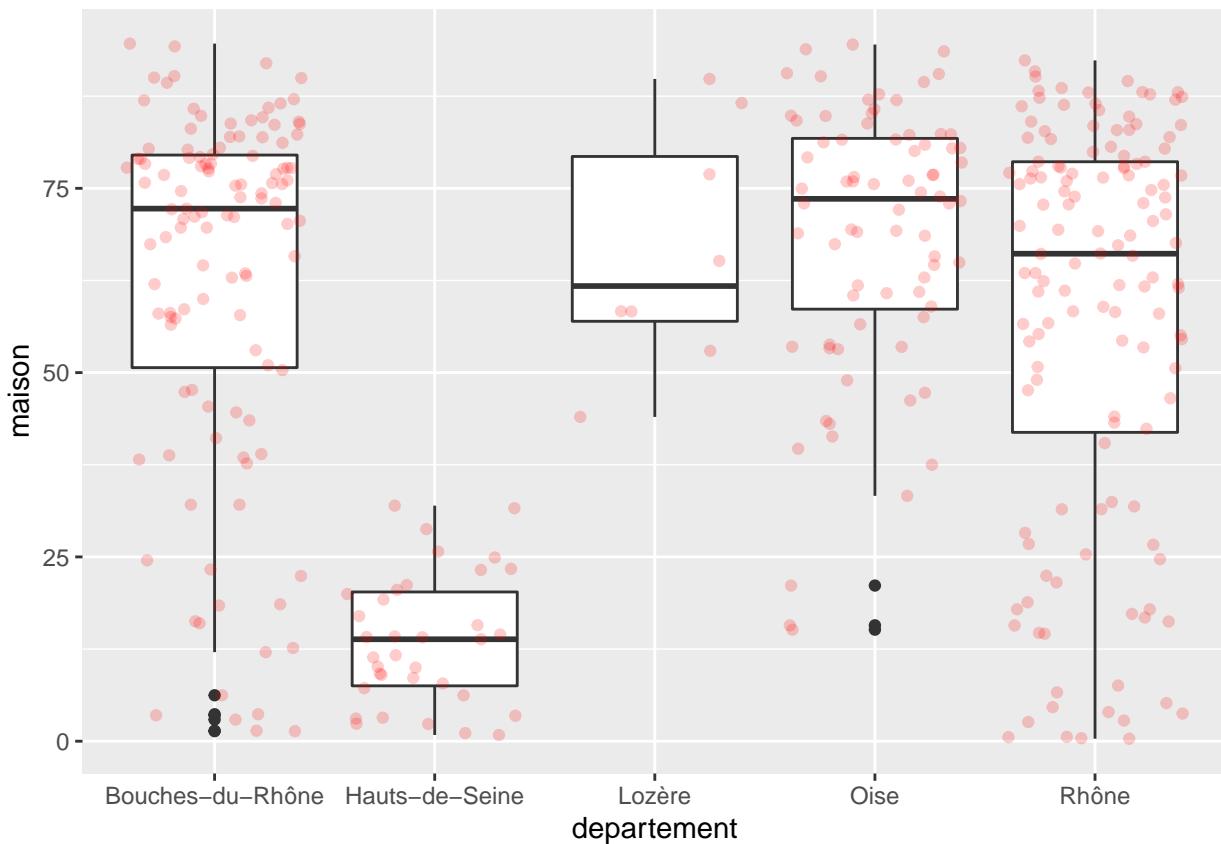
```
ggplot(rp) +
  geom_boxplot(aes(x = département, y = maison)) +
  geom_point(
    aes(x = département, y = maison),
    col = "red", alpha = 0.2
  )
```



Quand une commande `ggplot2` devient longue, il peut être plus lisible de la répartir sur plusieurs lignes. Dans ce cas, il faut penser à placer l'opérateur `+` en fin de ligne, afin que R comprenne que la commande n'est pas complète et qu'il prenne en compte la suite.

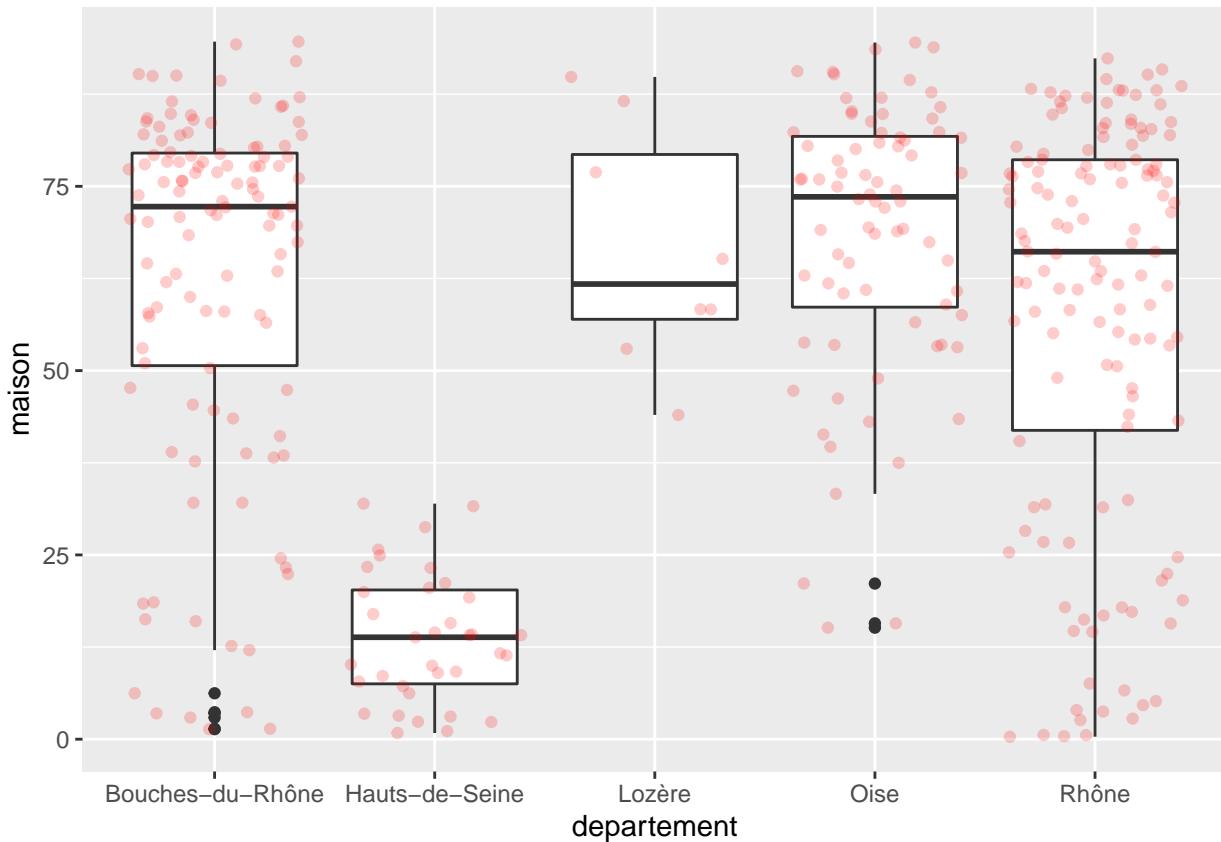
Pour un résultat un peu plus lisible, on peut remplacer `geom_point` par `geom_jitter`, qui disperse les points horizontalement et facilite leur visualisation.

```
ggplot(rp) +
  geom_boxplot(aes(x = departement, y = maison)) +
  geom_jitter(
    aes(x = departement, y = maison),
    col = "red", alpha = 0.2
  )
```



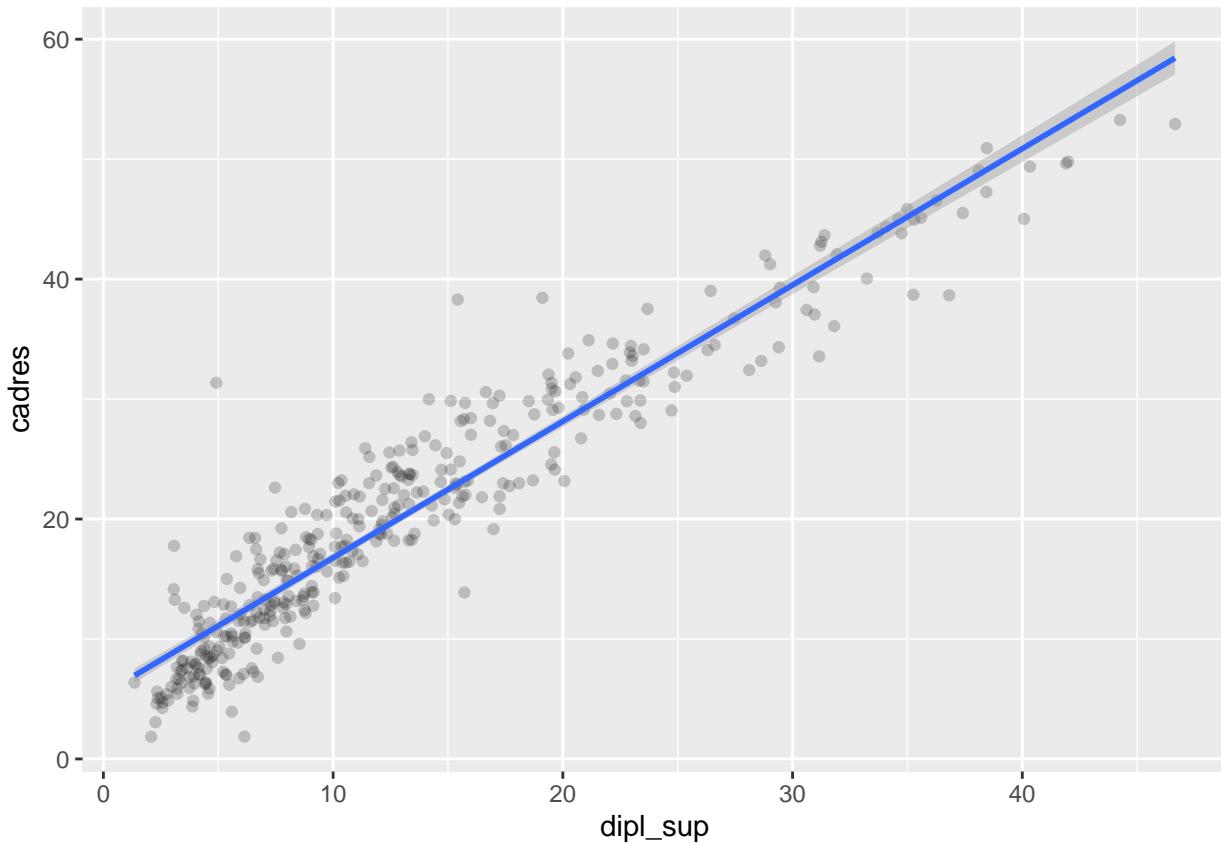
Pour simplifier un peu le code, plutôt que de déclarer les mappages dans chaque `geom`, on peut les déclarer dans l'appel à `ggplot()`. Ils seront automatiquement “hérités” par les `geom` ajoutés (sauf s'ils redéfinissent les mêmes mappages).

```
ggplot(rp, aes(x = departement, y = maison)) +  
  geom_boxplot() +  
  geom_jitter(color = "red", alpha = 0.2)
```



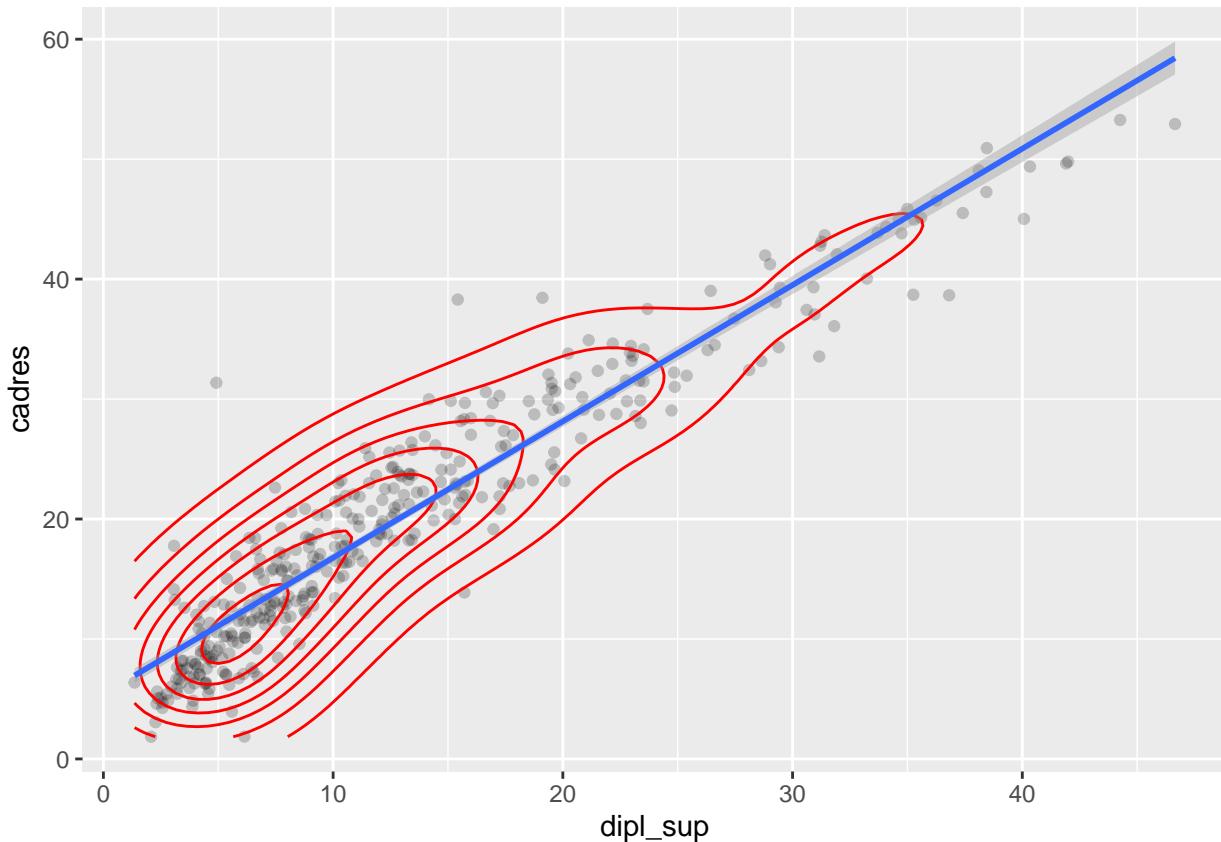
Autre exemple, on peut vouloir ajouter à un nuage de points une ligne de régression linéaire à l'aide de `geom_smooth` :

```
ggplot(rp, aes(x = dipl_sup, y = cadres)) +
  geom_point(alpha = 0.2) +
  geom_smooth(method = "lm")
#> `geom_smooth()` using formula 'y ~ x'
```



Et on peut même superposer une troisième visualisation de la répartition des points dans l'espace avec `geom_density2d` :

```
ggplot(rp, aes(x = dipl_sup, y = cadres)) +
  geom_point(alpha = 0.2) +
  geom_density2d(color = "red") +
  geom_smooth(method = "lm")
#> `geom_smooth()` using formula 'y ~ x'
```



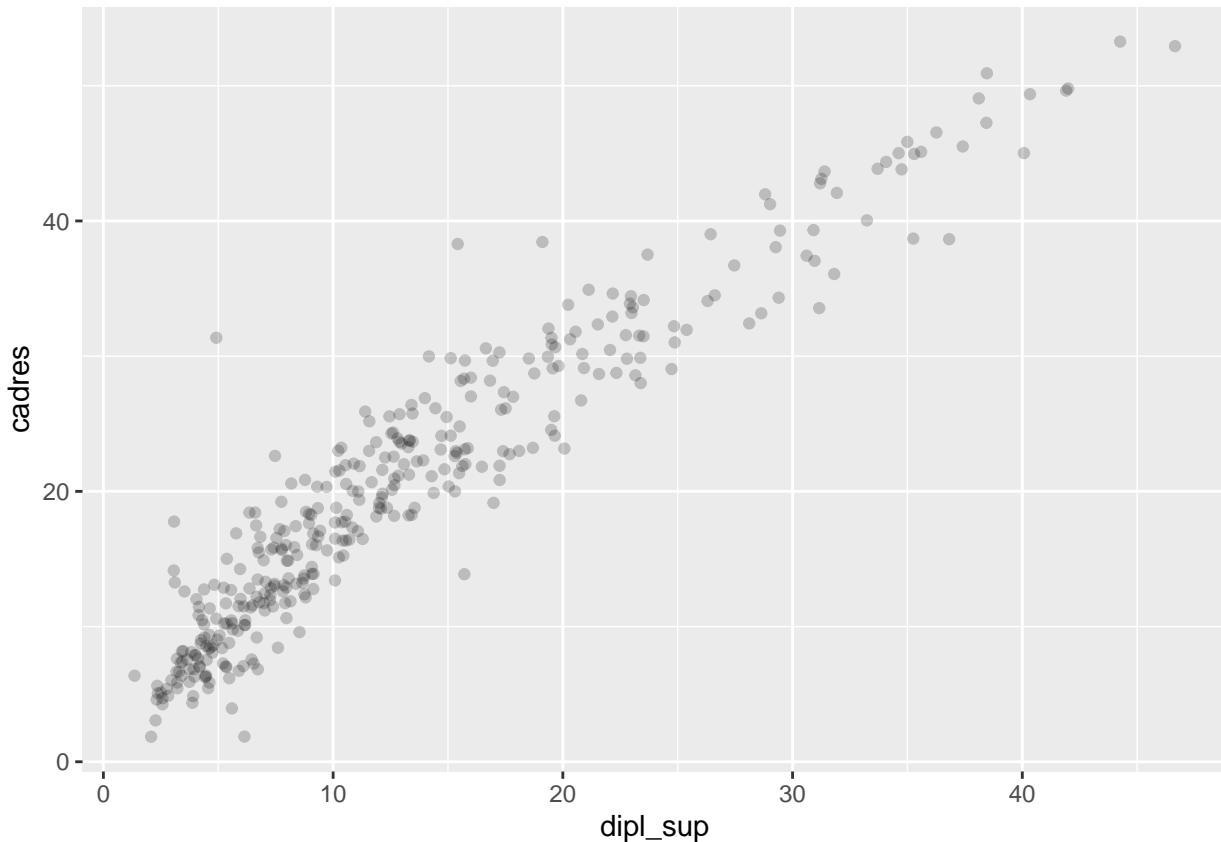
8.5.1 Plusieurs sources de données

On peut associer à différents `geom` des sources de données différentes. Supposons qu'on souhaite afficher sur un nuage de points les noms des communes de plus de 50000 habitants. On commence par créer un tableau de données contenant uniquement ces communes à l'aide de la fonction `filter`.

```
com50 <- filter(rp, pop_tot >= 50000)
```

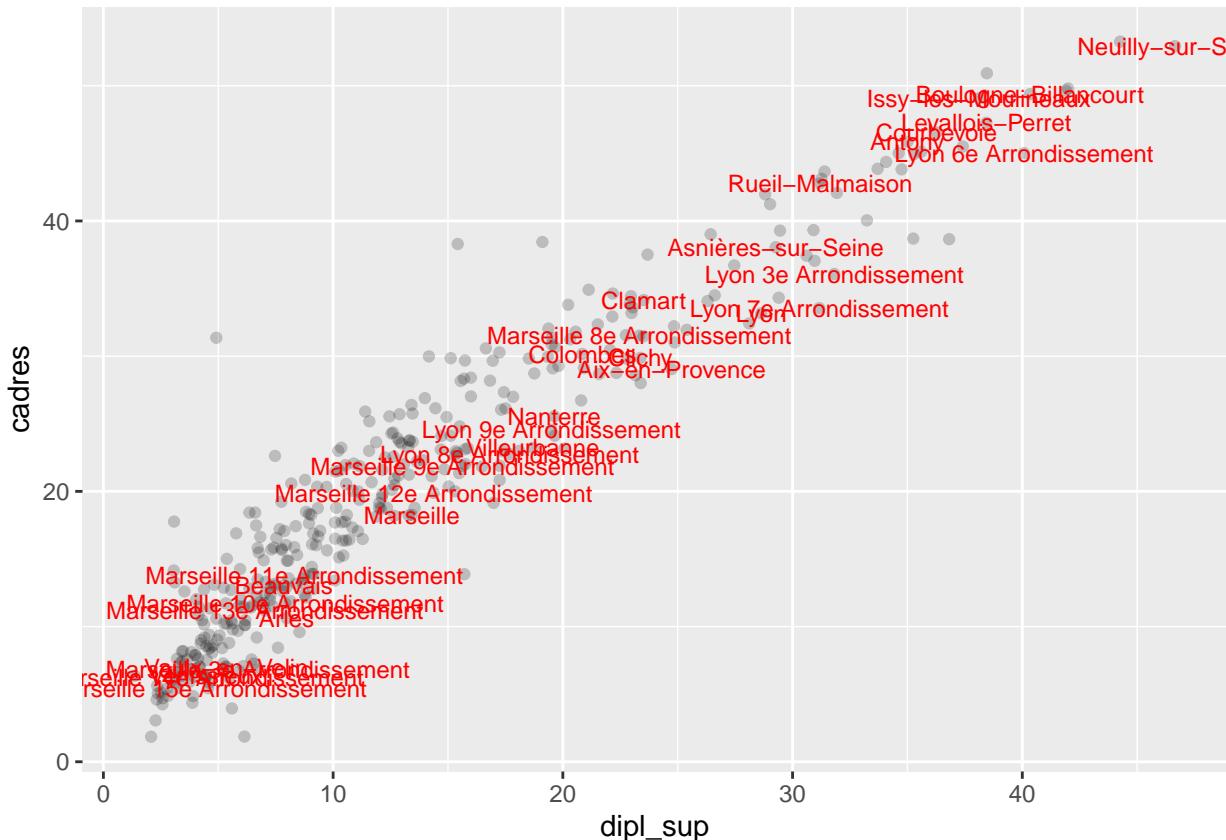
On fait ensuite le nuage de points comme précédemment :

```
ggplot(data = rp, aes(x = dipl_sup, y = cadres)) +
  geom_point(alpha = 0.2)
```



Pour superposer les noms de communes de plus de 50 000 habitants, on peut ajouter un `geom_text`, mais en spécifiant que les données proviennent du nouveau tableau `com50` et non de notre tableau initial `rp`. On le fait en passant un argument `data` spécifique à `geom_text` :

```
ggplot(data = rp, aes(x = dipl_sup, y = cadres)) +
  geom_point(alpha = 0.2) +
  geom_text(
    data = com50, aes(label = commune),
    color = "red", size = 3
  )
```



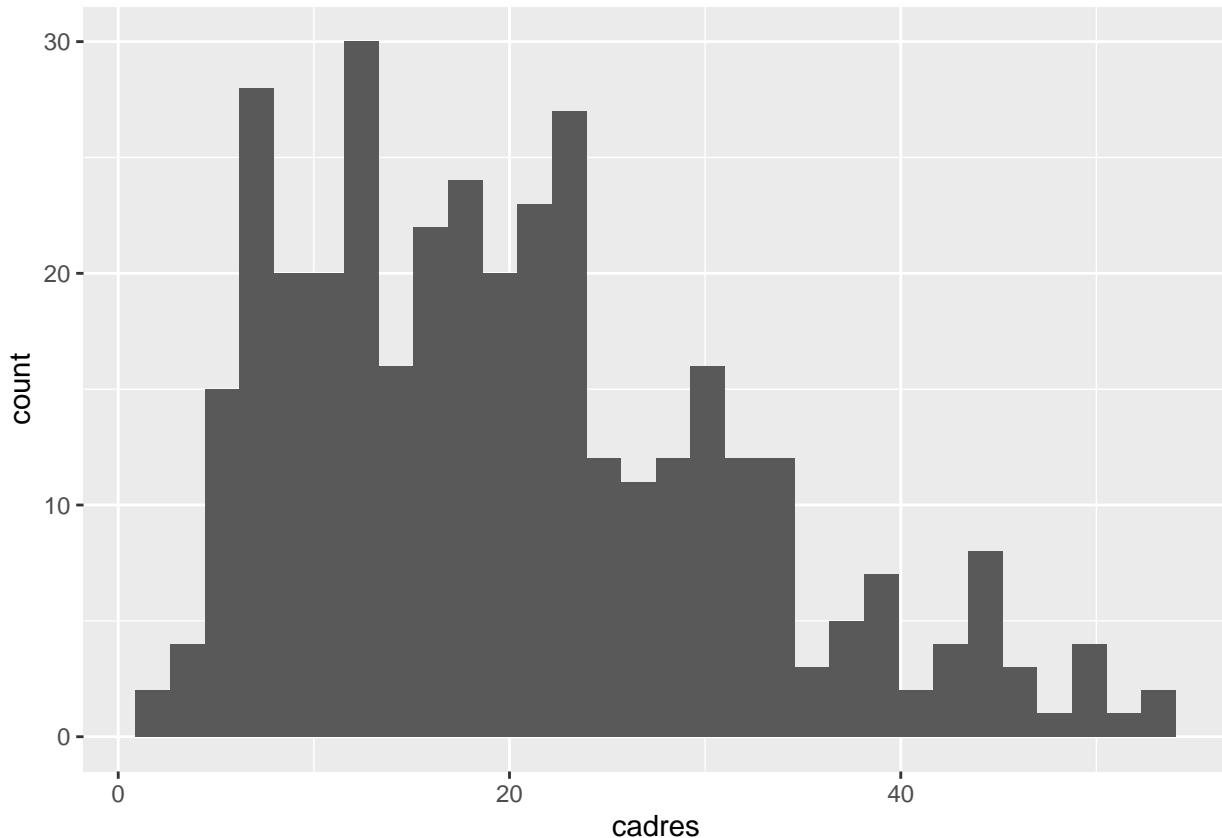
Ainsi, on obtient un graphique avec deux `geom` superposés, mais dont les données proviennent de deux tableaux différents.

8.6 Faceting

Le *faceting* permet d'effectuer plusieurs fois le même graphique selon les valeurs d'une ou plusieurs variables qualitatives.

Par exemple, on a vu qu'on peut représenter l'histogramme du pourcentage de cadres dans nos communes avec le code suivant :

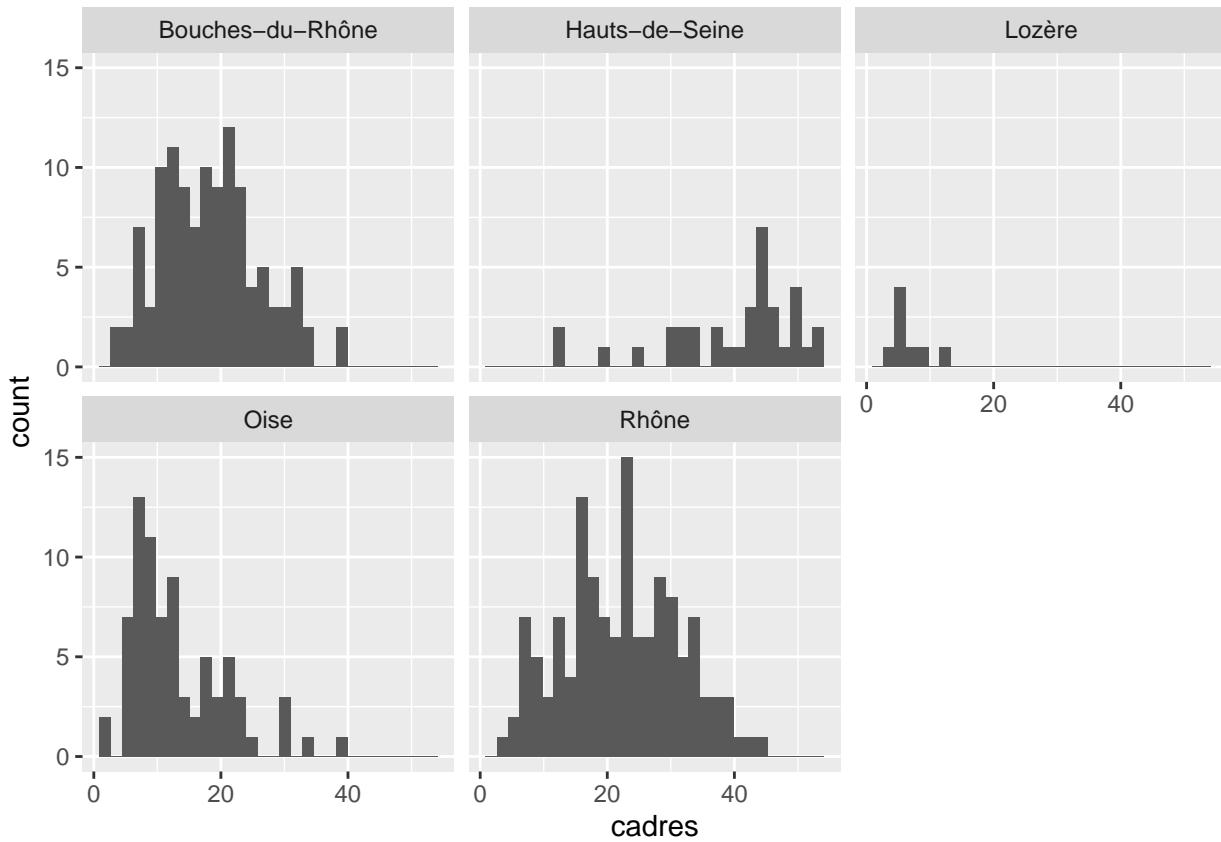
```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres))
```



On souhaite comparer cette distribution de la part des cadres selon le département, et donc faire un histogramme pour chacun de ces départements. C'est ce que permettent les fonctions `facet_wrap` et `facet_grid`.

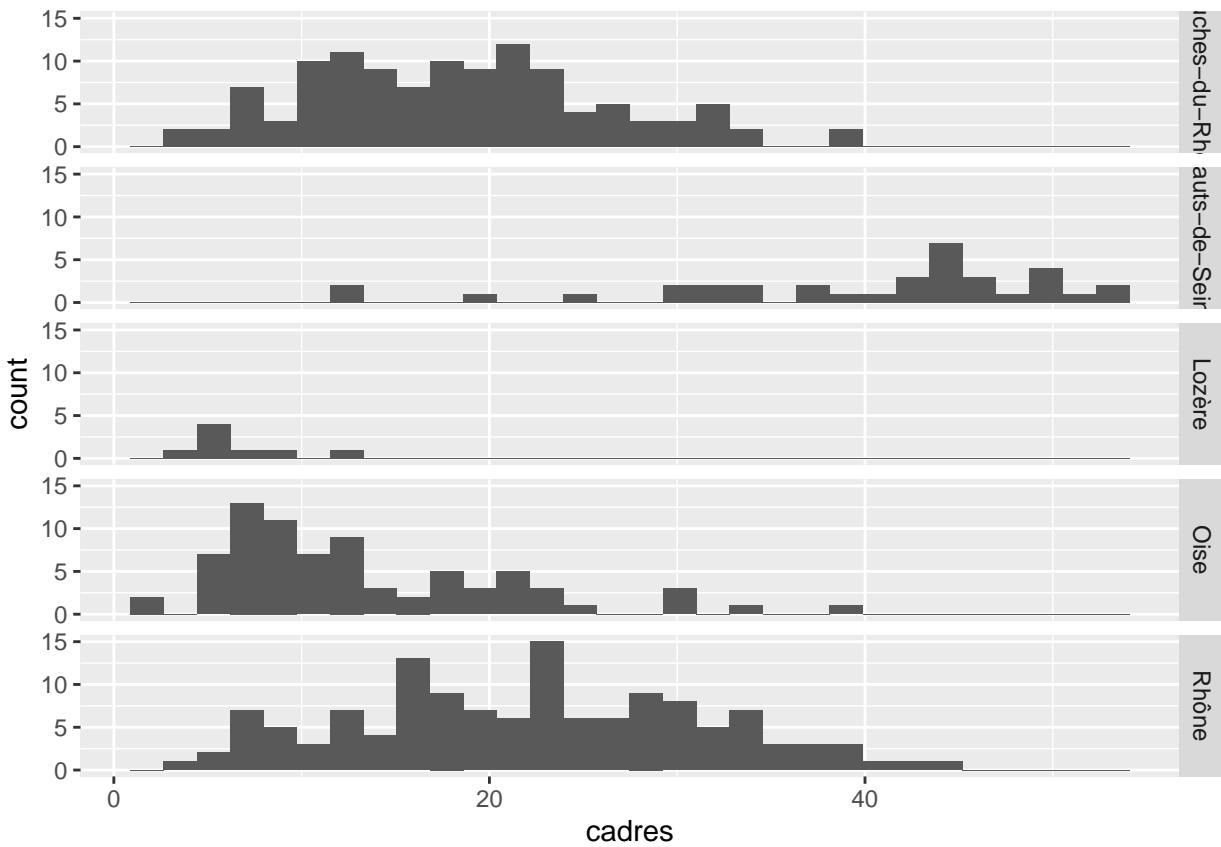
`facet_wrap` prend un paramètre de la forme `vars(variable)`, où `variable` est le nom de la variable en fonction de laquelle on souhaite faire les différents graphiques. Ceux-ci sont alors affichés les uns à côté des autres et répartis automatiquement dans la page.

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_wrap(vars(departement))
```



Pour `facet_grid`, les graphiques sont disposés selon une grille. La fonction prend alors deux arguments, `rows` et `cols`, auxquels on passe les variables à afficher en ligne ou en colonne via la fonction `vars()`.

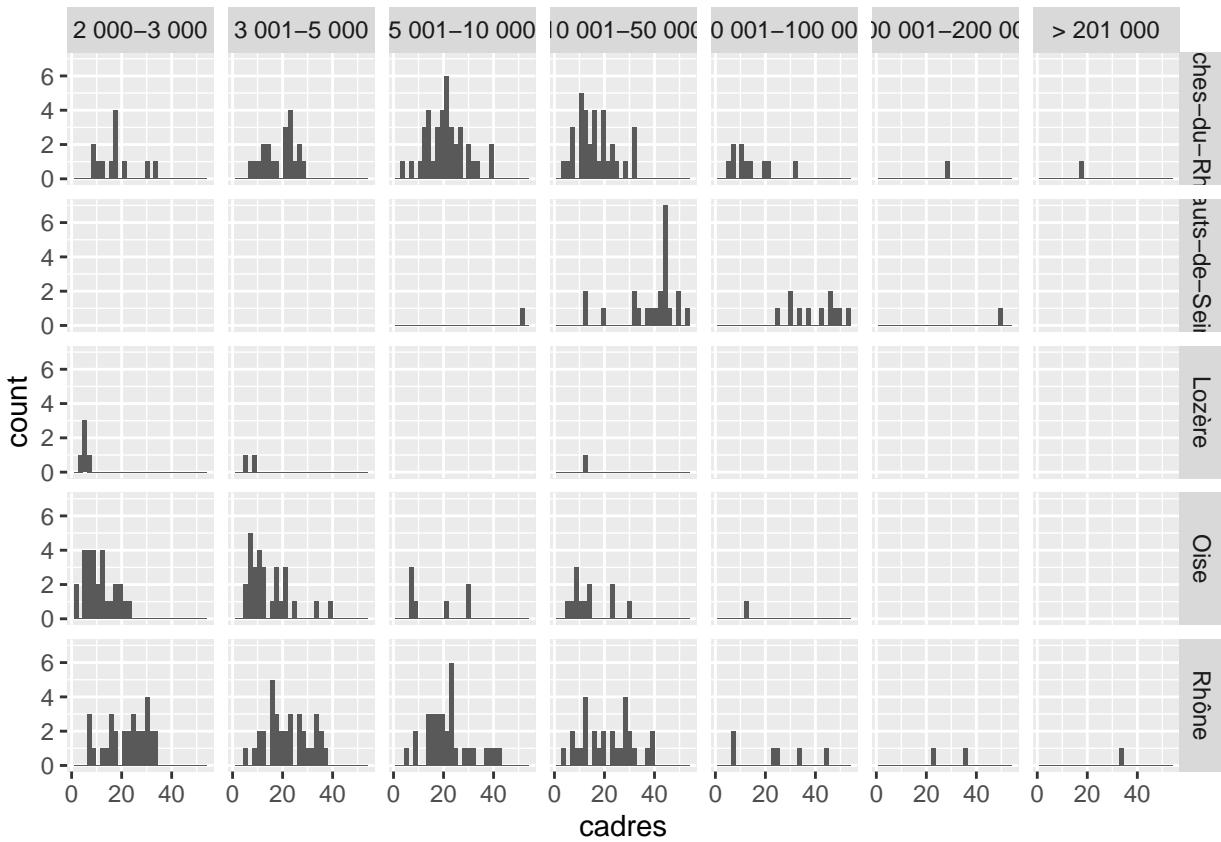
```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_grid(rows = vars(departement))
```



Un des intérêts du facetting dans `ggplot2` est que tous les graphiques générés ont les mêmes échelles, ce qui permet une comparaison directe.

Enfin, notons qu'on peut même faire du facetting sur plusieurs variables à la fois. On peut par exemple faire des histogrammes de la répartition de la part des cadres pour chaque croisement des variables `département` et `pop_cl` :

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  facet_grid(
    rows = vars(departement), cols = vars(pop_cl)
  )
```



L’histogramme en haut à gauche représente la répartition du pourcentage de cadres parmi les communes de 2000 à 3000 habitants dans les Bouches-du-Rhône, etc.

8.7 Scales

On a vu qu’avec *ggplot2* on définit des mappages entre des attributs graphiques (position, taille, couleur, etc.) et des variables d’un tableau de données. Ces mappages sont définis, pour chaque *geom*, via la fonction *aes()*.

Les *scales* dans *ggplot2* permettent de modifier la manière dont un attribut graphique va être relié aux valeurs d’une variable, et dont la légende correspondante va être affichée. Par exemple, pour l’attribut *color*, on pourra définir la palette de couleur utilisée. Pour *size*, les tailles minimales et maximales, etc.

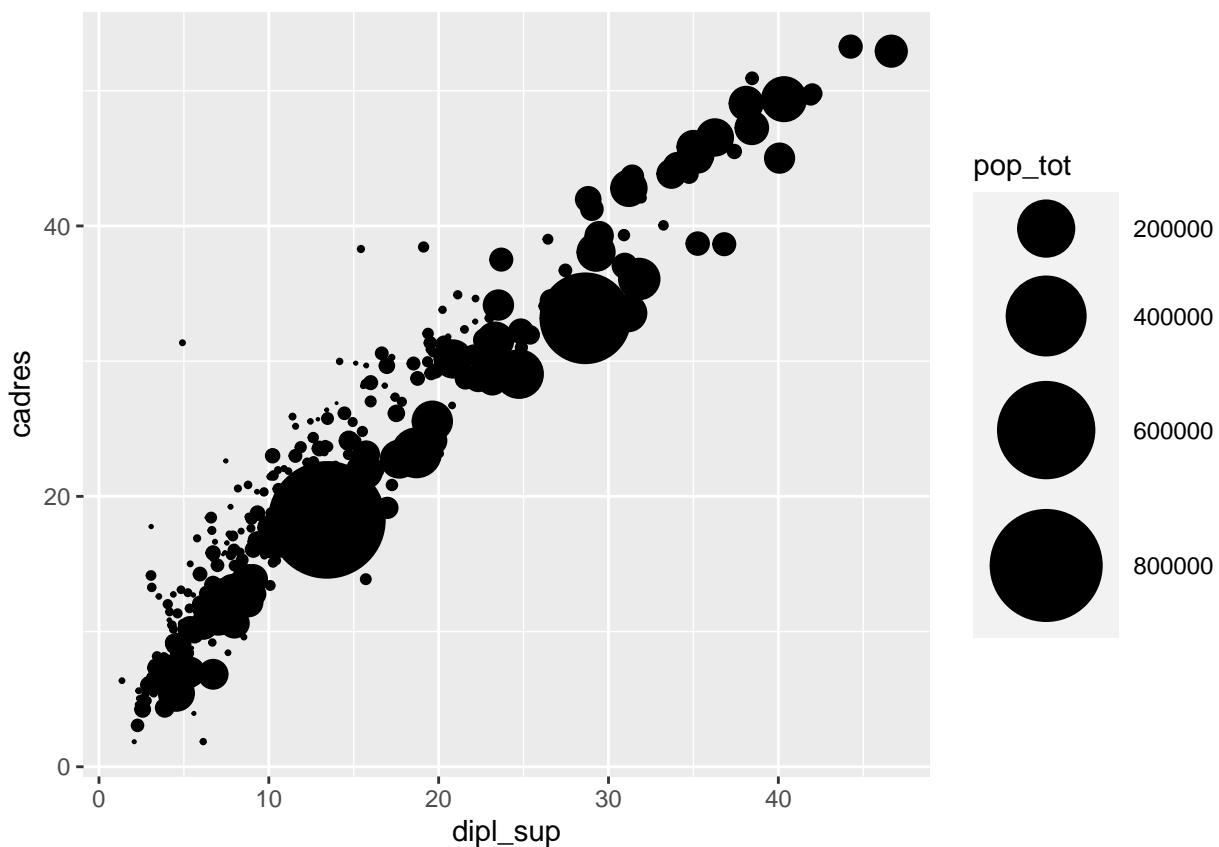
Pour modifier une *scale* existante, on ajoute un nouvel élément à notre objet *ggplot2* avec l’opérateur *+*. Cet élément prend la forme *scale_<attribut>_<type>*.

Voyons tout de suite quelques exemples.

8.7.1 *scale_size*

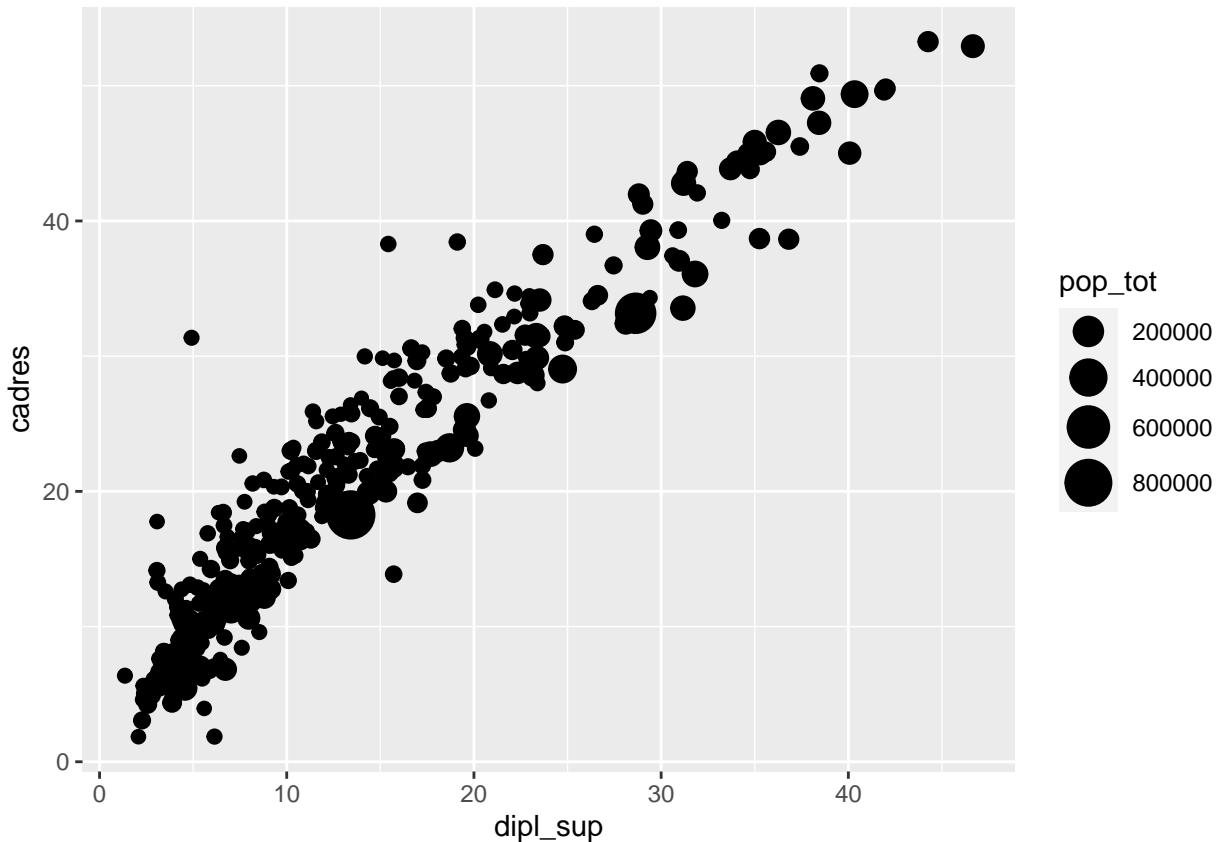
Si on souhaite modifier les tailles minimales et maximales des objets quand on a effectué un mappage de type *size*, on peut utiliser la fonction *scale_size* et son argument *range*.

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size(range = c(0, 20))
```



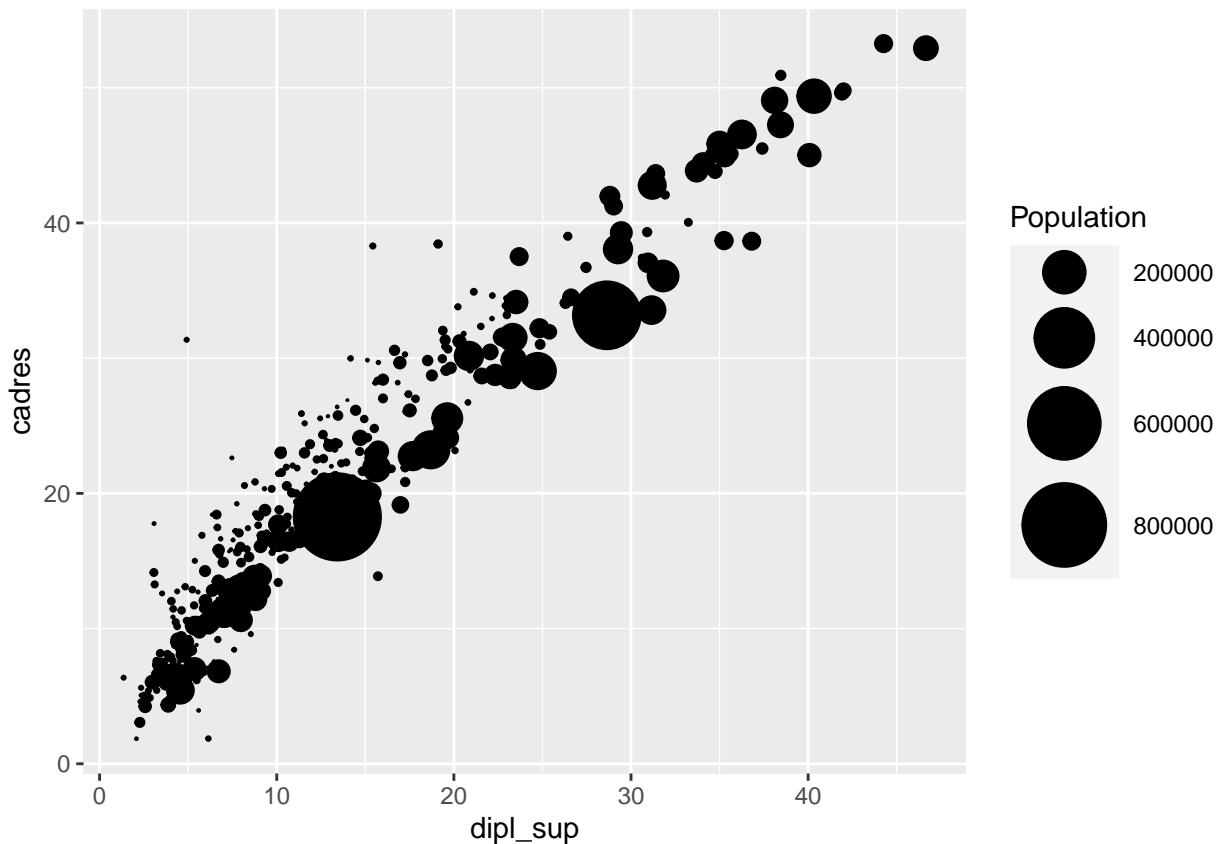
À comparer par exemple à :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +  
  scale_size(range = c(2, 8))
```



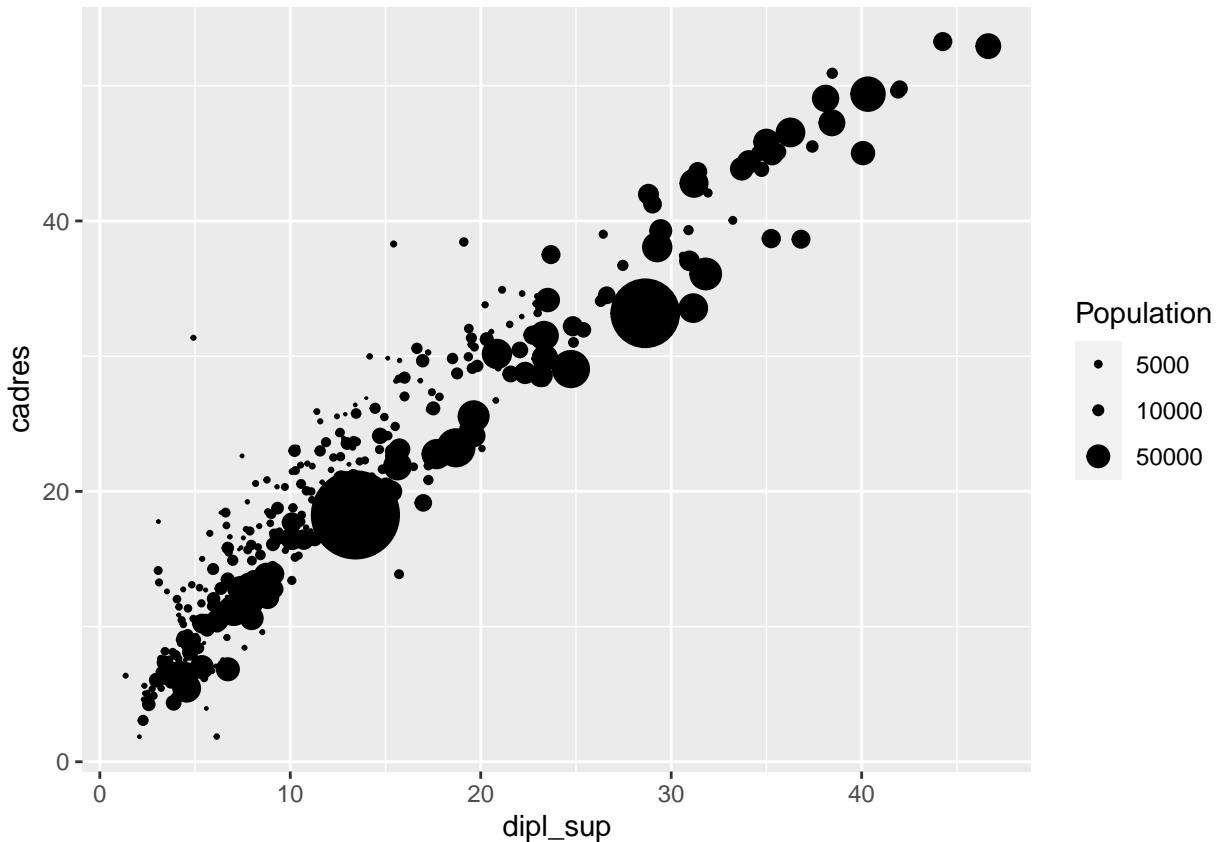
On peut ajouter d'autres paramètres à `scale_size`. Le premier argument est toujours le titre donné à la légende.

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size(
    "Population",
    range = c(0, 15)
  )
```



On peut aussi définir manuellement les éléments de légende représentés.

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, size = pop_tot)) +
  scale_size(
    "Population",
    range = c(0, 15),
    breaks = c(1000, 5000, 10000, 50000)
  )
```



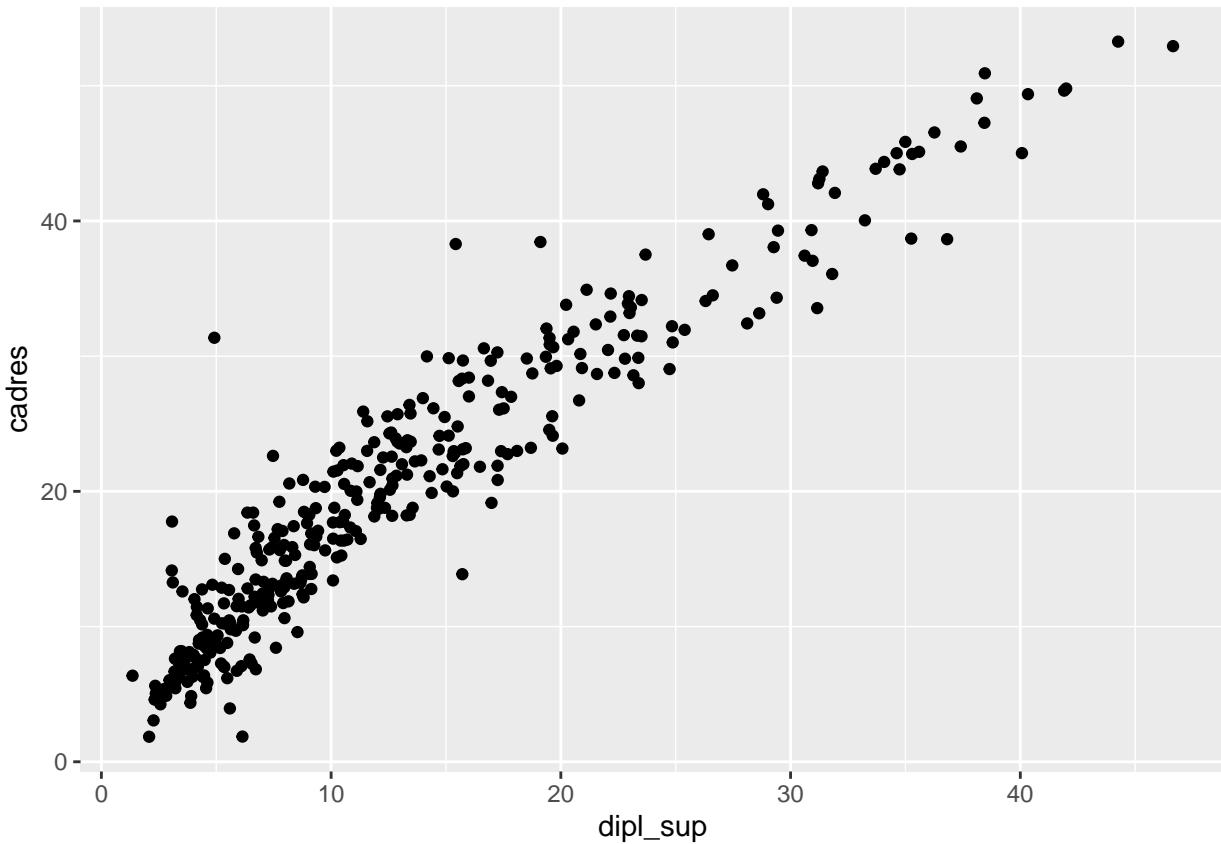
8.7.2 scale_x, scale_y

Les scales `scale_x_<type>` et `scale_y_<type>` modifient les axes x et y du graphique.

`scale_x_continuous` et `scale_y_continuous` s'appliquent lorsque la variable x ou y est numérique (quantitative).

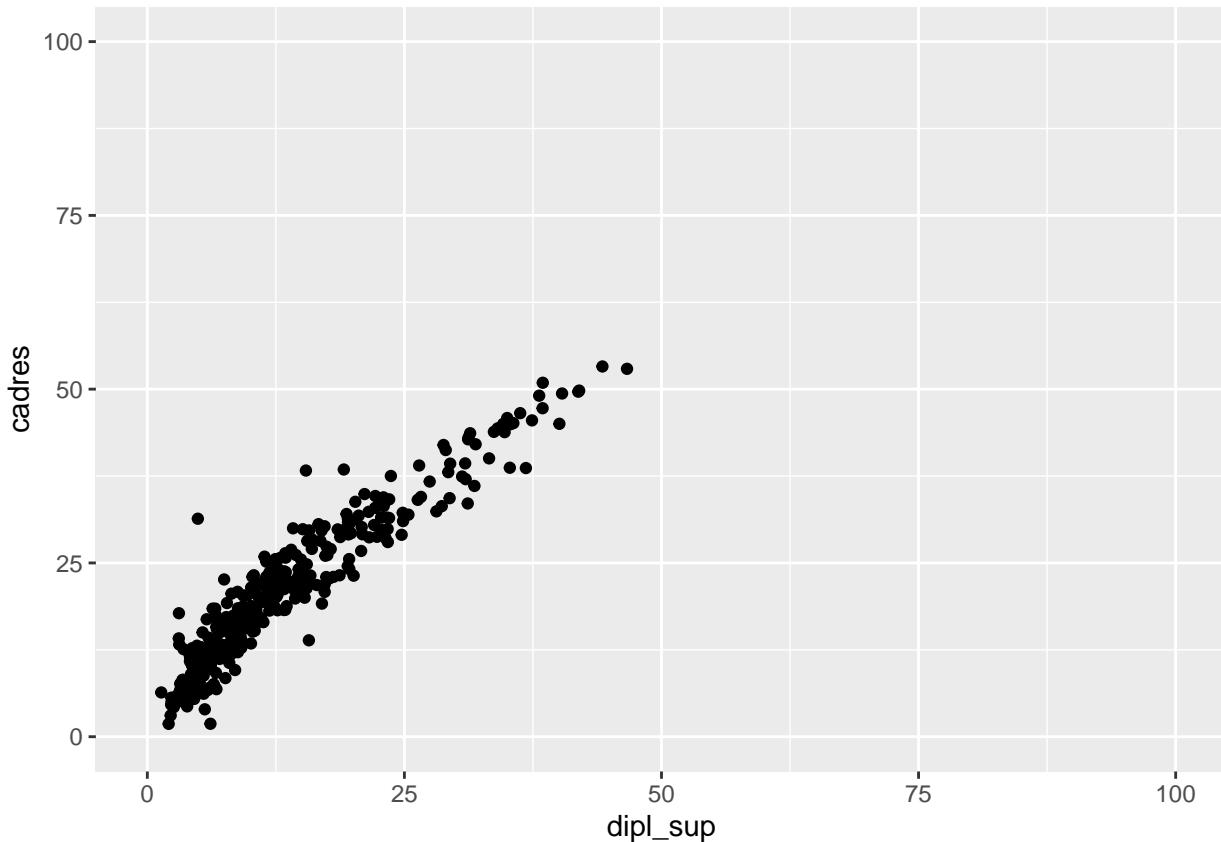
C'est le cas de notre nuage de points croisant part de cadres et part de diplômés du supérieur.

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres))
```



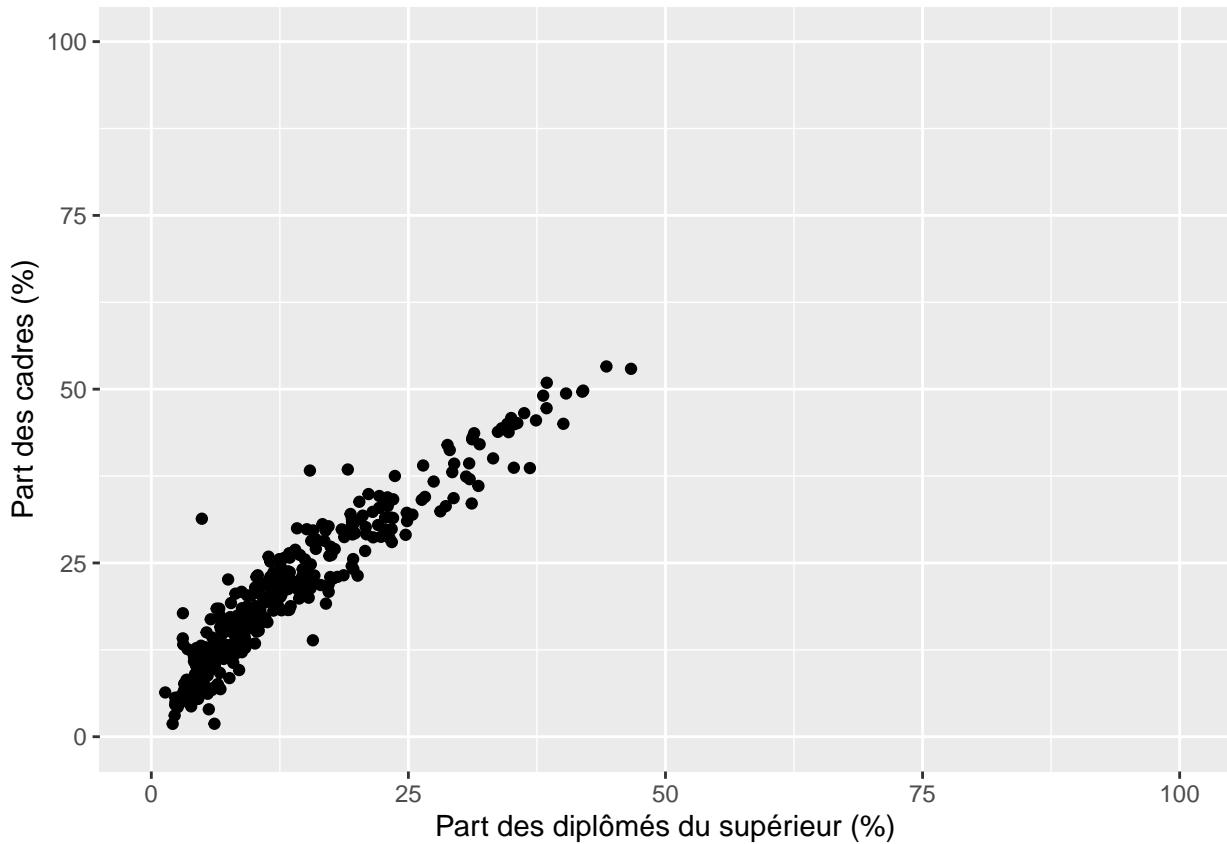
Comme on représente des pourcentages, on peut vouloir forcer les axes x et y à s'étendre des valeurs 0 à 100. On peut le faire en ajoutant un élément `scale_x_continuous` et un élément `scale_y_continuous`, et en utilisant leur argument `limits`.

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres)) +
  scale_x_continuous(limits = c(0,100)) +
  scale_y_continuous(limits = c(0,100))
```



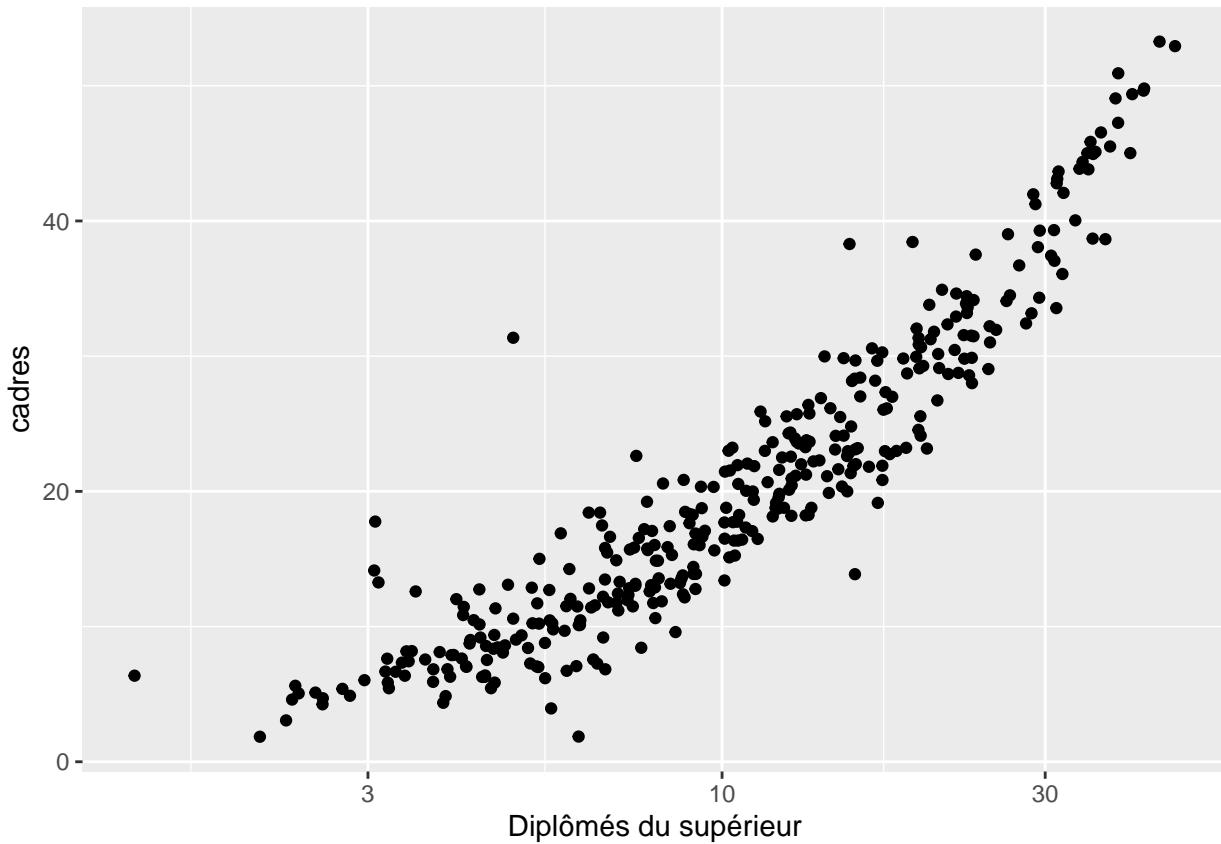
Là aussi, on peut modifier les étiquettes des axes en indiquant une chaîne de caractères en premier argument.

```
ggplot(rp) +
  geom_point(aes(x = diplo_sup, y = cadres)) +
  scale_x_continuous("Part des diplômés du supérieur (%)", limits = c(0,100)) +
  scale_y_continuous("Part des cadres (%)", limits = c(0,100))
```



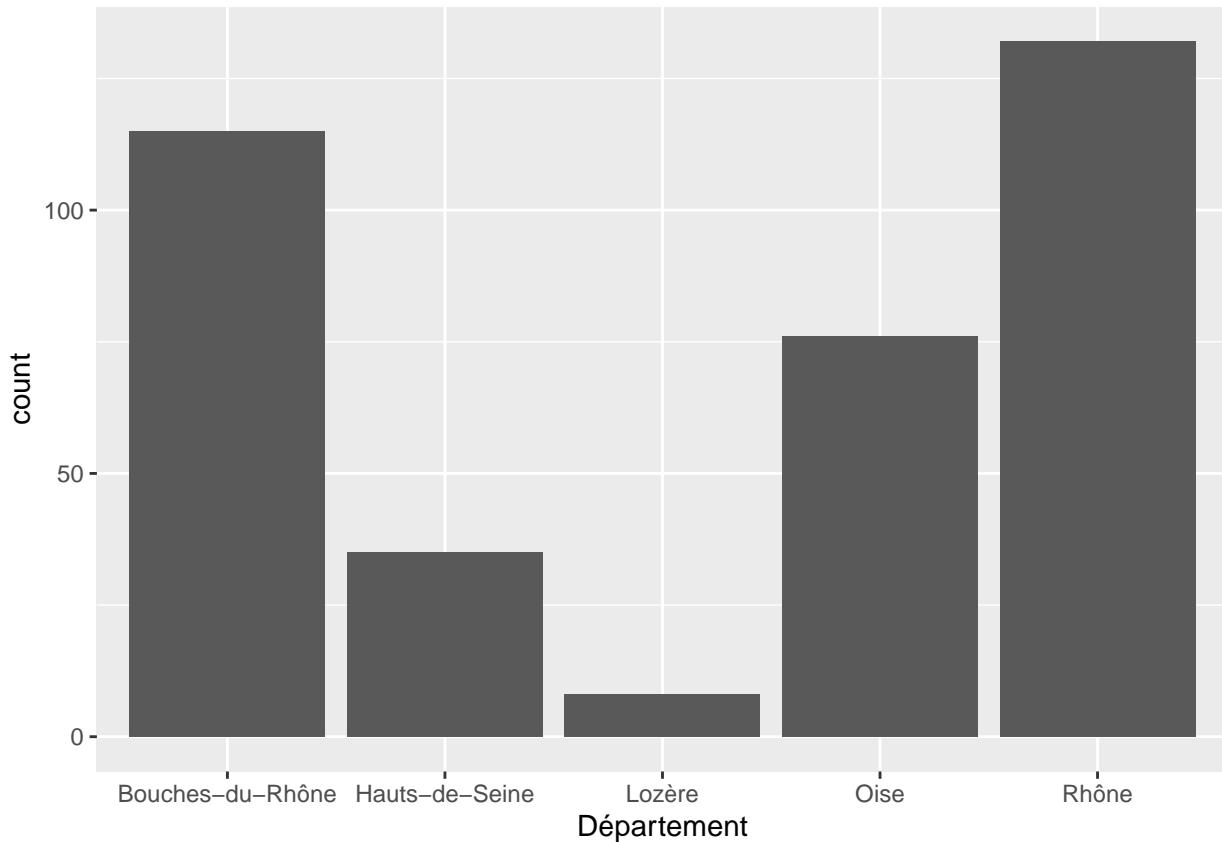
On peut utiliser `scale_x_log10` et `scale_y_log10` pour passer un axe à une échelle logarithmique.

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres)) +
  scale_x_log10("Diplômés du supérieur")
```



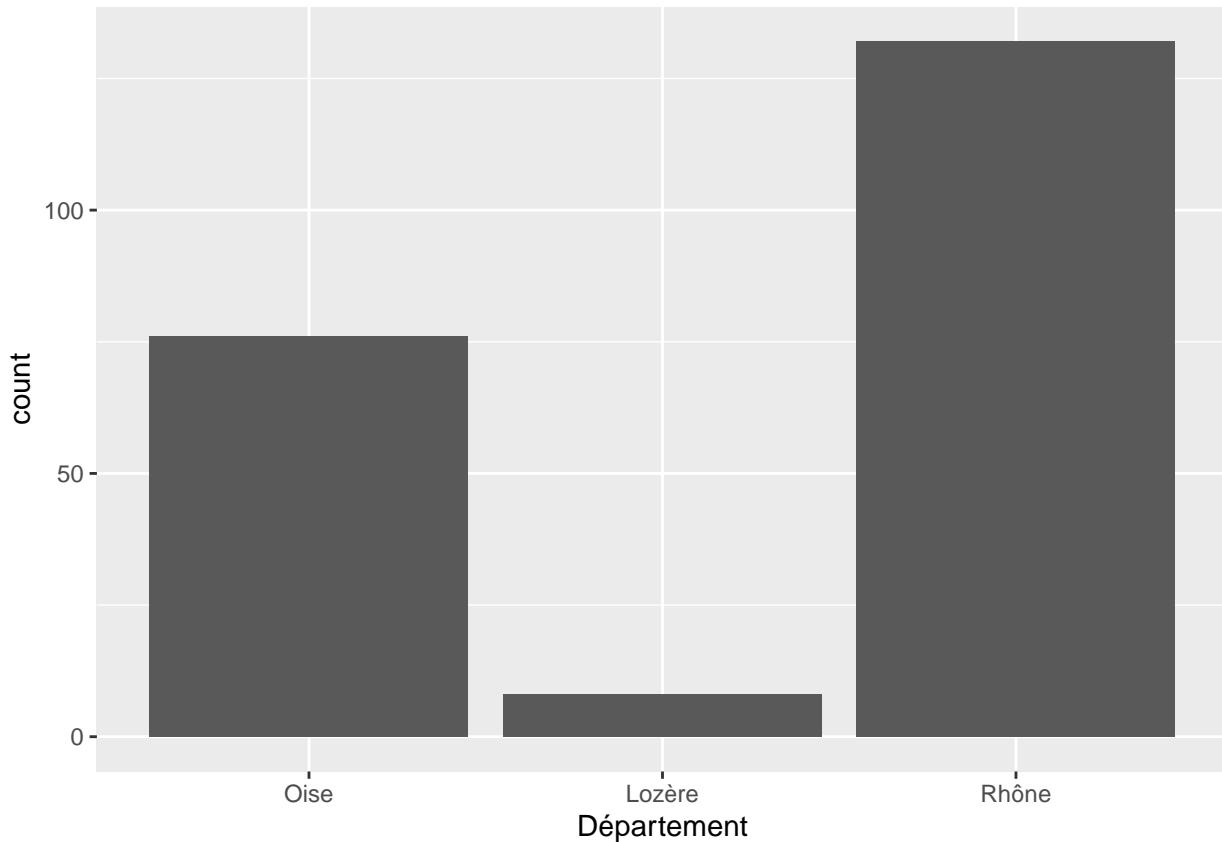
`scale_x_discrete` et `scale_y_discrete` s'appliquent lorsque l'axe correspond à une variable discrète (qualitative). C'est le cas par exemple de l'axe des x dans un diagramme en barres.

```
ggplot(rp) +
  geom_bar(aes(x = département)) +
  scale_x_discrete("Département")
```



L'argument `limits` de `scale_x_discrete` permet d'indiquer quelles valeurs sont affichées et dans quel ordre.

```
ggplot(rp) +
  geom_bar(aes(x = departement)) +
  scale_x_discrete("Département", limits = c("Oise", "Lozère", "Rhône"))
#> Warning: Removed 150 rows containing non-finite values (stat_count).
```



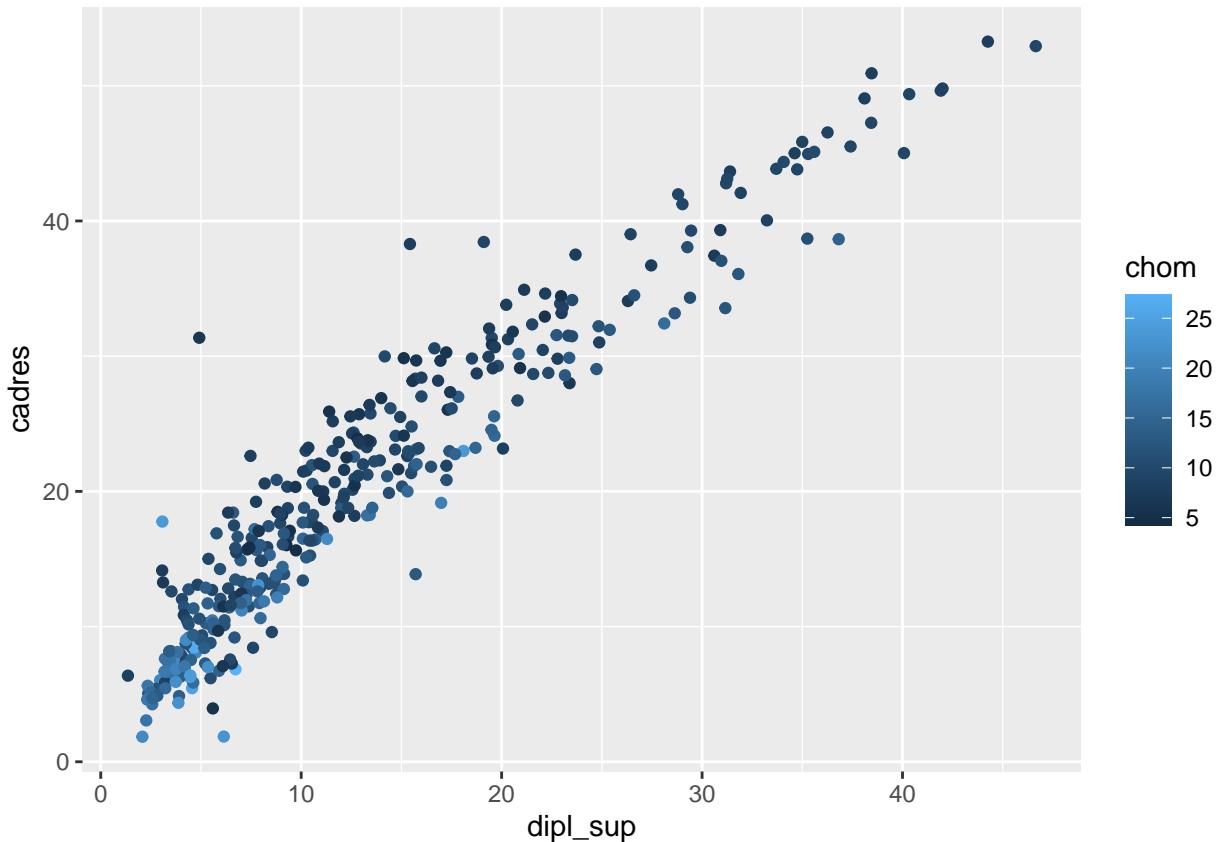
8.7.3 `scale_color`, `scale_fill`

Ces *scales* permettent, entre autre, de modifier les palettes de couleur utilisées pour le dessin (`color`) ou le remplissage (`fill`) des éléments graphiques. Dans ce qui suit, pour chaque fonction `scale_color` présentée il existe une fonction `scale_fill` équivalente et avec en général les mêmes arguments.

8.7.3.1 Variables quantitatives

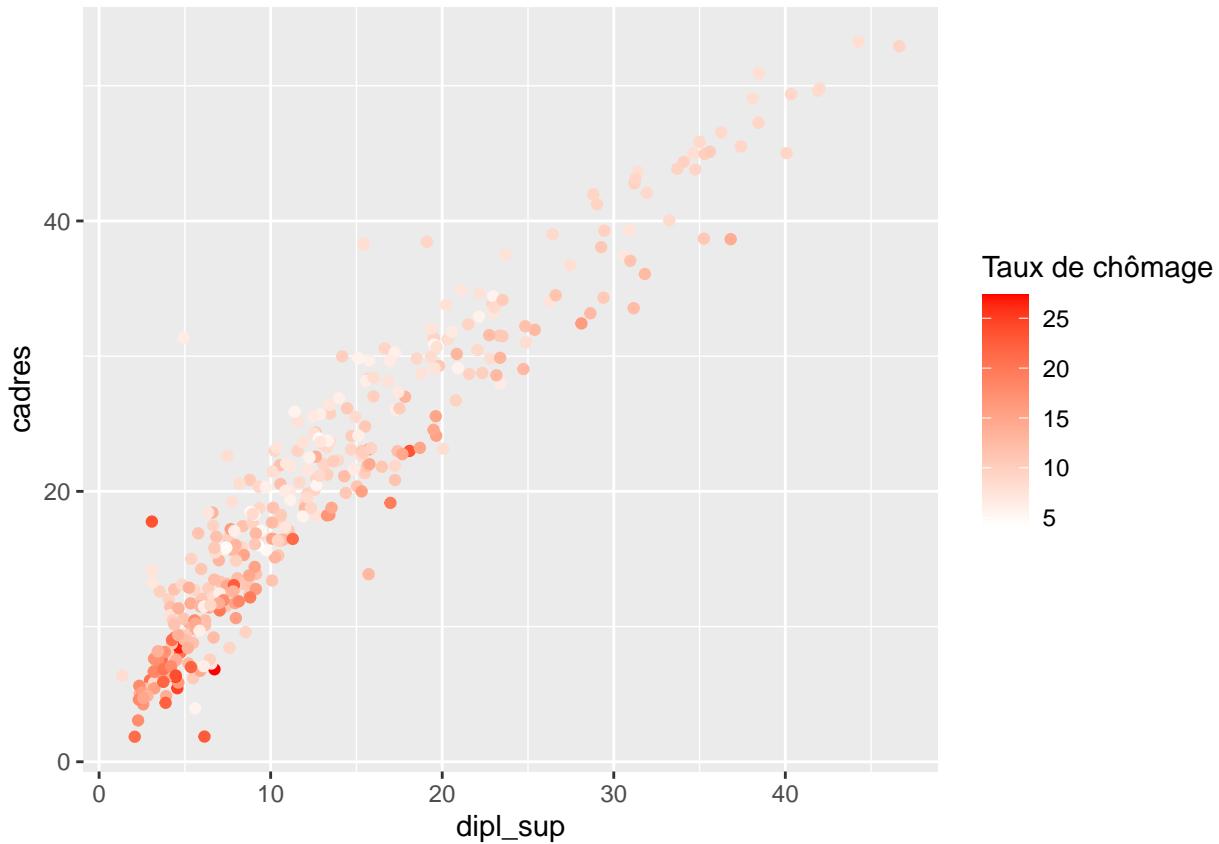
Le graphique suivant colore les points selon la valeur d'une variable numérique quantitative (ici la part de chômeurs) :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom))
```



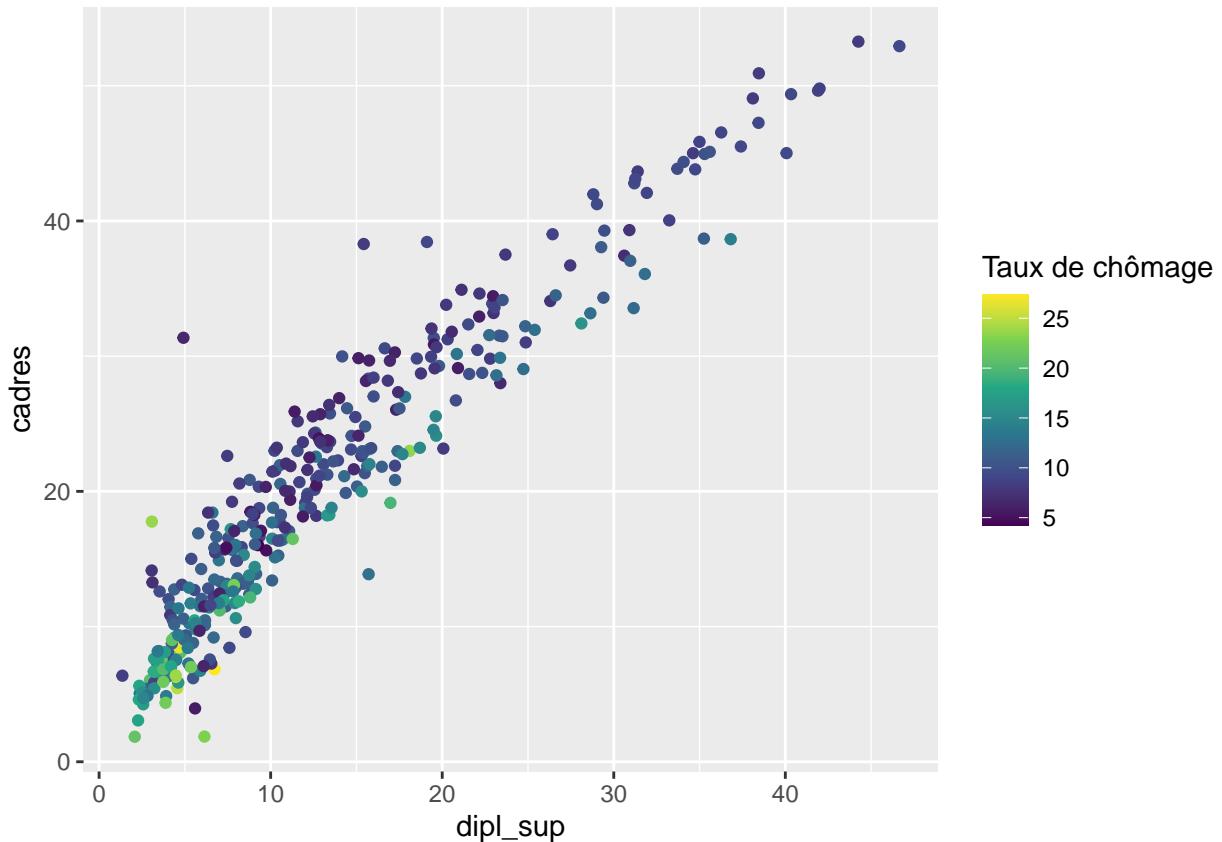
On peut modifier les couleurs utilisées avec les arguments `low` et `high` de la fonction `scale_color_gradient`. Ici on souhaite que la valeur la plus faible soit blanche, et la plus élevée rouge :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +  
  scale_color_gradient("Taux de chômage", low = "white", high = "red")
```



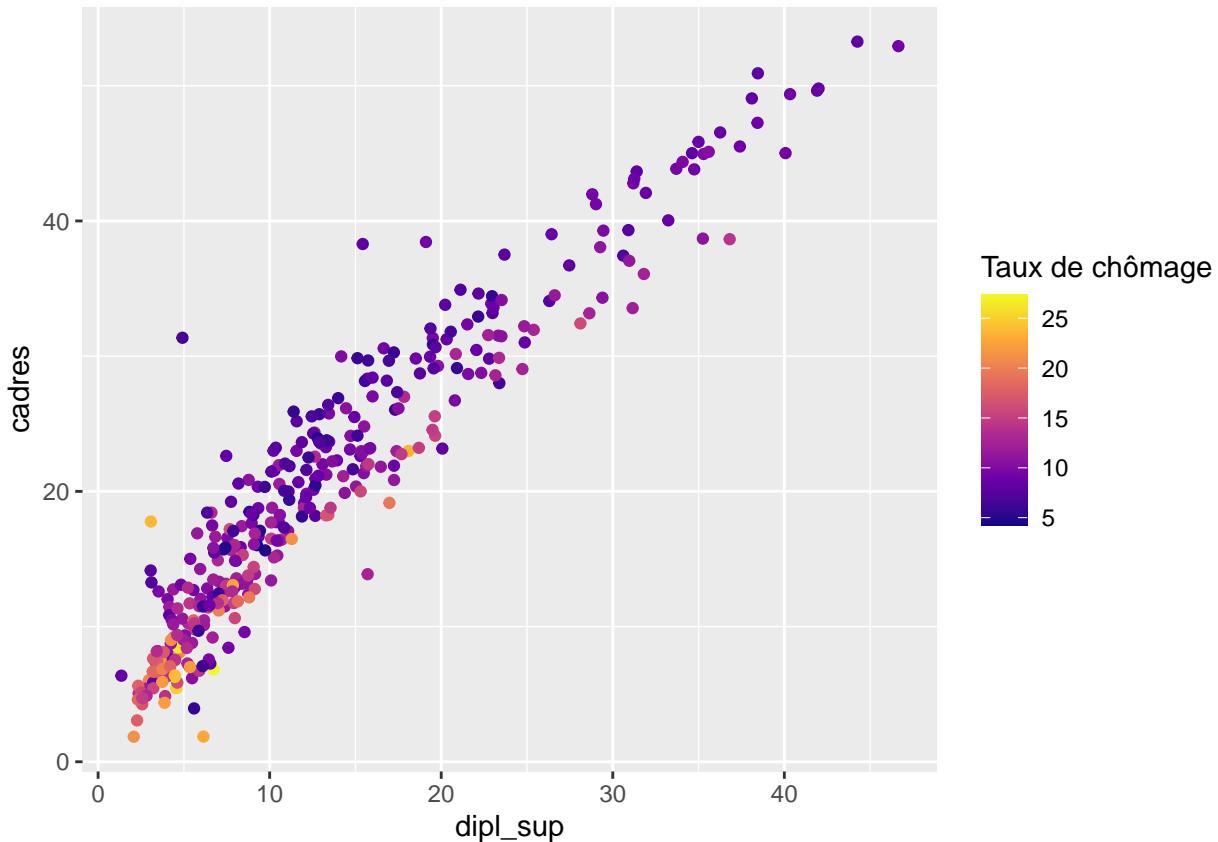
On peut aussi utiliser des palettes prédéfinies. L'une des plus populaires est la palette *viridis*, accessible en utilisant `scale_color_viridis_c` :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +  
  scale_color_viridis_c("Taux de chômage")
```



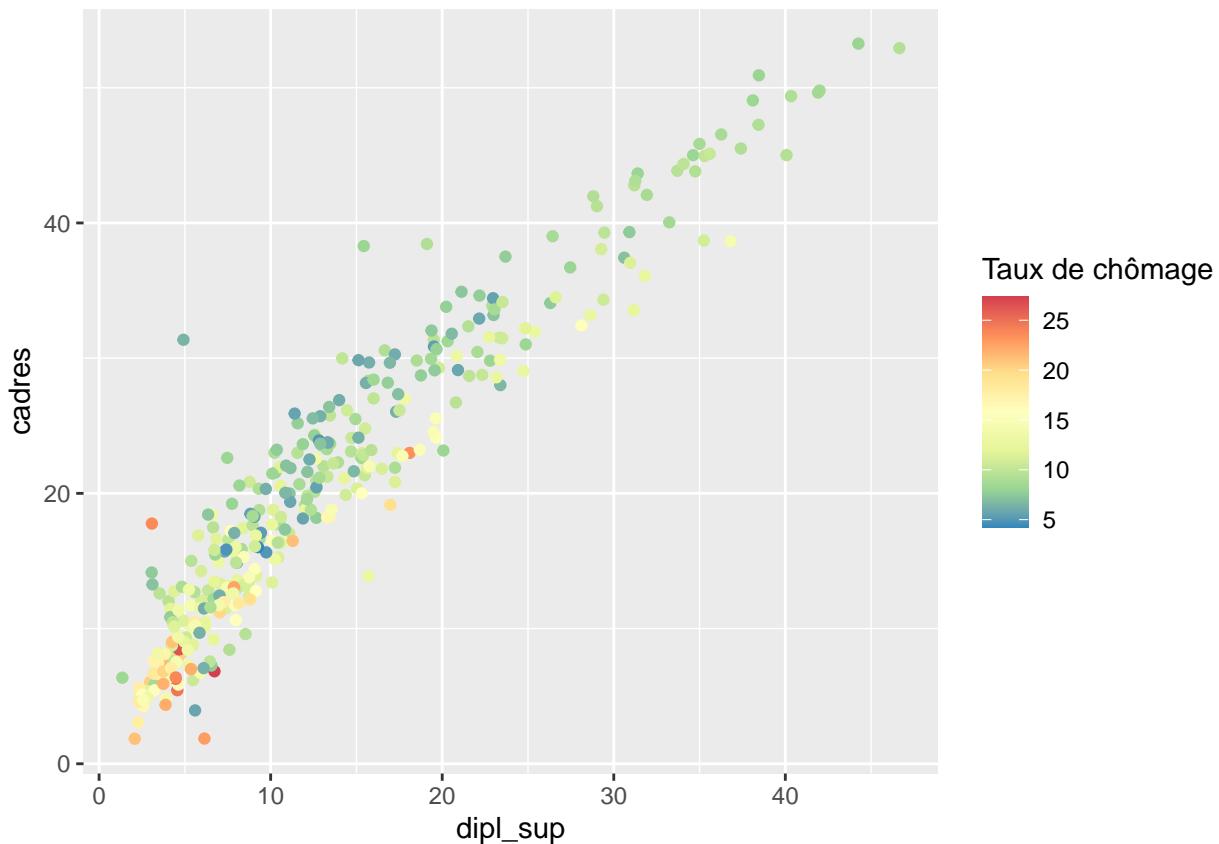
viridis propose également trois autres palettes, *magma*, *inferno* et *plasma*, accessibles via l'argument `option` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +
  scale_color_viridis_c("Taux de chômage", option = "plasma")
```



On peut aussi utiliser `scale_color_distiller`, qui transforme une des palettes pour variable qualitative de `scale_color_brewer` en palette continue pour variable numérique :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = chom)) +  
  scale_color_distiller("Taux de chômage", palette = "Spectral")
```

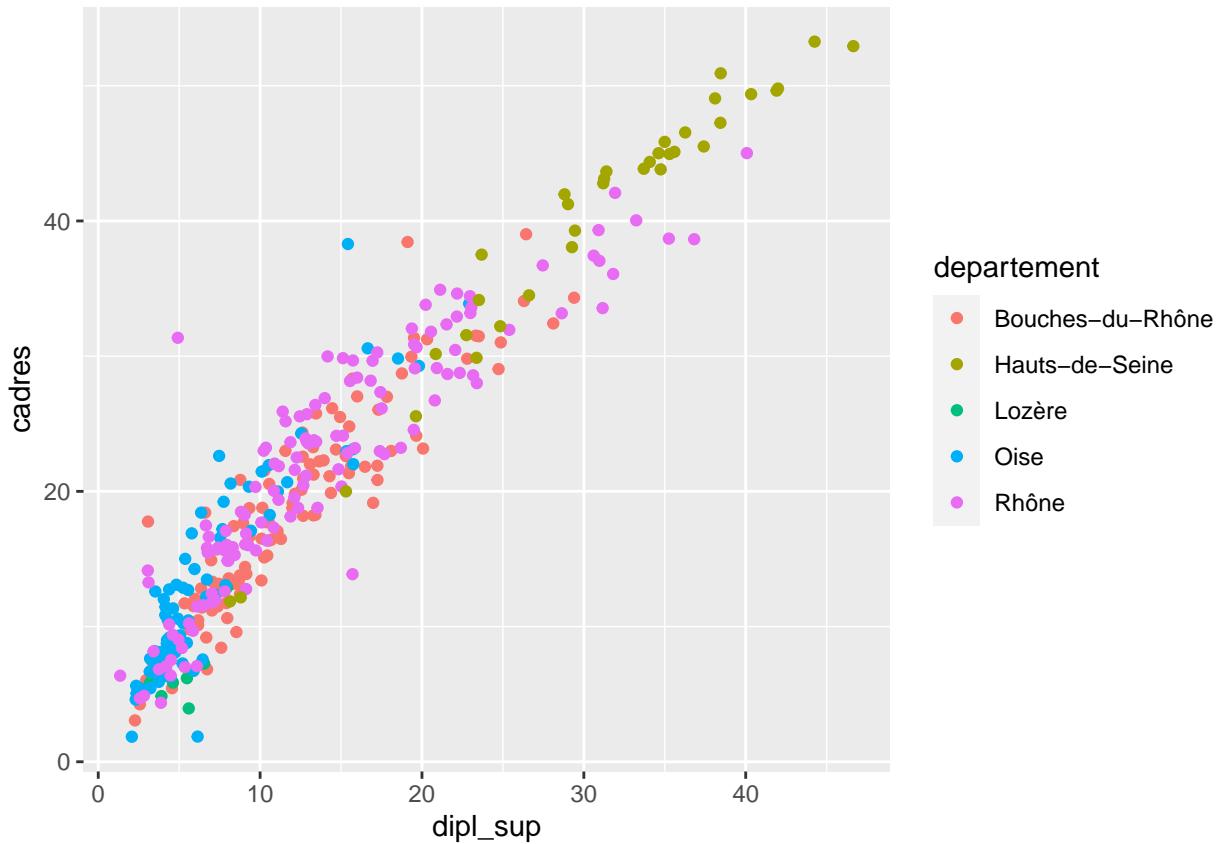


La liste des palettes de `scale_color_brewer` est indiquée en fin de section suivante.

8.7.3.2 Variables qualitatives

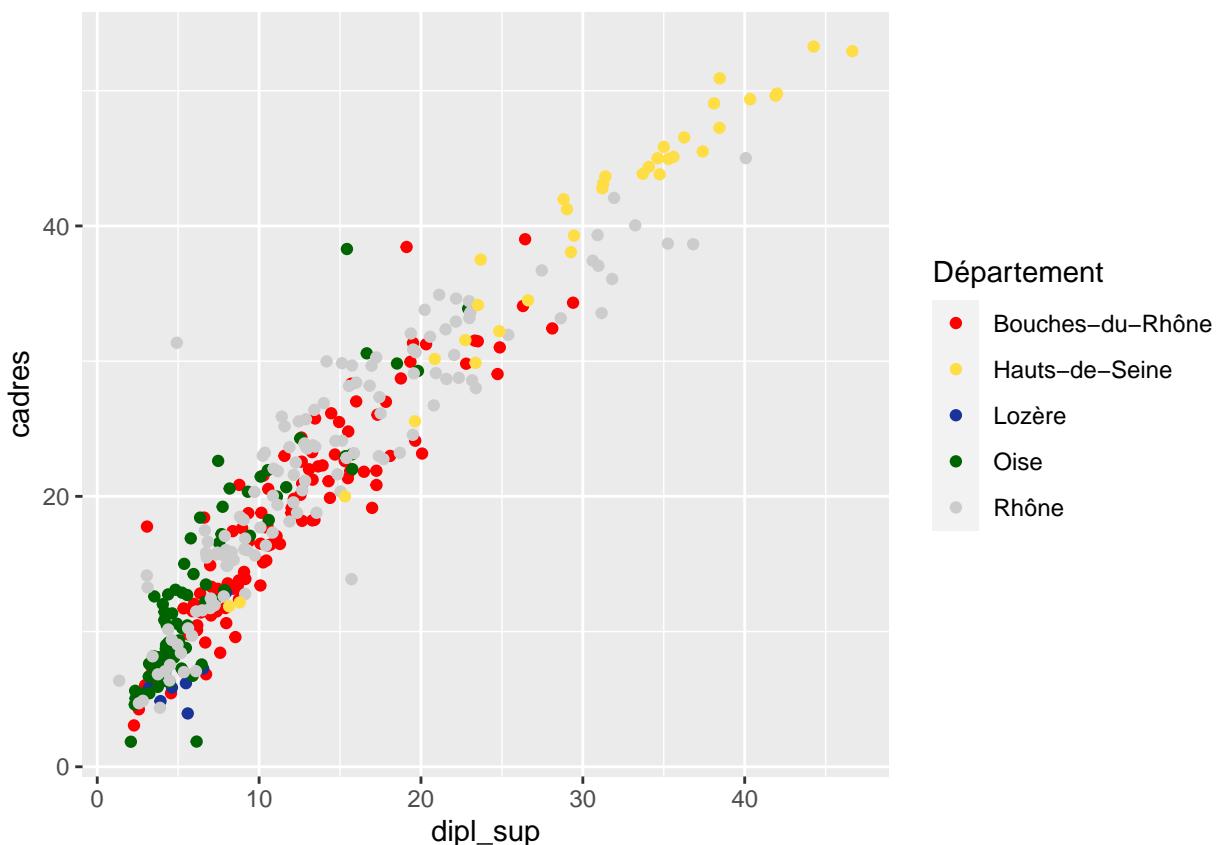
Si on a fait un mappage avec une variable discrète (qualitative), comme ici avec le département :

```
ggplot(rp) +  
  geom_point(aes(x = dipl_sup, y = cadres, color = département))
```



Une première possibilité est de modifier la palette manuellement avec `scale_color_manual` et son argument `values` :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = departement)) +
  scale_color_manual(
    "Département",
    values = c("red", "#FFDD45", "rgb(0.1,0.2,0.6)", "darkgreen", "grey80")
  )
```



L'exemple précédent montre plusieurs manières de définir manuellement des couleurs dans R :

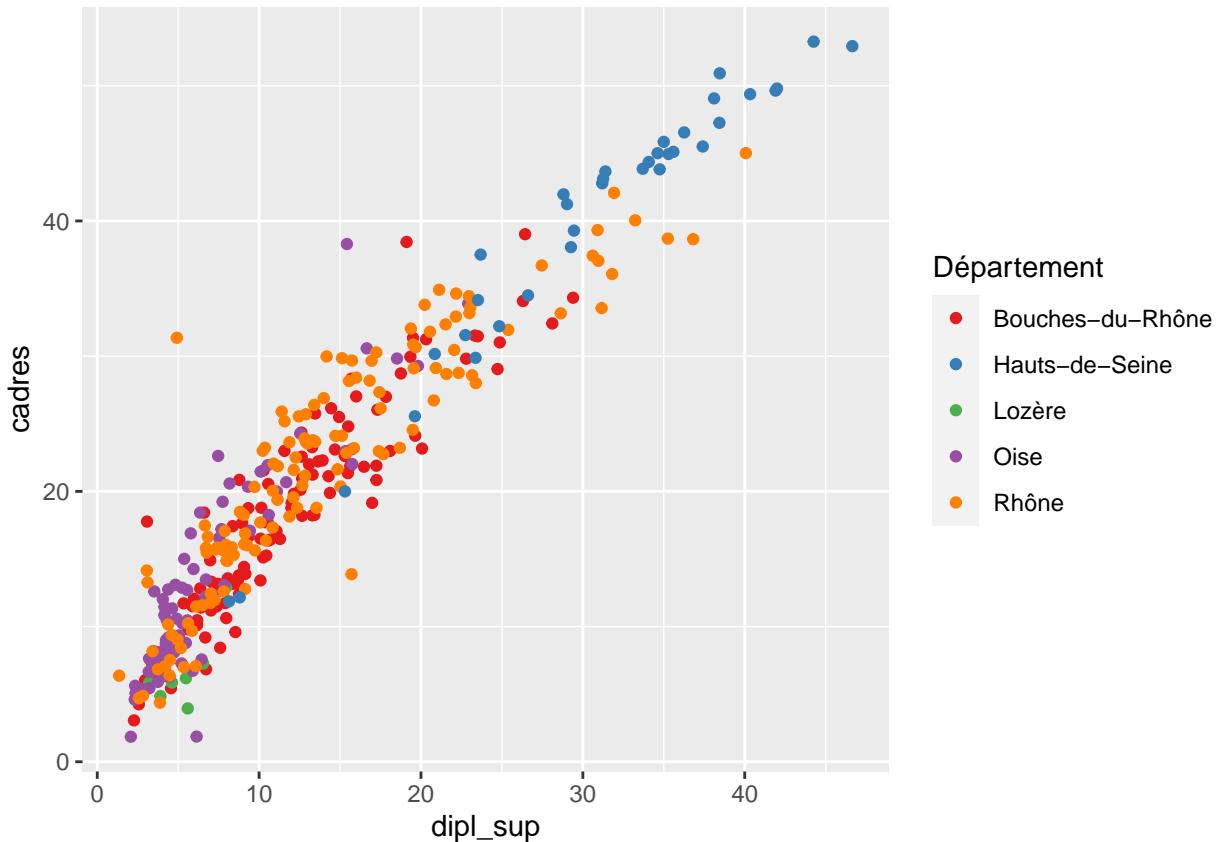
- Par code hexadécimal : "#FFDD45"
- En utilisant la fonction `rgb` et en spécifiant les composantes rouge, vert, bleu par des nombres entre 0 et 1 (et optionnellement une quatrième composante d'opacité, toujours entre 0 et 1) : `rgb(0.1,0.2,0.6)`
- En donnant un nom de couleur : "red", "darkgreen"

La liste complète des noms de couleurs connus par R peut être obtenu avec la fonction `colors()`. Vous pouvez aussi retrouver en ligne [la liste des couleurs et leur nom](#) (PDF).

Il est cependant souvent plus pertinent d'utiliser des palettes prédéfinies. Celles du site [Colorbrewer](#), initialement prévues pour la cartographie, permettent une bonne lisibilité, et peuvent être adaptées pour certains types de daltonisme.

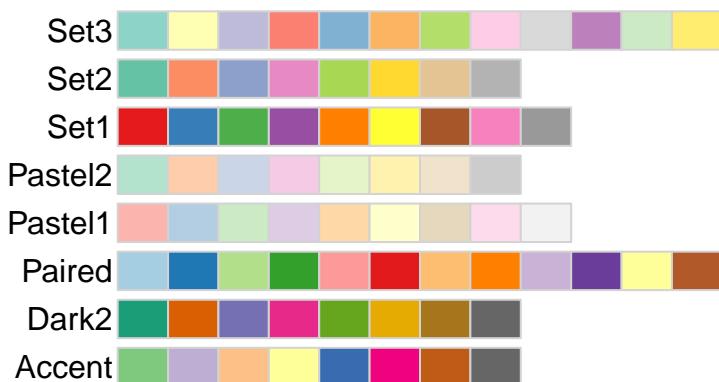
Ces palettes s'utilisent via la fonction `scale_color_brewer`, en passant le nom de la palette via l'argument `palette`. Par exemple, si on veut utiliser la palette Set1 :

```
ggplot(rp) +
  geom_point(aes(x = dipl_sup, y = cadres, color = departement)) +
  scale_color_brewer("Département", palette = "Set1")
```



Le graphique suivant, accessible via la fonction `display.brewer.all()`, montre la liste de toutes les palettes disponibles via `scale_color_brewer`. Elles sont réparties en trois familles : les palettes séquentielles (pour une variable quantitative), les palettes qualitatives, et les palettes divergentes (typiquement pour une variable quantitative avec une valeur de référence, souvent 0, et deux palettes continues distinctes pour les valeurs inférieures et pour les valeurs supérieures).

```
RColorBrewer::display.brewer.all()
```



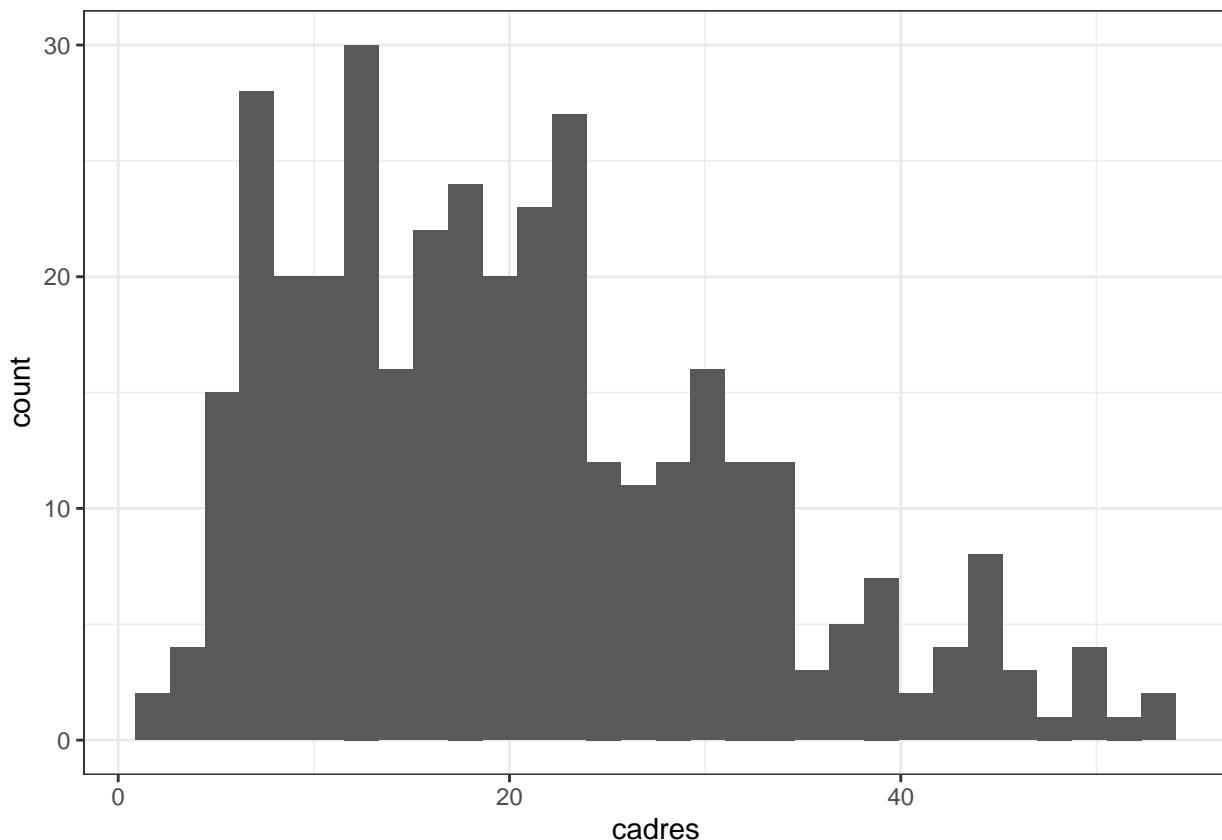
Il existe d'autres méthodes pour définir les couleurs : pour plus d'informations on pourra se reporter à [l'article de la documentation officielle sur ce sujet](#).

8.8 Thèmes

Les thèmes permettent de contrôler l'affichage de tous les éléments du graphique qui ne sont pas reliés aux données : titres, grilles, fonds, etc.

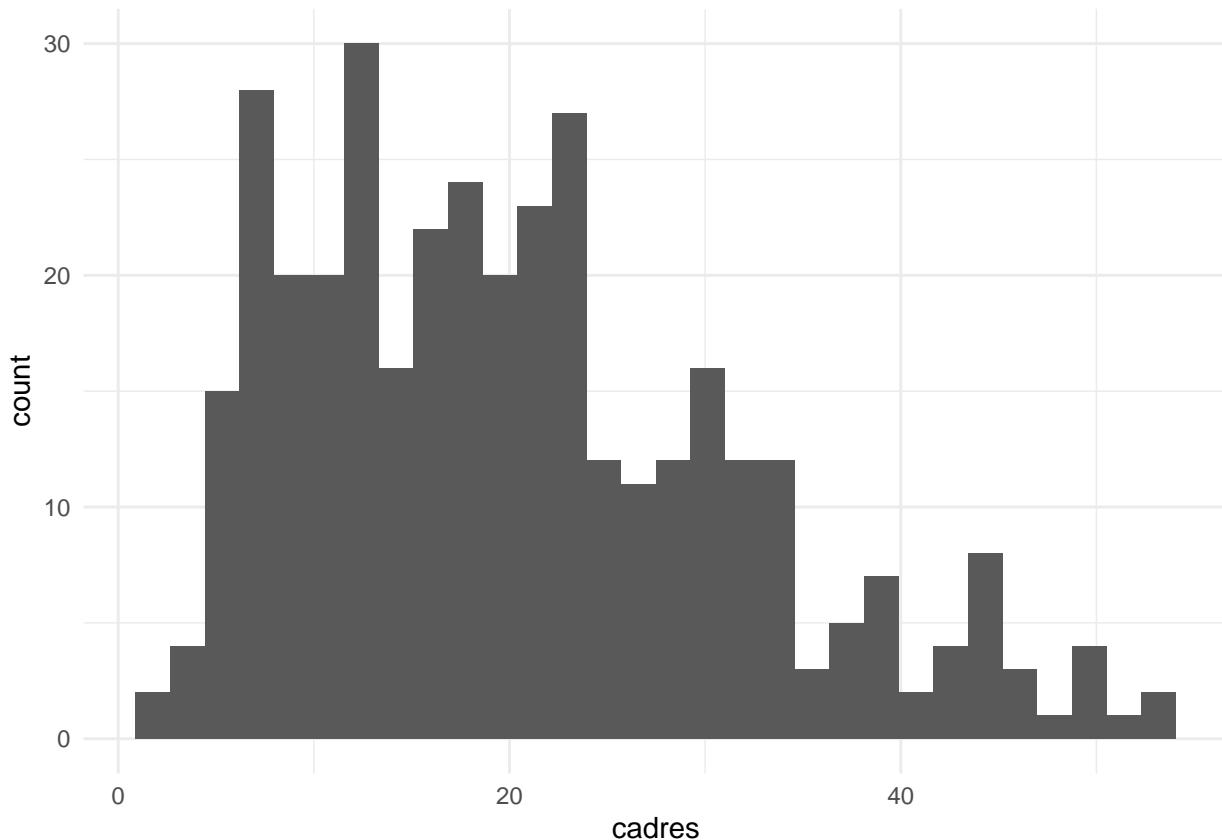
Il existe un certain nombre de thèmes préexistants, par exemple le thème `theme_bw` :

```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  theme_bw()
```



Ou le thème `theme_minimal` :

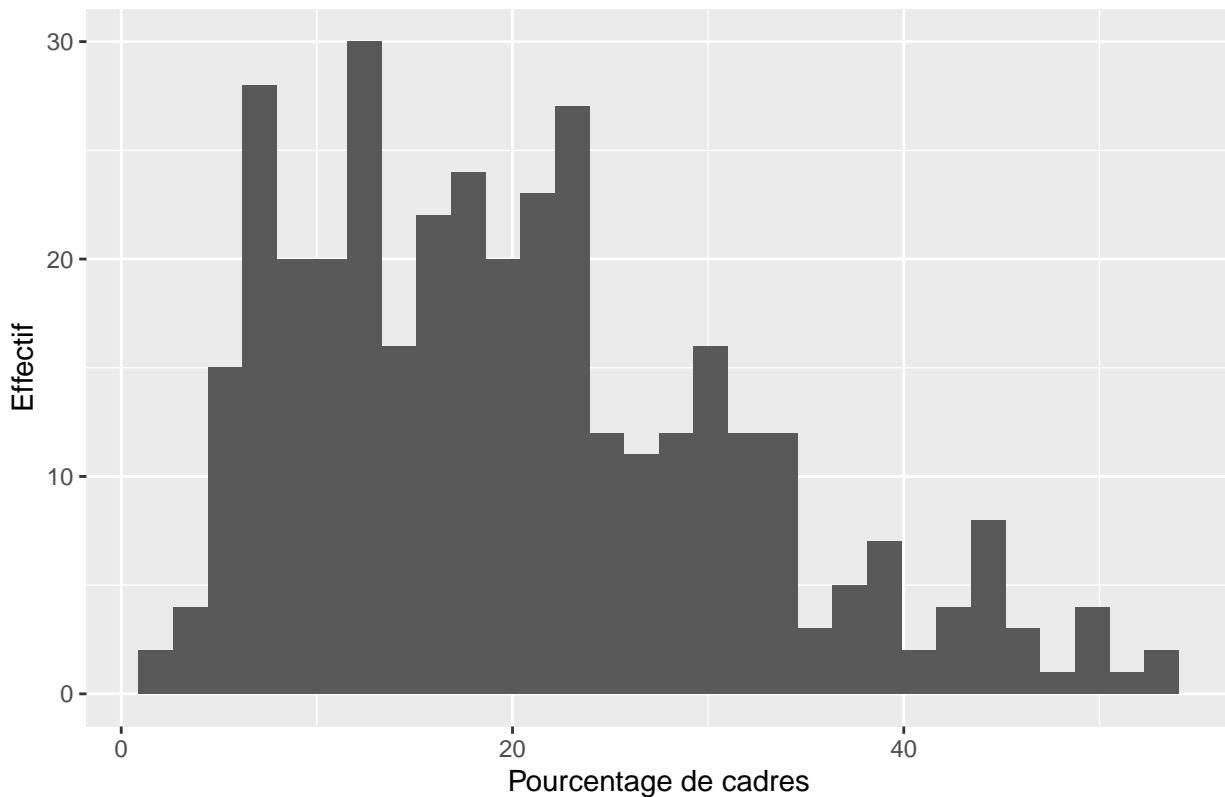
```
ggplot(data = rp) +
  geom_histogram(aes(x = cadres)) +
  theme_minimal()
```



On peut cependant modifier manuellement les différents éléments. Par exemple, les fonctions `ggtitle`, `xlab` et `ylab` permettent d'ajouter ou de modifier le titre du graphique, ainsi que les étiquettes des axes `x` et `y` :

```
ggplot(data = rp) +  
  geom_histogram(aes(x = cadres)) +  
  ggtitle("Un bien bel histogramme") +  
  xlab("Pourcentage de cadres") +  
  ylab("Effectif")
```

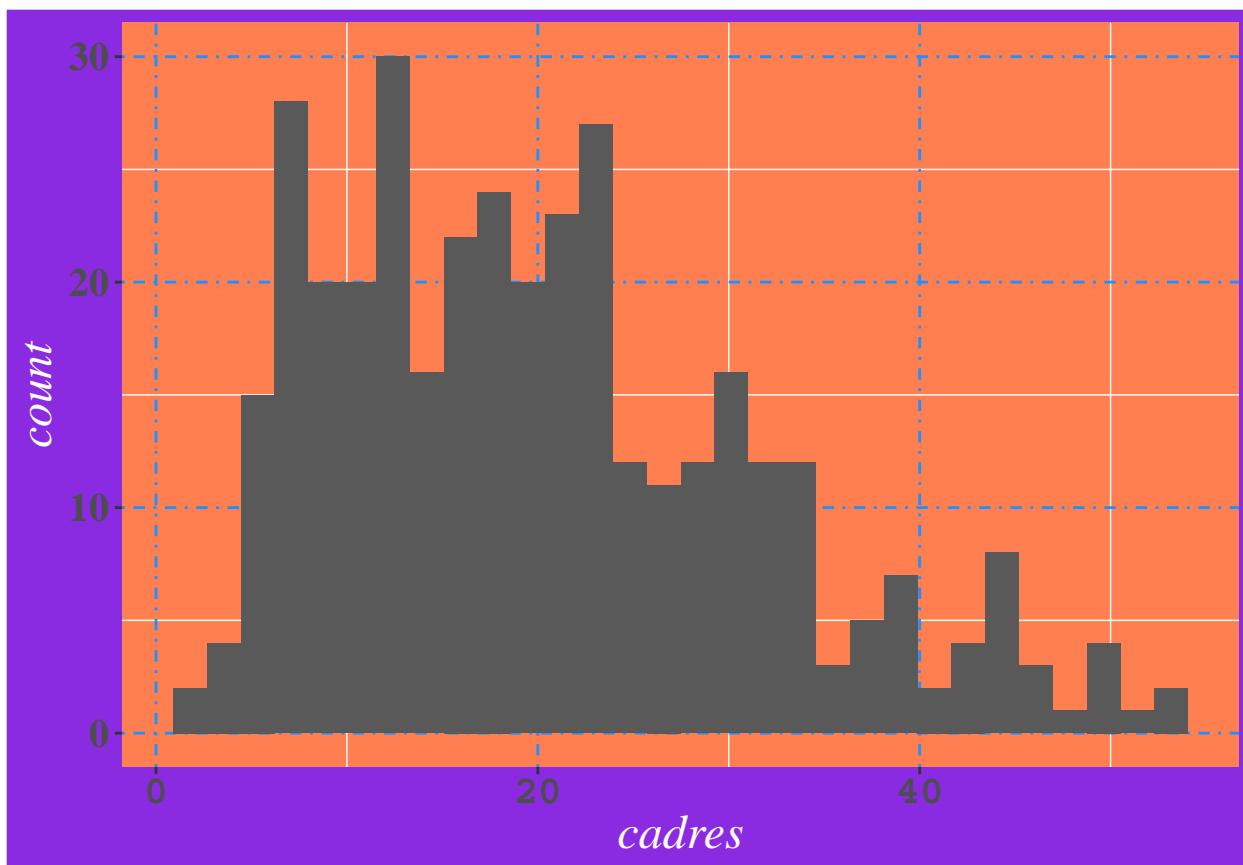
Un bien bel histogramme



Les éléments personnalisables étant nombreux, un bon moyen de se familiariser avec tous les arguments est sans doute l'addin RStudio *ggThemeAssist*. Pour l'utiliser il suffit d'installer le package du même nom, de sélectionner dans son script RStudio le code correspondant à un graphique *ggplot2*, puis d'aller dans le menu *Addins* et choisir *ggplot Theme Assistant*. Une interface graphique s'affiche alors permettant de modifier les différents éléments. Si on clique sur *Done*, le code sélectionné dans le script est alors automatiquement mis à jour pour correspondre aux modifications effectuées.

Ce qui permet d'obtenir très facilement des résultats extrêmement moches :

```
ggplot(data = rp) + geom_histogram(aes(x = cadres)) +
  theme(panel.grid.major = element_line(colour = "dodgerblue",
    size = 0.5, linetype = "dotdash"), axis.title = element_text(family = "serif",
    size = 18, face = "italic", colour = "white"),
  axis.text = element_text(family = "serif",
    size = 15, face = "bold"), axis.text.x = element_text(family = "mono"),
  plot.title = element_text(family = "serif"),
  legend.text = element_text(family = "serif"),
  legend.title = element_text(family = "serif"),
  panel.background = element_rect(fill = "coral"),
  plot.background = element_rect(fill = "blueviolet"))
```



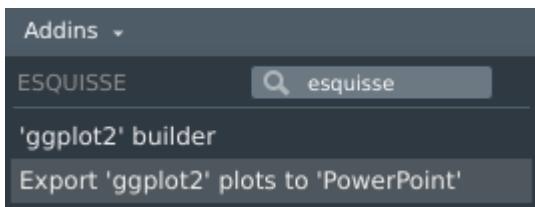
8.9 L'add-in esquisse

`esquisse` est un package développé notamment par Victor Perrier de `dreamRs` et qui fournit une interface graphique pour la construction de graphiques avec `ggplot2`.

Pour l'utiliser, il faut évidemment préalablement installer l'extension :

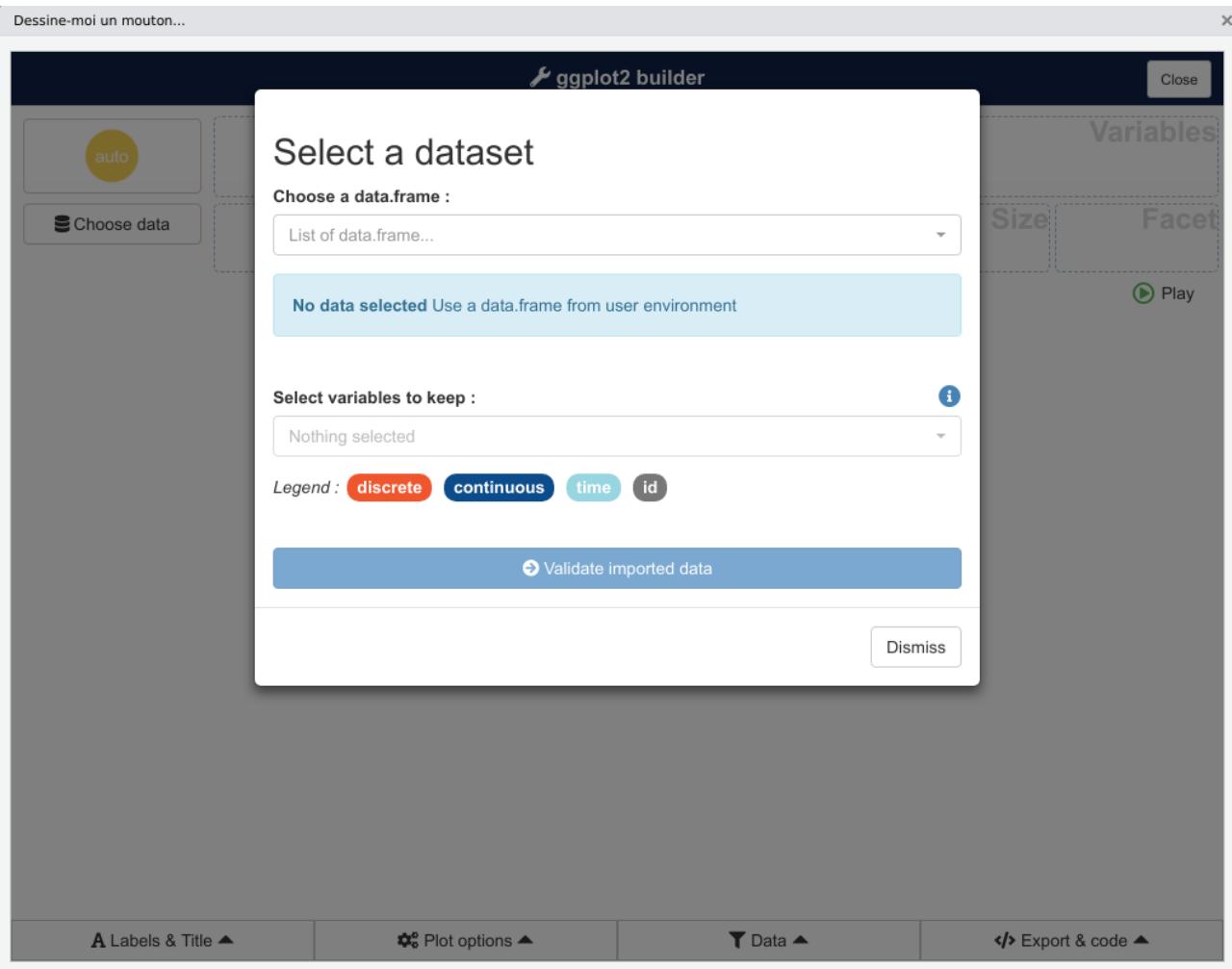
```
install.packages("esquisse")
```

Pour lancer l'interface, ouvrez le menu *Addins* dans la barre d'outils de RStudio, et cliquez sur '`ggplot2`' builder¹.



Une fenêtre s'ouvre : la première étape consiste à choisir un *data frame* de votre environnement, et éventuellement à ne sélectionner que certaines de ses variables.

¹Vous pouvez aussi lancer la commande `esquisser::esquisse()` dans la Console.

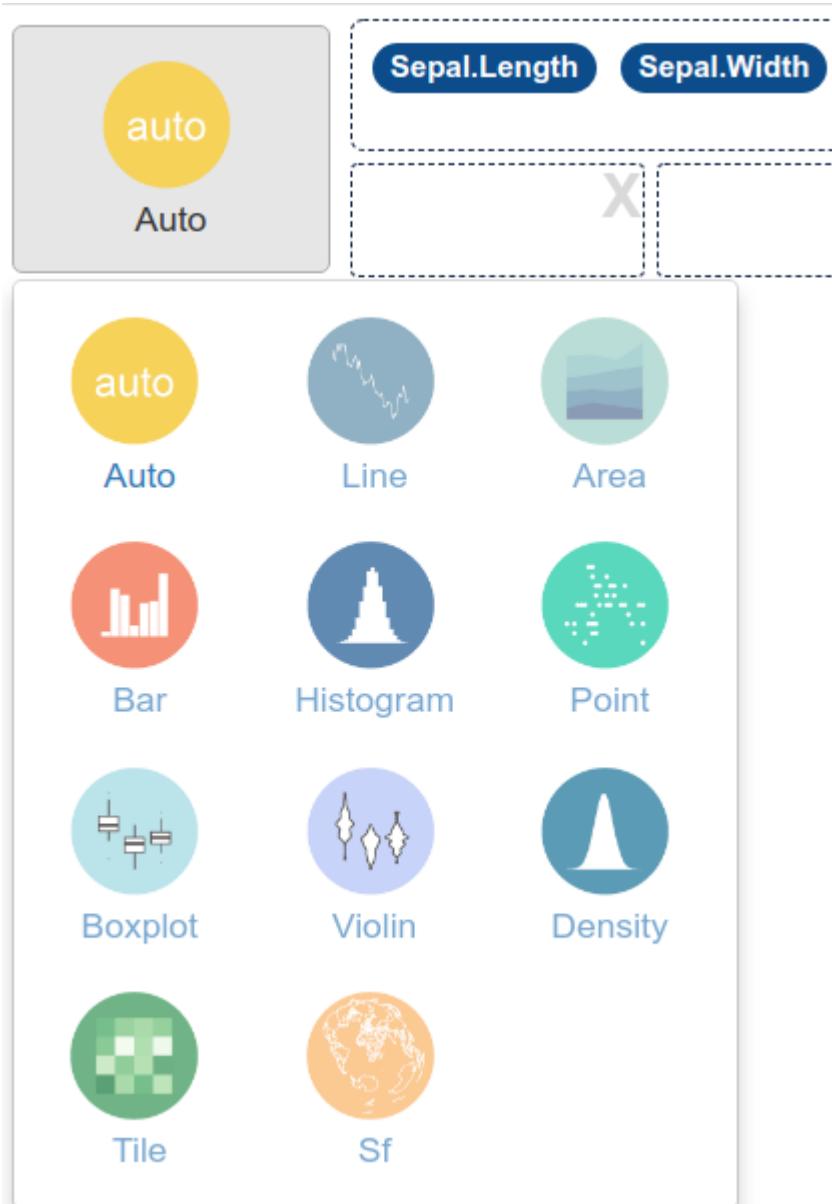


Une fois le choix effectué, cliquez sur *Validate imported data*.

L'interface principale s'affiche alors. La liste des variables du *data frame* apparaît en haut, et vous pouvez les faire glisser dans les zones *X*, *Y*, *Fill*, *Color*, *Size* et *Facet* pour créer des mappages. Le graphique se met automatiquement à jour.

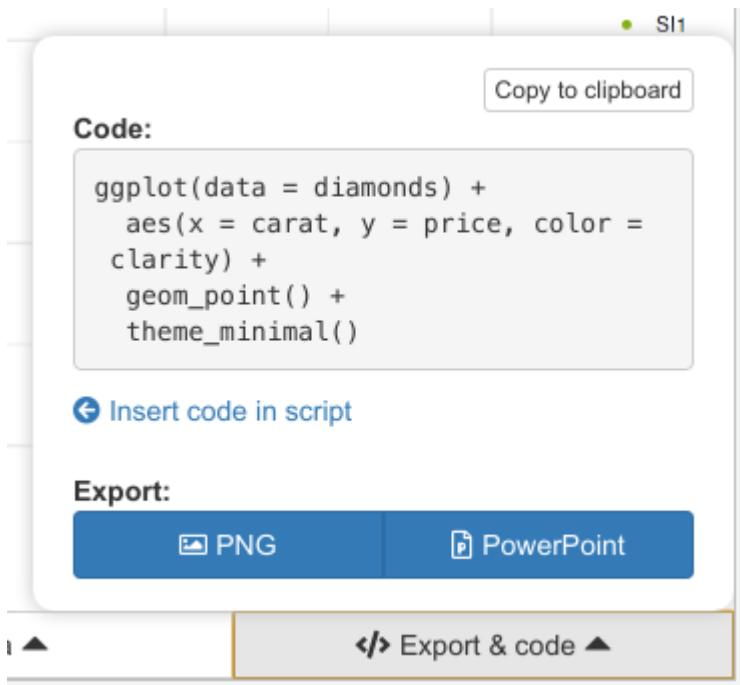


Par défaut, **esquisse** sélectionne le type de graphique le plus approprié selon la nature de vos variables. Mais vous pouvez choisir un autre type de graphique à l'aide de l'icône en haut à gauche, parmi onze disponibles (dont *Auto*):



Enfin, une série de menus en bas de l'interface vous permet de personnaliser les titres, les annotations (*labels*), la présentation ou de filtrer des valeurs de vos variables.

Quand vous avez généré un graphique que vous souhaitez conserver, ouvrez le menu *Export & code* :



Vous y trouverez le code R correspondant au graphique actuellement affiché. Vous pouvez dès lors le copier pour le coller dans votre script, ou cliquer sur *Insert code in script* pour l'insérer directement dans votre script à l'endroit où se trouve votre curseur.

esquisse ne propose pas (encore) tous les `geom` ou toutes les possibilités de `ggplot2`, mais ça peut être un outil très utile et pratique pour une exploration rapide de données ou lorsqu'on est un peu perdu dans la syntaxe et les fonctions de l'extension.

Pour plus d'informations, vous pouvez vous référer à la [page du projet sur GitHub](#) (en anglais).

8.10 Ressources

[La documentation officielle](#) (en anglais) de `ggplot2` est très complète et accessible en ligne.

Une “antisèche” (en anglais) résumant en deux pages l’ensemble des fonctions et arguments et disponible soit directement depuis RStudio (menu *Help > Cheatsheets > Data visualization with ggplot2*) ou [en ligne](#).

Les parties [Data visualisation](#) et [Graphics for communication](#) de l’ouvrage en ligne *R for data science*, de Hadley Wickham, sont une très bonne introduction à `ggplot2`.

Plusieurs ouvrages, toujours en anglais, abordent en détail l’utilisation de `ggplot2`, en particulier [ggplot2: Elegant Graphics for Data Analysis](#), toujours de Hadley Wickham, et le [R Graphics Cookbook](#) de Winston Chang.

Le [site associé](#) à ce dernier ouvrage comporte aussi pas mal d’exemples et d’informations intéressantes.

Enfin, si `ggplot2` présente déjà un très grand nombre de fonctionnalités, il existe aussi un système d’extensions permettant d’ajouter des `geom`, des thèmes, etc. Le site [ggplot2 extensions](#) est une très bonne ressource pour les parcourir et les découvrir, notamment grâce à sa [galerie](#).

8.11 Exercices

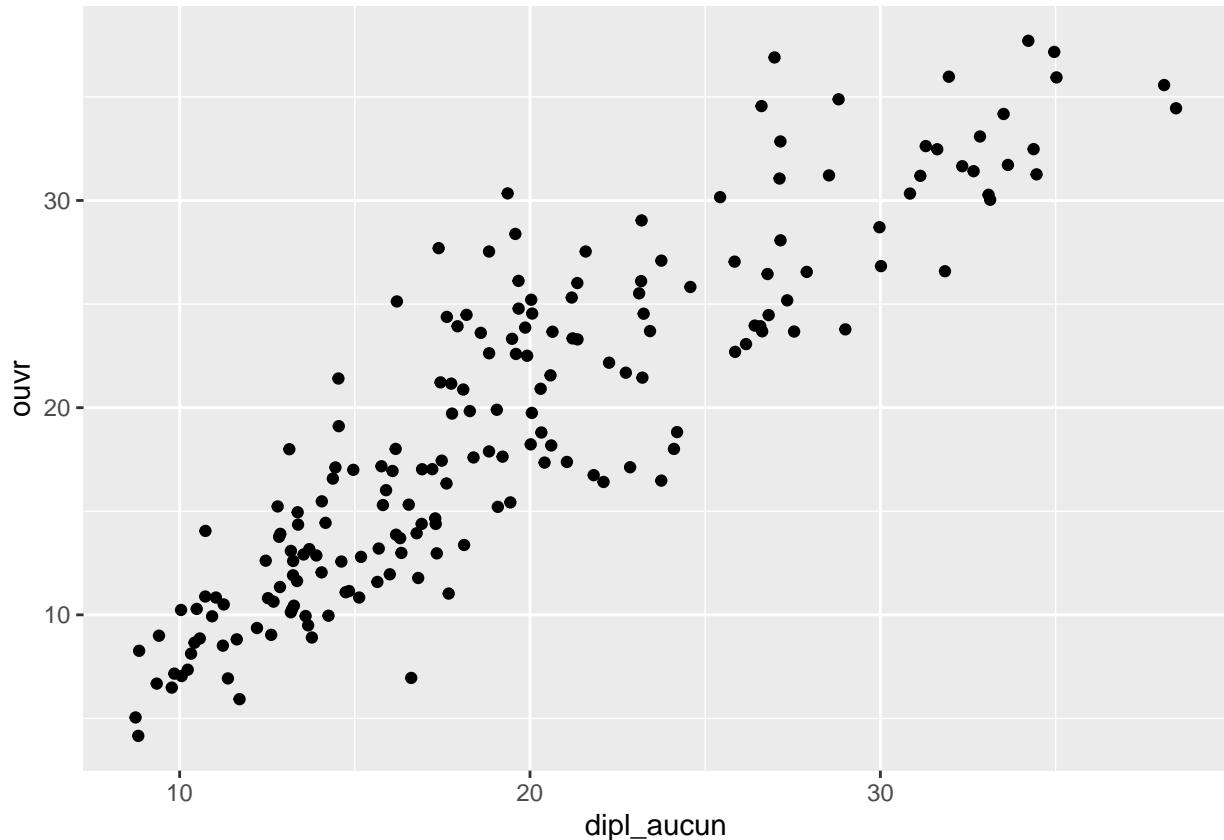
Pour les exercices qui suivent, on commence par charger les extensions nécessaires et les données du jeu de données `rp2018`. On crée alors un objet `rp69` comprenant uniquement les communes du Rhône et de la Loire.

```
library(tidyverse)
library(questionr)
data(rp2018)

rp69 <- filter(rp2018, departement %in% c("Rhône", "Loire"))
```

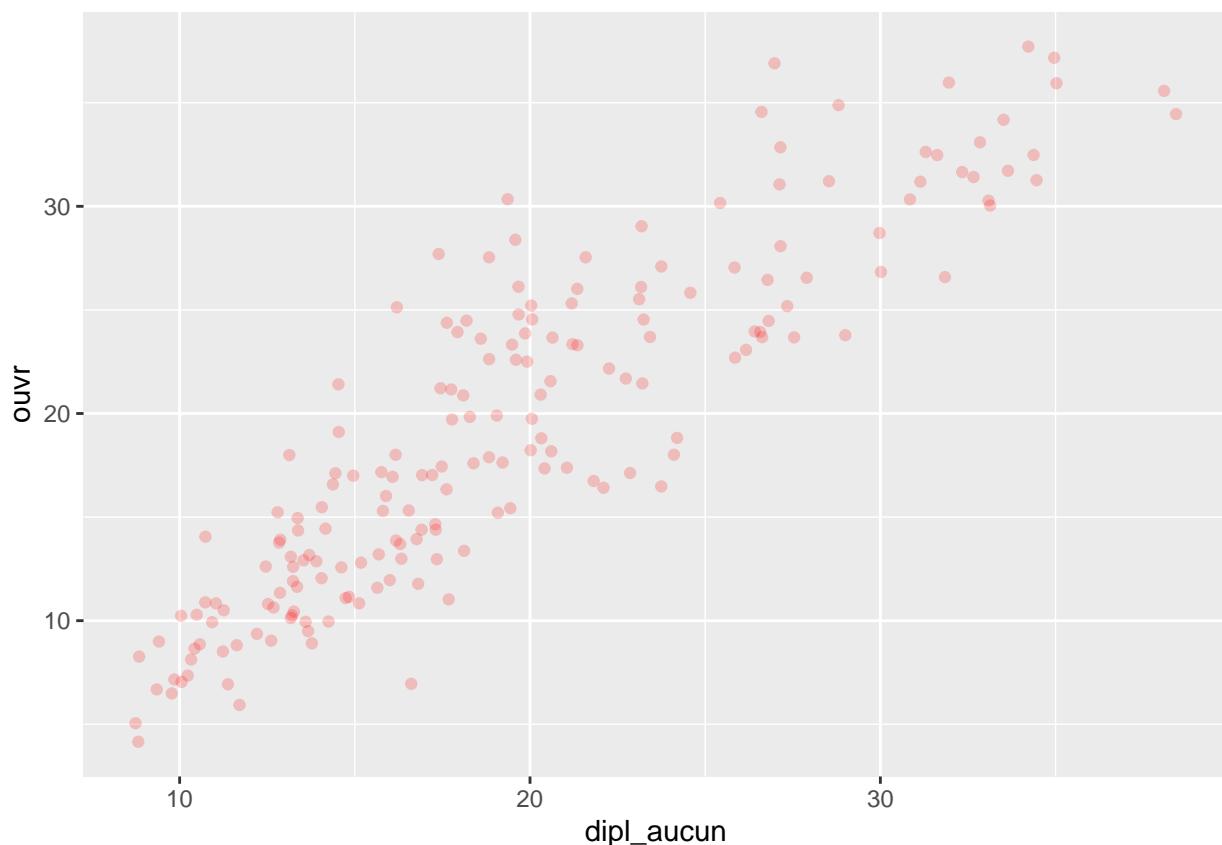
Exercice 1

Faire un nuage de points croisant le pourcentage de sans diplôme (`dipl_aucun`) et le pourcentage d'ouvriers (`ouvr`).

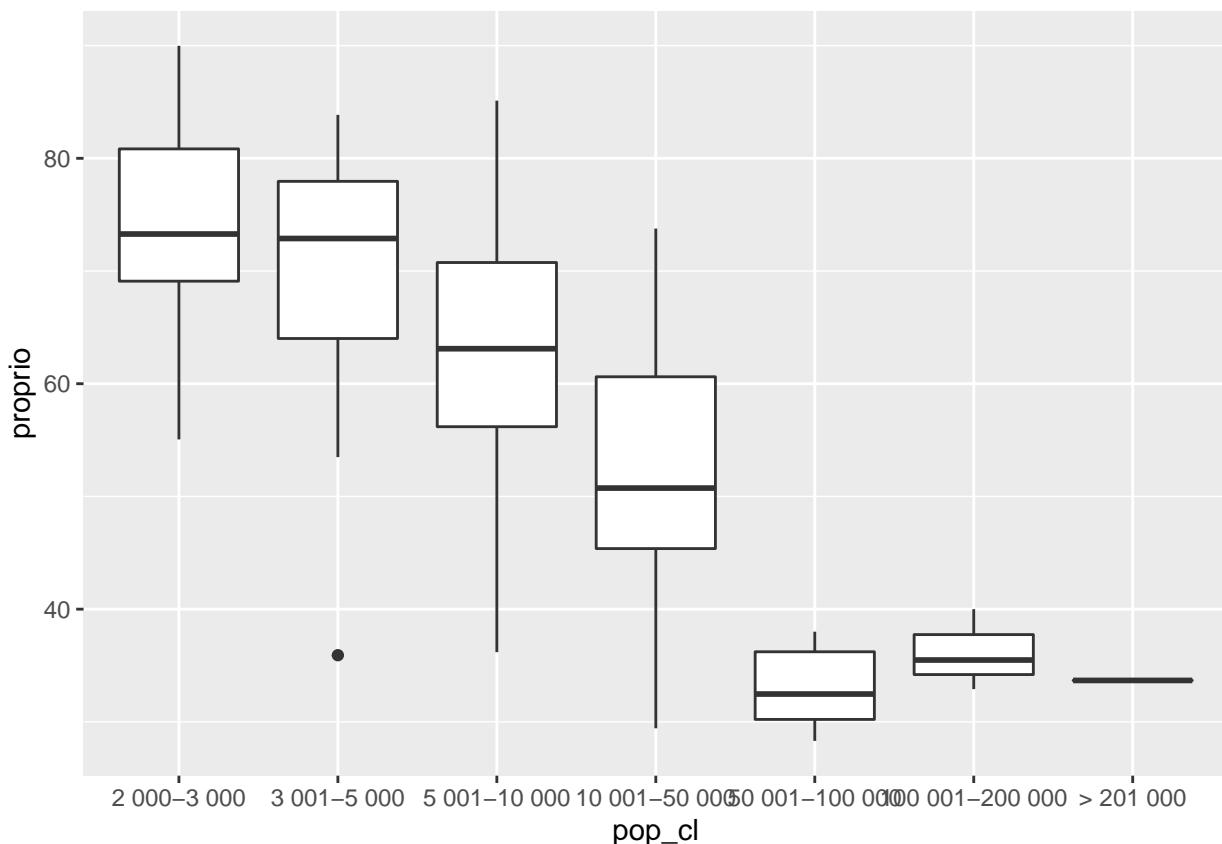


Exercice 2

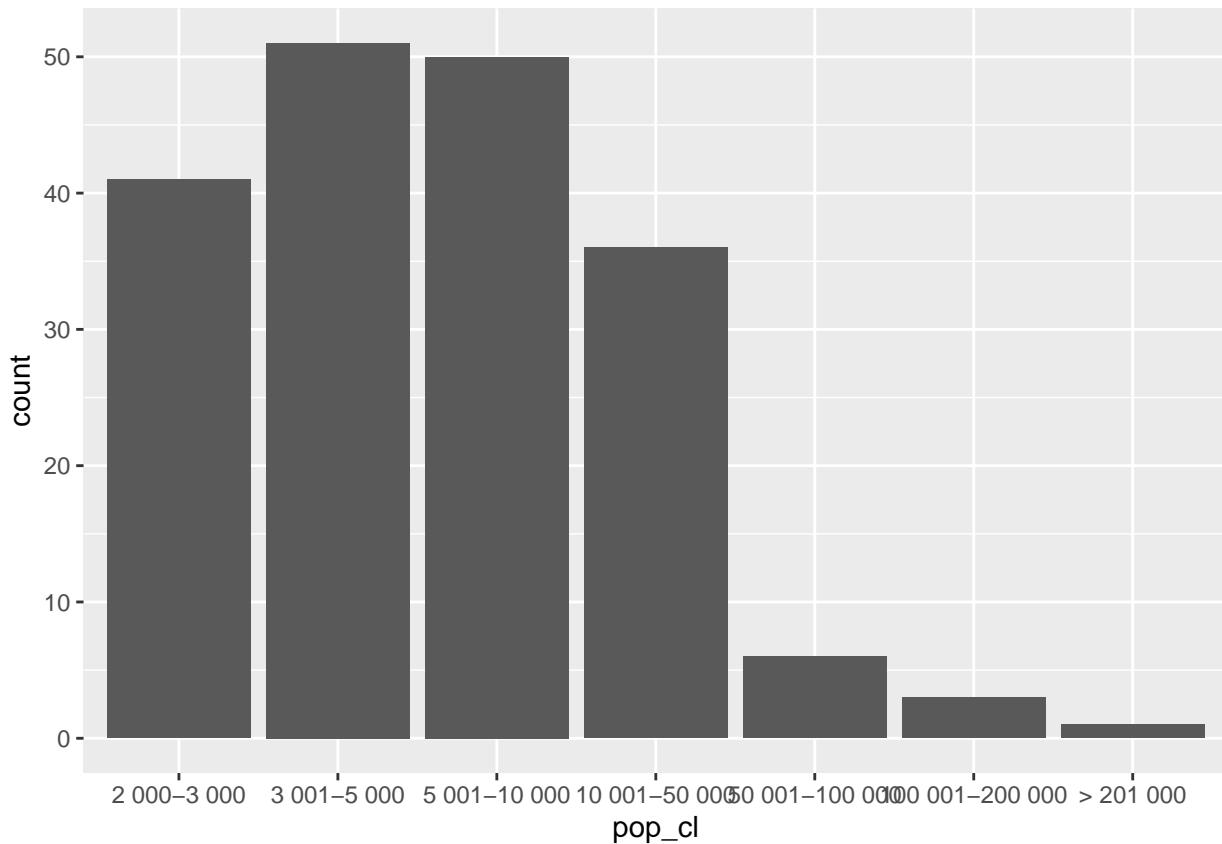
Faire un nuage de points croisant le pourcentage de sans diplôme et le pourcentage d'ouvriers, avec les points en rouge et de transparence 0.2.

**Exercice 3**

Représenter la répartition du pourcentage de propriétaires (variable `proprio`) selon la taille de la commune en classes (variable `pop_cl`) sous forme de boîtes à moustaches.

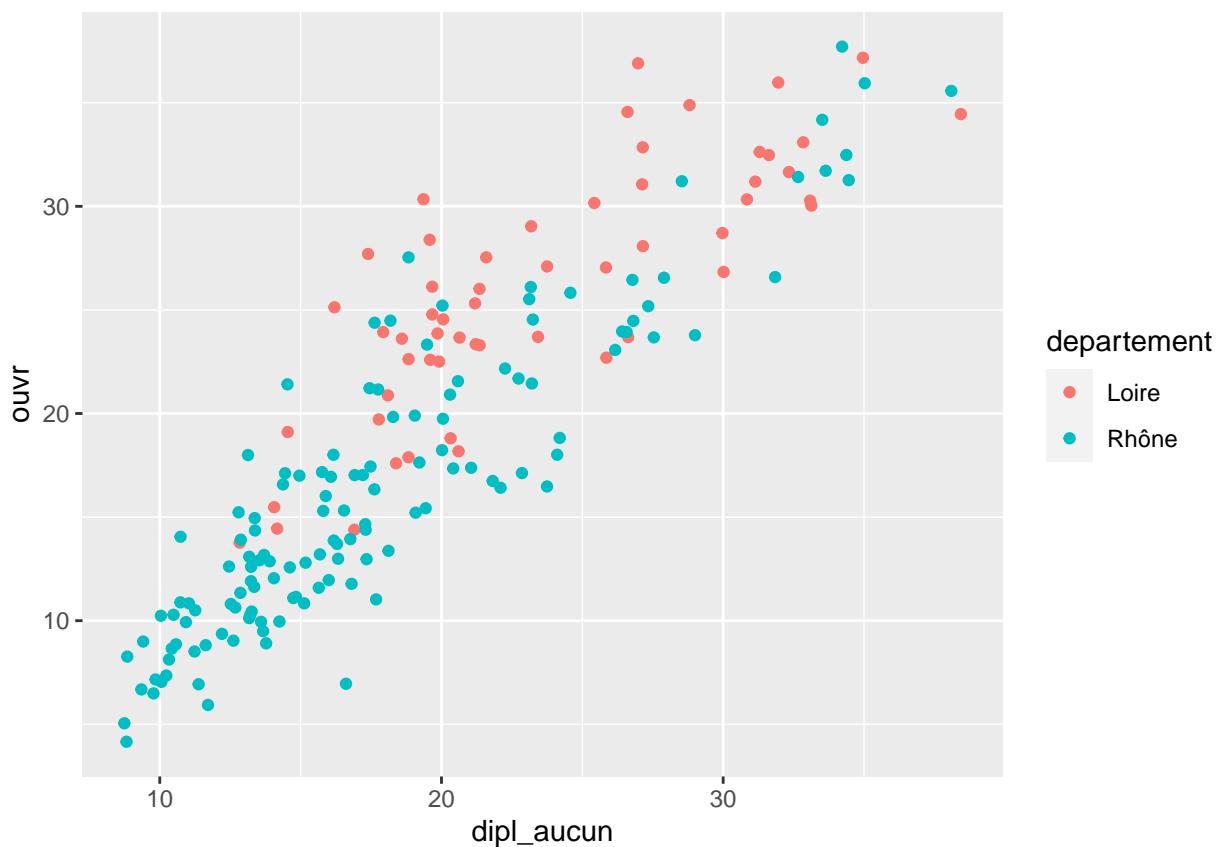
**Exercice 4**

Représenter la répartition du nombre de communes selon la taille de la commune en classes sous la forme d'un diagramme en bâtons.

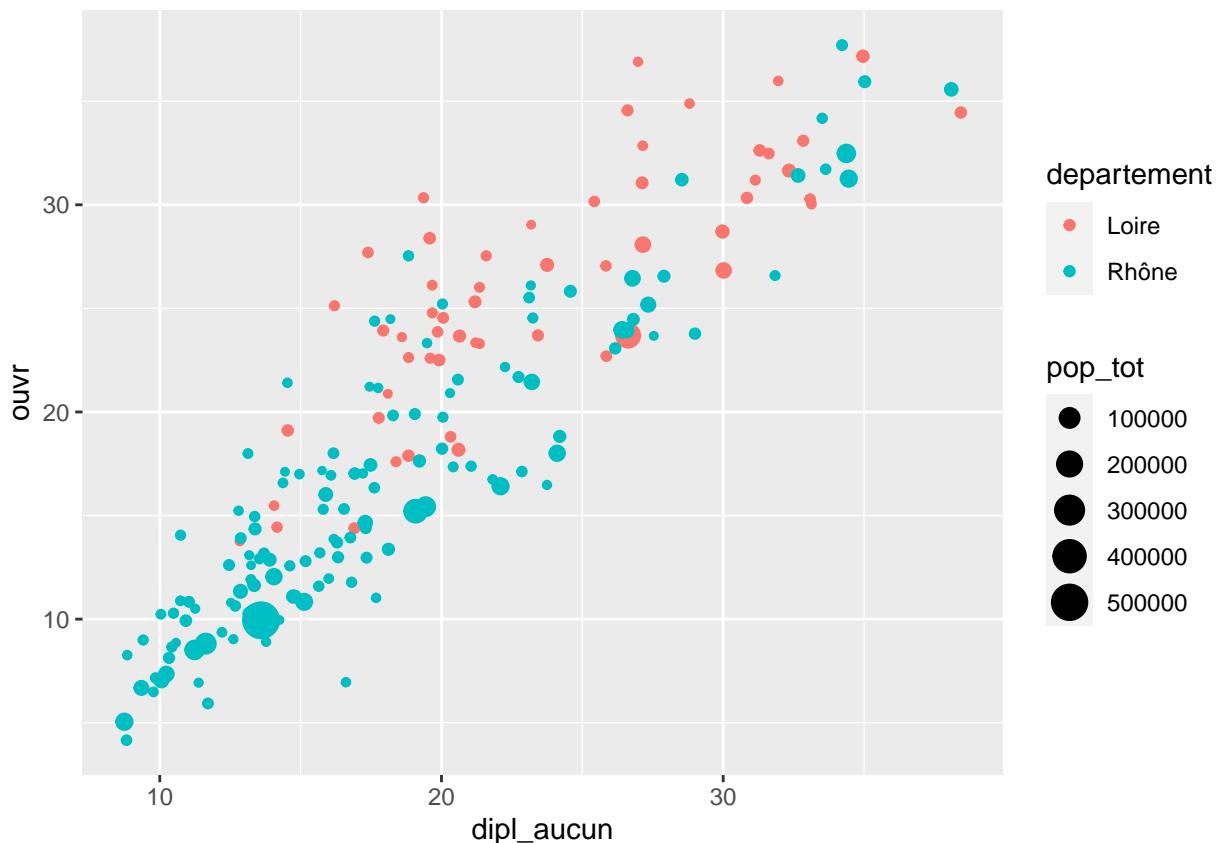


Exercice 5

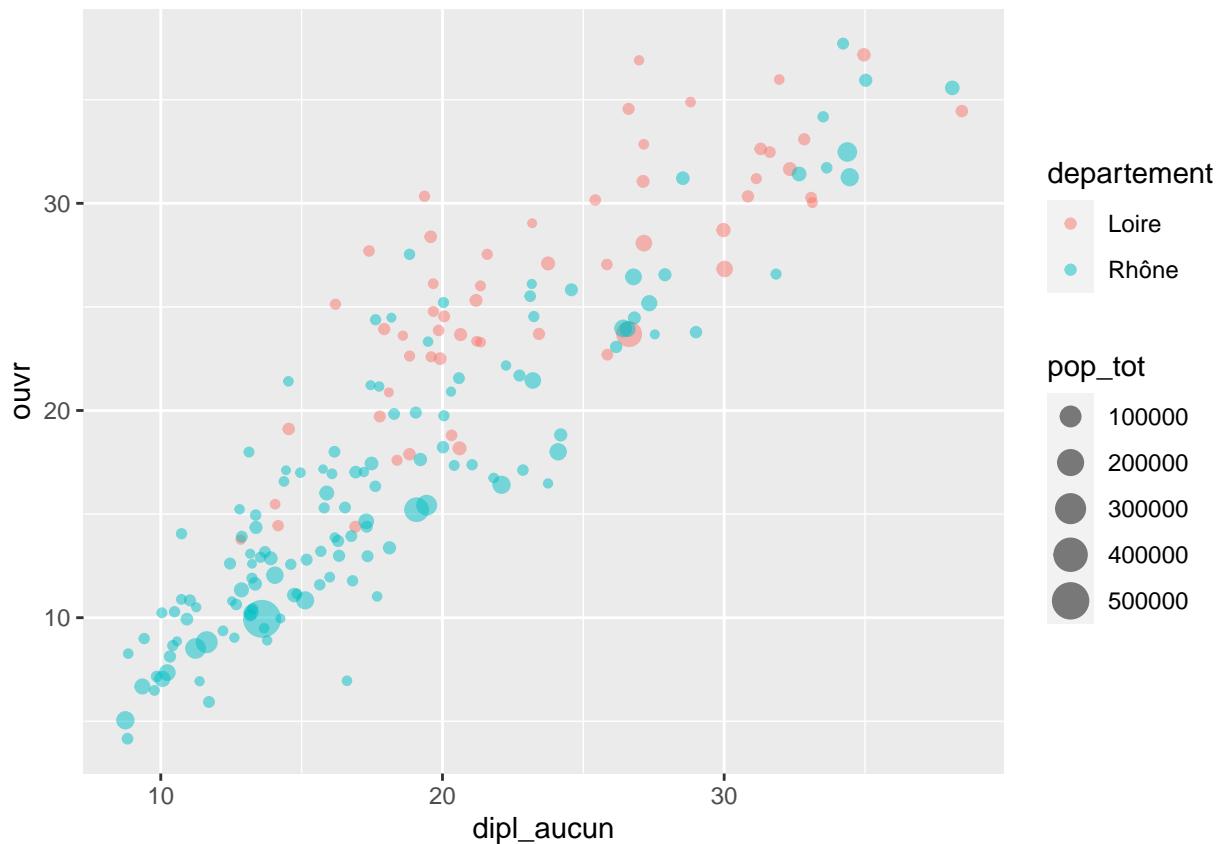
Faire un nuage de points croisant le pourcentage de sans diplôme et le pourcentage d'ouvriers. Faire varier la couleur selon le département (`departement`).



Sur le même graphique, faire varier la taille des points selon la population totale de la commune (`pop_tot`).

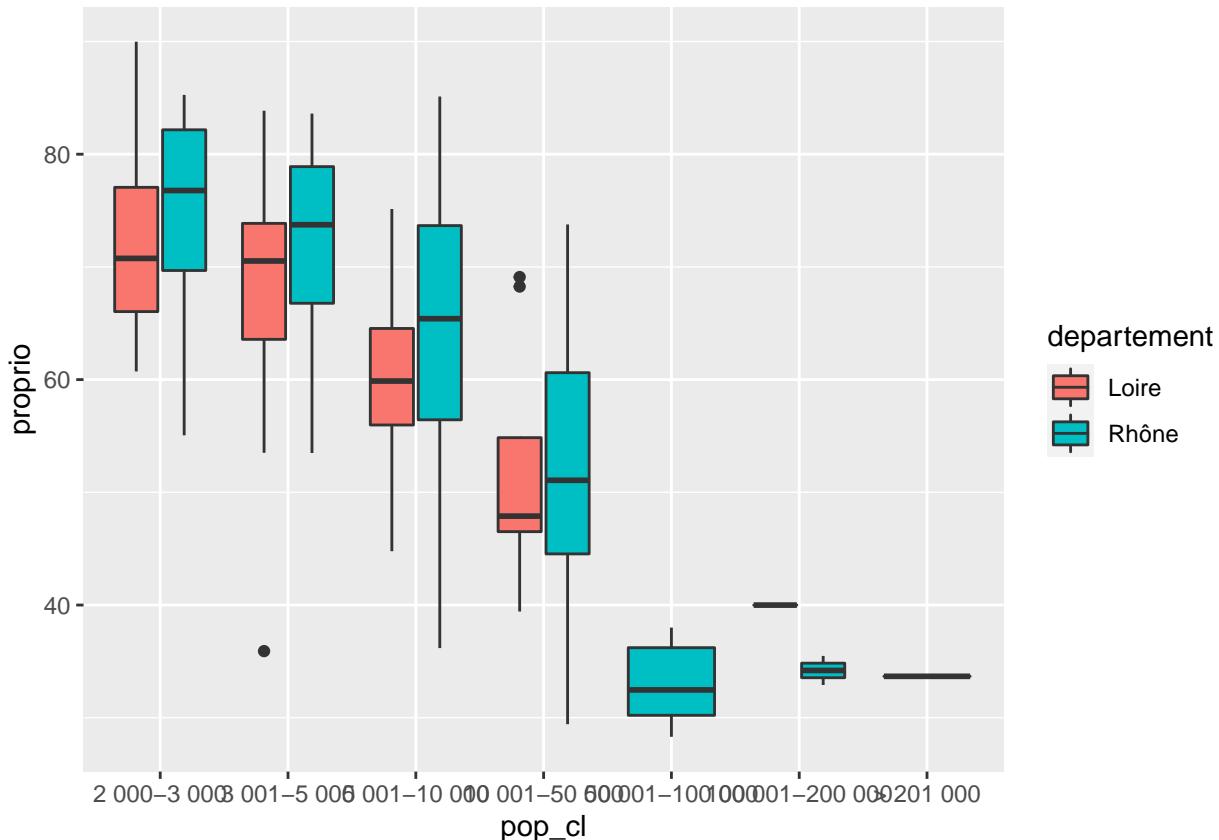


Enfin, toujours sur le même graphique, rendre les points transparents en plaçant leur opacité à 0.5.



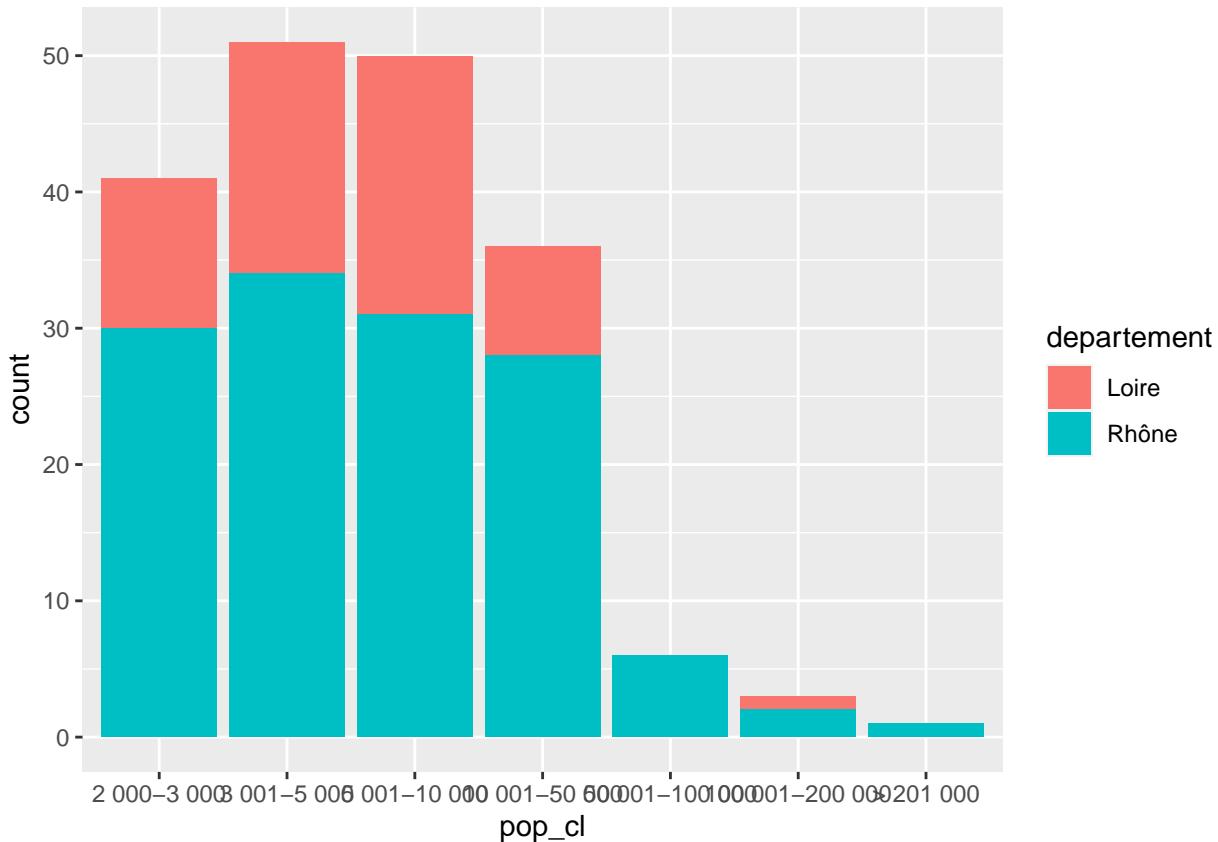
Exercice 6

Représenter la répartition du pourcentage de propriétaires (variable `proprio`) selon la taille de la commune en classes (variable `pop_cl`) sous forme de boîtes à moustaches. Faire varier la couleur de remplissage (attribut `fill`) selon le département.

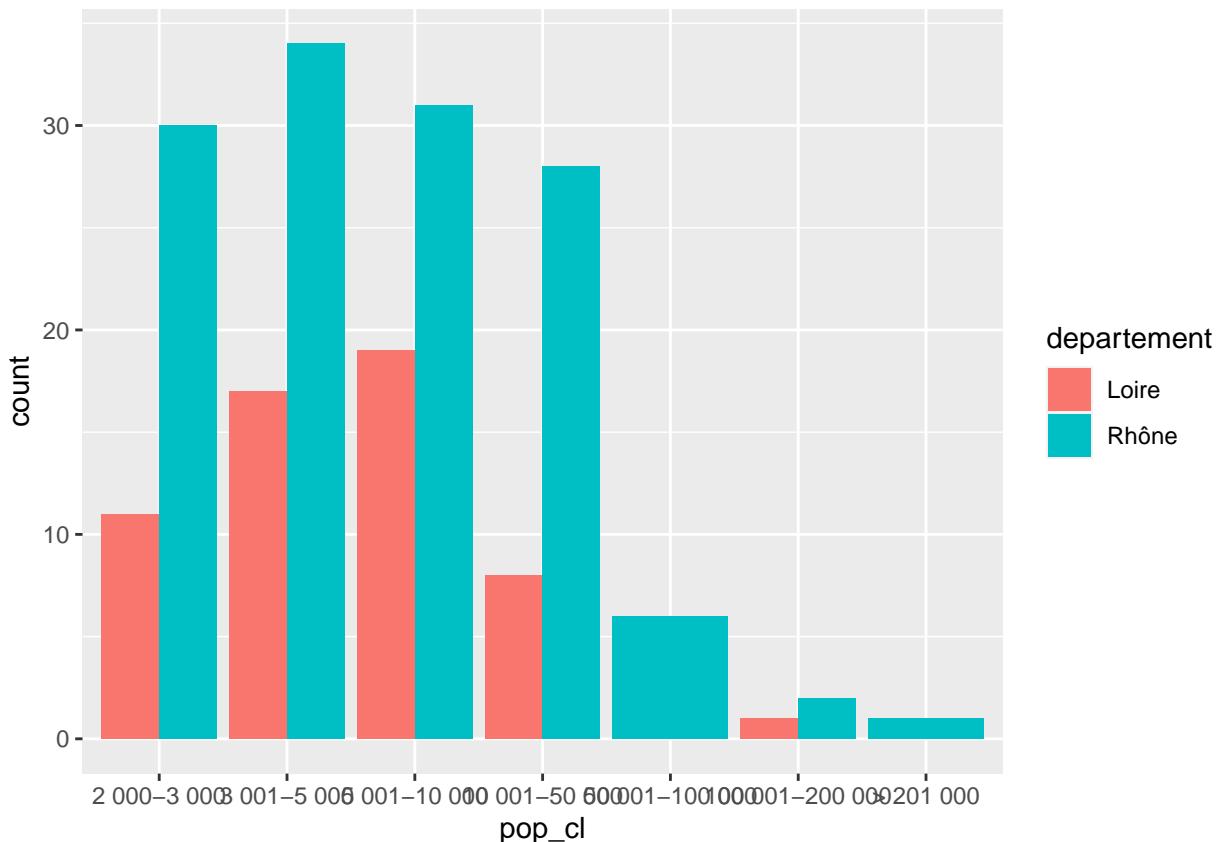


Exercice 7

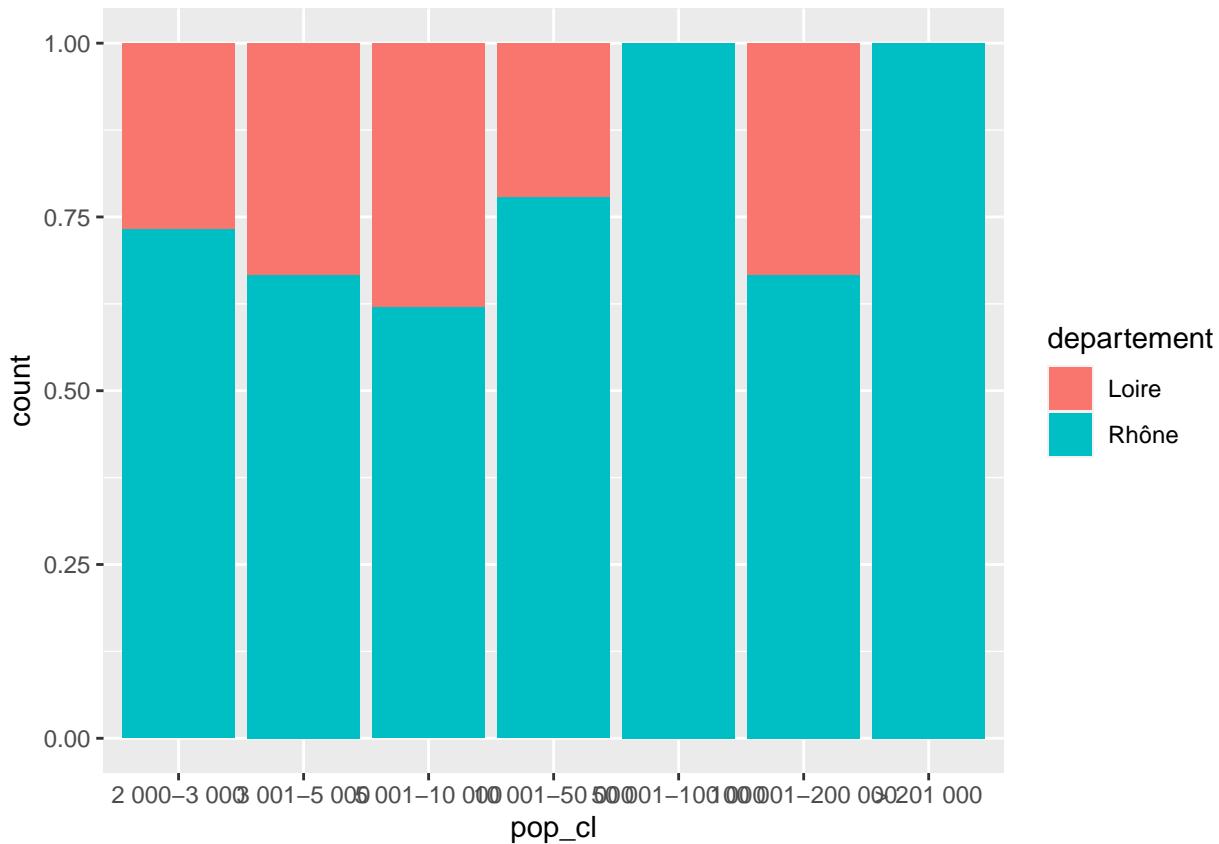
Représenter la répartition du nombre de communes selon la taille de la commune en classes (variable `pop_cl`) sous forme de diagramme en bâtons empilés, avec une couleur différente selon le département.



Faire varier la valeur du paramètre `position` pour afficher les barres les unes à côté des autres.

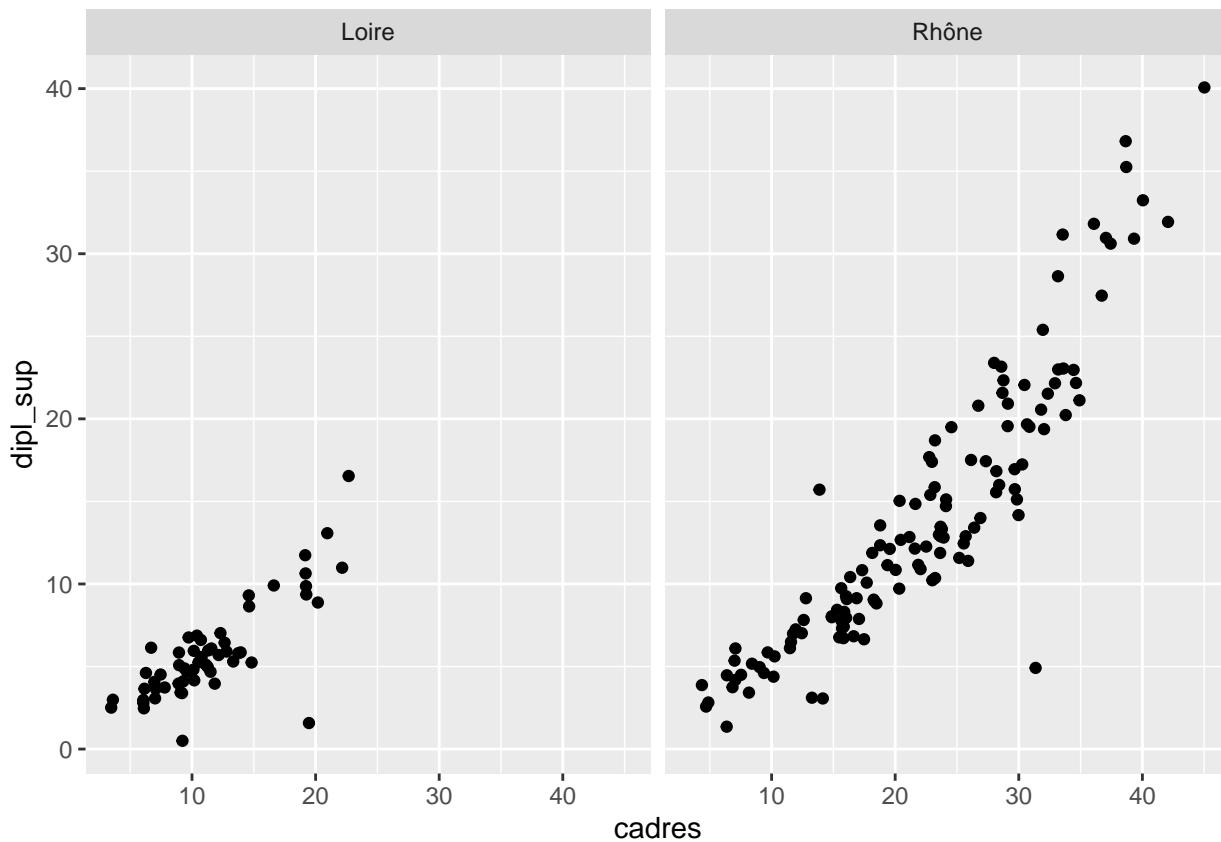


Changer à nouveau la valeur du paramètre `position` pour représenter les proportions de communes de chaque département pour chaque catégorie de taille.

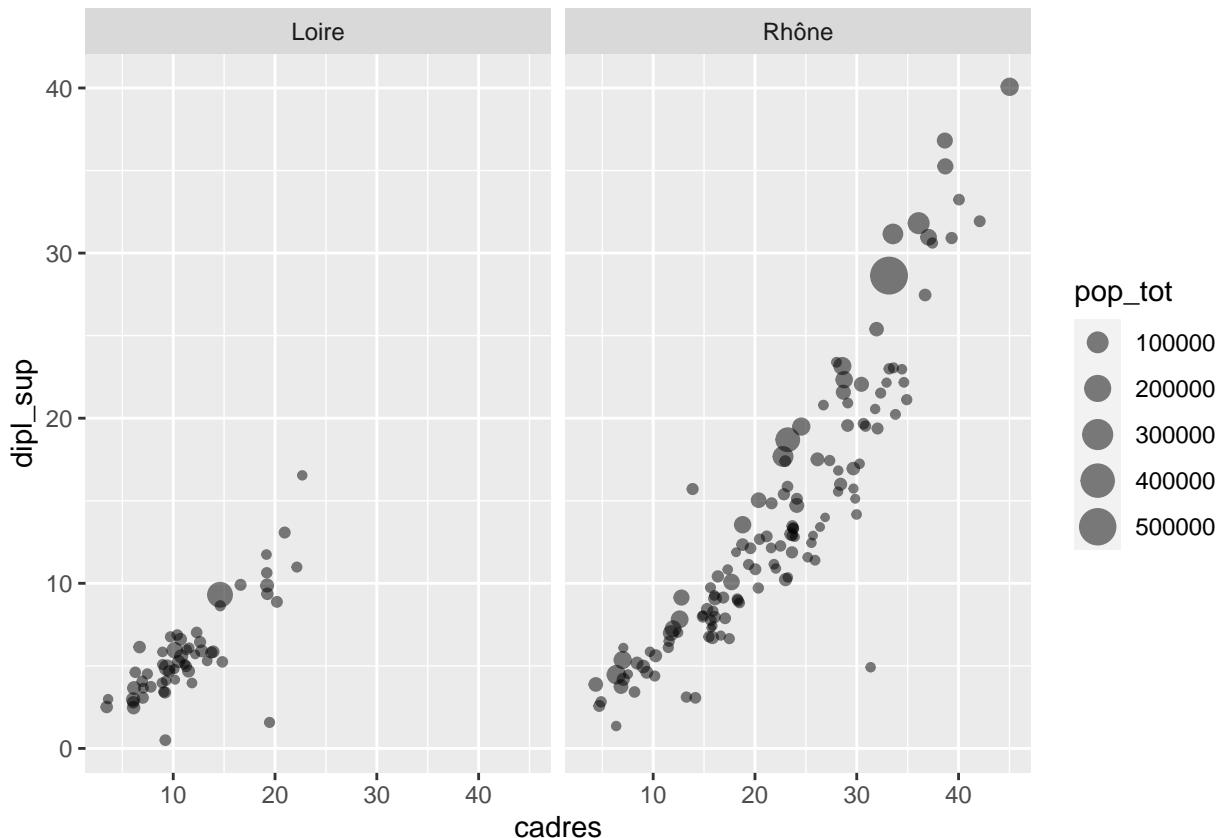


Exercice 8

Faire un nuage de points représentant en abscisse le pourcentage de cadres (`cadres`) et en ordonnée le pourcentage de diplômés du supérieur (`dipl_sup`). Représenter ce nuage par deux graphiques différents selon le département en utilisant `facet_grid`.



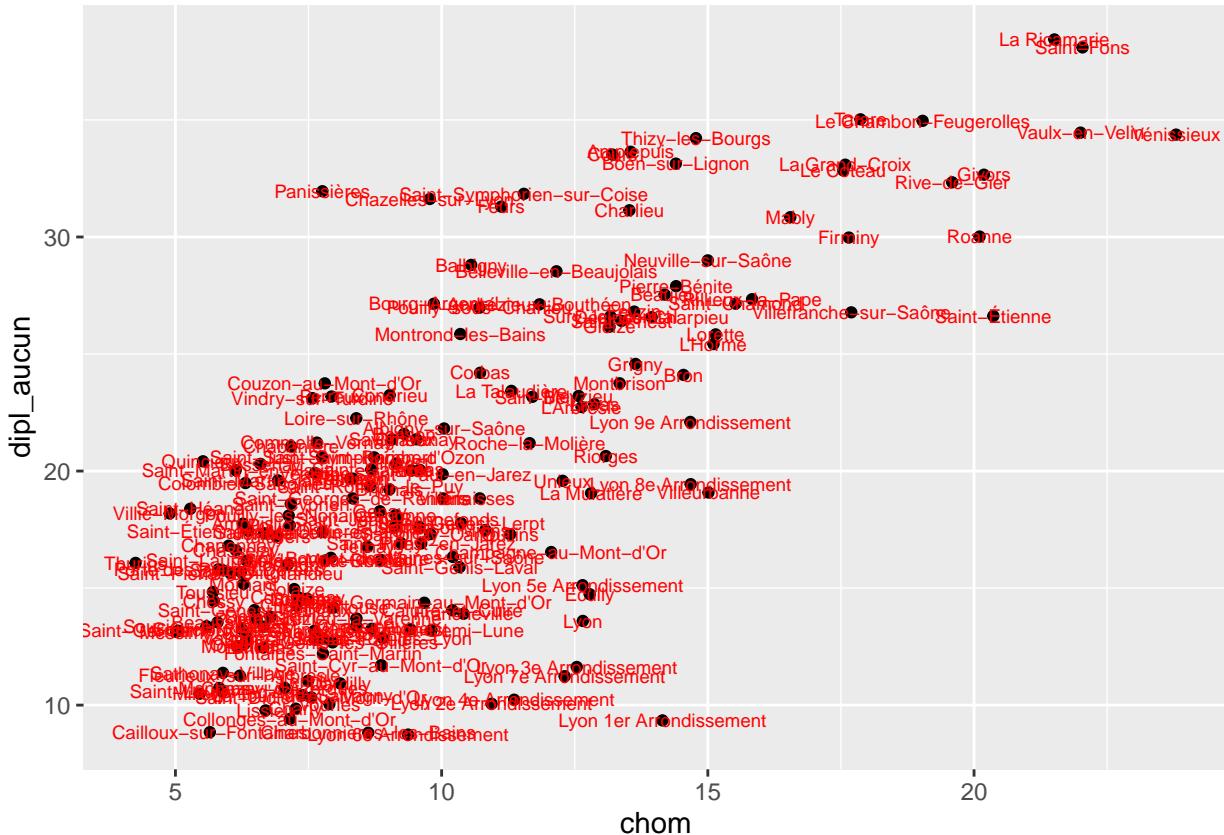
Sur le même graphique, faire varier la taille des points selon la population totale de la communes (variable `pop_tot`) et rendre les points transparents.



Exercice 9

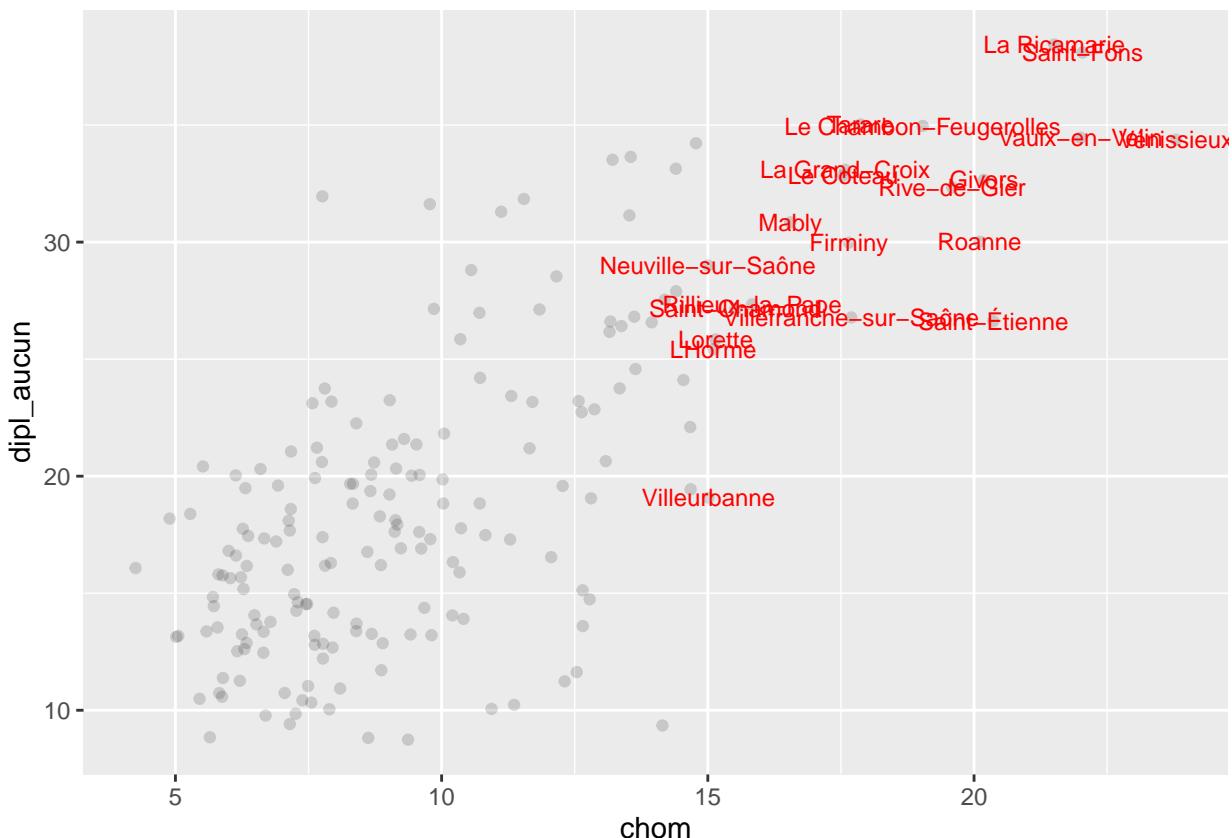
Faire le nuage de points croisant pourcentage de chômeurs (`chom`) et pourcentage de sans diplôme. Y ajouter

les noms des communes correspondant (variable `commune`), en rouge et en taille 2.5 :



Exercice 10

Dans le graphique précédent, n'afficher que le nom des communes ayant plus de 15% de chômage.



Chapitre 9

Recoder des variables

9.1 Rappel sur les variables et les vecteurs

Dans R, une variable, en général une colonne d'un tableau de données, est un objet de type *vecteur*. Un vecteur est un ensemble d'éléments, tous du même type.

On a vu qu'on peut construire un vecteur manuellement de différentes manières :

```
couleur <- c("Jaune", "Jaune", "Rouge", "Vert")
nombres <- 1:10
```

Mais le plus souvent on manipule des vecteurs faisant partie d'une table importée dans R. Dans ce qui suit on va utiliser le jeu de données d'exemple `hdv2003` de l'extension `questionr`.

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

Quand on veut accéder à un vecteur d'un tableau de données, on peut utiliser l'opérateur `$` :

```
hdv2003$qualif
```

On peut facilement créer de nouvelles variables (ou colonnes) dans un tableau de données en utilisant le `$` dans une assignation :

```
hdv2003$minutes.tv <- hdv2003$heures.tv * 60
```

Les vecteurs peuvent être de classes différentes, selon le type de données qu'ils contiennent.

On a ainsi des vecteurs de type `integer` ou `double`, qui contiennent respectivement des nombres entiers ou décimaux :

```
typeof(hdv2003$age)
#> [1] "integer"
```

```
typeof(hdv2003$heures.tv)
#> [1] "double"
```

Des vecteurs de type `character`, qui contiennent des chaînes de caractères :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
typeof(vec)
#> [1] "character"
```

Et des vecteurs de type `logical`, qui ne peuvent contenir que les valeurs vraie (`TRUE`) ou fausse (`FALSE`).

```
vec <- c(TRUE, FALSE, FALSE, TRUE)
typeof(vec)
#> [1] "logical"
```

On peut convertir un vecteur d'un type en un autre en utilisant les fonctions `as.numeric`, `as.character` ou `as.logical`. Les valeurs qui n'ont pas pu être converties sont automatiquement transformées en `NA`.

```
x <- c("1", "2.35", "8.2e+03", "foo")
as.numeric(x)
#> Warning: NAs introduced lors de la conversion automatique
#> [1] 1.00 2.35 8200.00 NA

y <- 2:6
as.character(y)
#> [1] "2" "3" "4" "5" "6"
```

On peut sélectionner certains éléments d'un vecteur à l'aide de l'opérateur `[]`. La manière la plus simple est d'indiquer la position des éléments qu'on veut sélectionner :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec[c(1, 3)]
#> [1] "Jaune" "Rouge"
```

La sélection peut aussi être utilisée pour modifier certains éléments d'un vecteur, par exemple :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec[2] <- "Violet"
vec
#> [1] "Jaune" "Violet" "Rouge" "Vert"
```

9.2 Tests et comparaison

Un test est une opération logique de comparaison qui renvoie vrai (`TRUE`) ou faux (`FALSE`) pour chacun des éléments d'un vecteur.

Parmi les opérateurs de comparaison disponibles, on trouve notamment :

- `==` qui teste l'égalité
- `!=` qui teste la différence
- `>`, `<`, `<=`, `>=` qui testent la supériorité ou l'infériorité
- `%in%` qui teste l'appartenance à un ensemble de valeurs

Exemple le plus simple :

```
2 == 3
#> [1] FALSE
```

```
2 != 3
#> [1] TRUE
```

Exemple appliqué à un vecteur :

```
x <- 1:10
x < 5
#> [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

On peut combiner plusieurs tests avec les opérateurs logiques *et* (`&`) et *ou* (`|`). Ainsi, si on veut tester qu'une valeur est comprise entre 3 et 6 inclus, on peut faire :

```
x >= 3 & x <= 6
#> [1] FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
```

Si on veut tester qu'une valeur est égale à “Bleu” ou à “Vert”, on peut faire :

```
vec <- c("Jaune", "Jaune", "Rouge", "Vert")
vec == "Jaune" | vec == "Vert"
#> [1] TRUE TRUE FALSE TRUE
```

À noter que dans ce cas, on peut utiliser l'opérateur `%in%1`, qui teste si une valeur fait partie des éléments d'un vecteur :

```
vec %in% c("Jaune", "Vert")
#> [1] TRUE TRUE FALSE TRUE
```

 Attention, si on souhaite tester si une valeur `x` est inconnue (ou ‘manquante’), c'est-à-dire si elle est codée `NA` (*Not Available*), faire le test `x == NA` ne donnera pas le résultat escompté. En effet, fidèle à sa réputation de rigueur informatienne, pour R `NA == NA` ne vaut pas `TRUE` mais... `NA` (on ne sait pas si une valeur inconnue est égale à une autre valeur inconnue).

Pour tester si une valeur est inconnue (`NA`), il faut utiliser la fonction dédiée `is.na` et faire `is.na(x)`.

Cependant, par convention, `NA %in% NA` vaut `TRUE`.

¹Pour accéder à la page de documentation de fonctions comme `%in%`, on ne peut pas utiliser `?%in%`, qui renvoie une erreur. Vous pouvez faire `?"%in%"`, `help("%in%")` ou, dans ce cas, `?match`, car les deux fonctions sont documentées sur la même page d'aide.

Enfin, on peut inverser un test avec l'opérateur *non* (!) :

```
!(vec %in% c("Jaune", "Vert"))
#> [1] FALSE FALSE TRUE FALSE
```

Les tests sont notamment utilisés par le verbe **filter** de **dplyr** (voir section 10.2.2) qui permet de sélectionner certaines lignes d'un tableau de données. On peut ainsi sélectionner les individus ayant entre 20 et 40 ans en filtrant sur la variable **age** :

```
filter(hdv2003, age >= 20 & age <= 40)
```

Ou sélectionner les personnes ayant comme catégorie socio-professionnelle **Ouvrier specialise** ou **Ouvrier qualifie** en filtrant sur la variable **qualif** :

```
filter(hdv2003, qualif %in% c("Ouvrier specialise", "Ouvrier qualifie"))
```

On peut utiliser les tests pour sélectionner certains éléments d'un vecteur. Si on passe un test à l'opérateur de sélection [], seuls les éléments pour lesquels ce test est vrai seront conservés :

```
x <- c(12, 8, 14, 7, 6, 18)
x[x > 10]
#> [1] 12 14 18
```

Enfin, on peut aussi utiliser les tests et la sélection pour modifier les valeurs d'un vecteur. Ainsi, si on assigne une valeur à une sélection, les éléments pour lesquels le test est vrai sont remplacés par cette valeur :

```
x <- c(12, 8, 14, 7, 6, 18)
x[x > 10] <- 100
x
#> [1] 100 8 100 7 6 100
```

En utilisant cette assignation via un test, on peut effectuer des recodages de variables. Soit le vecteur suivant :

```
vec <- c("Femme", "Homme", "Femme", "Garçon")
```

Si on souhaite recoder la modalité “Garçon” en “Homme”, on peut utiliser la syntaxe suivante :

```
vec[vec == "Garçon"] <- "Homme"
vec
#> [1] "Femme" "Homme" "Femme" "Homme"
```

Cette syntaxe est tout à fait valable et couramment utilisée. On va cependant voir dans la section suivante différentes fonctions qui facilitent ces opérations de recodage.

Pour plus d'informations sur les vecteurs et les structures de données de R, vous pouvez vous reporter au chapitre 16.

9.3 Recoder une variable qualitative

Pour rappel, on appelle variable qualitative une variable pouvant prendre un nombre limité de modalités (de valeurs possibles).

9.3.1 Facteurs et `forcats`

Dans R, les variables qualitatives peuvent être de deux types : ou bien des vecteurs de type `character` (des chaînes de caractères), ou bien des `factor` (facteurs). Si vous utilisez les fonctions des extensions du *tidyverse* comme `readr`, `readxl` ou `haven` pour importer vos données, vos variables qualitatives seront importées sous forme de `character`. Mais dans d'autres cas elles se retrouveront parfois sous forme de `factor`. C'est le cas de notre jeu de données d'exemple `hdv2003`.

```
class(hdv2003$qualif)
#> [1] "factor"
```

Les facteurs sont un type de variable ne pouvant prendre qu'un nombre défini de modalités nommés *levels*.

```
levels(hdv2003$qualif)
#> [1] "Ouvrier specialise"      "Ouvrier qualifie"
#> [3] "Technicien"             "Profession intermediaire"
#> [5] "Cadre"                  "Employe"
#> [7] "Autre"
```

Ceci complique les opérations de recodage car du coup l'opération suivante, qui tente de modifier une modalité de la variable, aboutit à un avertissement, et l'opération n'est pas effectuée.

```
hdv2003$qualif[hdv2003$qualif == "Ouvrier specialise"] <- "Ouvrier"
#> Warning in `[<-factor`(`*tmp*`, hdv2003$qualif == "Ouvrier specialise", :
#> niveau de facteur incorrect, NAs générés
```

`forcats` est une extension facilitant la manipulation des variables qualitatives, qu'elles soient sous forme de vecteurs `character` ou de facteurs. Elle fait partie du *tidyverse*, et est donc automatiquement chargée par :

```
library(tidyverse)
```

9.3.2 Modifier les modalités d'une variable qualitative

Une opération courante consiste à modifier les valeurs d'une variable qualitative, que ce soit pour avoir des intitulés plus courts ou plus clairs ou pour regrouper des modalités entre elles.

Il existe plusieurs possibilités pour effectuer ce type de recodage, mais ici on va utiliser la fonction `fct_recode` de l'extension `forcats`. Celle-ci prend en argument une liste de recodages sous la forme "`Nouvelle valeur`" = "`Ancienne valeur`".

Un exemple :

```
f <- c("Pomme", "Poire", "Pomme", "Cerise")
f <- fct_recode(
  f,
  "Fraise" = "Pomme",
  "Ananas" = "Poire"
)
f
#> [1] Fraise Ananas Fraise Cerise
#> Levels: Cerise Ananas Fraise
```

Autre exemple sur une “vraie” variable :

```
freq(hdv2003$qualif)
#>                               n   % val%
#> Ouvrier specialise          0   0.0  0.0
#> Ouvrier qualifie           292 14.6 20.1
#> Technicien                  86  4.3  5.9
#> Profession intermediaire  160  8.0 11.0
#> Cadre                       260 13.0 17.9
#> Employe                     594 29.7 41.0
#> Autre                        58  2.9  4.0
#> NA                           550 27.5  NA
```

```
hdv2003$qualif5 <- fct_recode(
  hdv2003$qualif,
  "Ouvrier" = "Ouvrier specialise",
  "Ouvrier" = "Ouvrier qualifie",
  "Interm" = "Technicien",
  "Interm" = "Profession intermediaire"
)

freq(hdv2003$qualif5)
#>                               n   % val%
#> Ouvrier 292 14.6 20.1
#> Interm  246 12.3 17.0
#> Cadre   260 13.0 17.9
#> Employe 594 29.7 41.0
#> Autre    58  2.9  4.0
#> NA      550 27.5  NA
```

Attention, les anciennes valeurs saisies doivent être exactement égales aux valeurs des modalités de la variable recodée : toute différence d’accent ou d’espace fera que ce recodage ne sera pas pris en compte. Dans ce cas, `forcats` affiche un avertissement nous indiquant qu’une valeur saisie n’a pas été trouvée dans les modalités de la variable.

```
hdv2003$qualif_test <- fct_recode(
  hdv2003$qualif,
  "Ouvrier" = "Ouvrier spécialisé",
  "Ouvrier" = "Ouvrier qualifié"
)
#> Warning: Unknown levels in `f`: Ouvrier spécialisé, Ouvrier qualifié
```

Si on souhaite recoder une modalité de la variable en NA, il faut (contre intuitivement) lui assigner la valeur NULL.

```
hdv2003$qualif_rec <- fct_recode(
  hdv2003$qualif,
  NULL = "Autre"
)

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Ouvrier specialise          0   0.0  0.0
#> Ouvrier qualifie           292 14.6 21.0
#> Technicien                  86  4.3  6.2
#> Profession intermediaire  160  8.0 11.5
#> Cadre                       260 13.0 18.7
#> Employe                     594 29.7 42.7
#> NA                           608 30.4  NA
```

À l'inverse, si on souhaite recoder les NA d'une variable, on utilisera la fonction `fct_explicit_na`, qui convertit toutes les valeurs manquantes (NA) d'un facteur en une modalité spécifique.

```
hdv2003$qualif_rec <- fct_explicit_na(hdv2003$qualif, na_level = "(Manquant)")

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Ouvrier specialise          0   0.0  0.0
#> Ouvrier qualifie           292 14.6 14.6
#> Technicien                  86  4.3  4.3
#> Profession intermediaire  160  8.0  8.0
#> Cadre                       260 13.0 13.0
#> Employe                     594 29.7 29.7
#> Autre                        58  2.9  2.9
#> (Manquant)                  550 27.5 27.5
```

D'autres fonctions sont proposées par `forcats` pour faciliter certains recodage, comme `fct_collapse`, qui propose une autre syntaxe pratique quand on doit regrouper ensemble des modalités.

```
hdv2003$qualif_rec <- fct_collapse(
  hdv2003$qualif,
  "Ouvrier" = c("Ouvrier specialise", "Ouvrier qualifie"),
  "Interm" = c("Technicien", "Profession intermediaire")
)

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Ouvrier 292 14.6 20.1
#> Interm  246 12.3 17.0
#> Cadre   260 13.0 17.9
#> Employe 594 29.7 41.0
#> Autre    58  2.9  4.0
#> NA      550 27.5  NA
```

`fct_other`, qui regroupe une liste de modalités en une seule modalité “Other”.

```
hdv2003$qualif_rec <- fct_other(
  hdv2003$qualif,
  drop = c("Ouvrier specialise", "Ouvrier qualifie", "Cadre", "Autre")
)

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Technicien                   86  4.3  5.9
#> Profession intermediaire 160  8.0 11.0
#> Employe                      594 29.7 41.0
#> Other                          610 30.5 42.1
#> NA                            550 27.5   NA
```

`fct_lump`, qui regroupe automatiquement les modalités les moins fréquentes en une seule modalité “Other” (avec possibilité d’indiquer des seuils de regroupement).

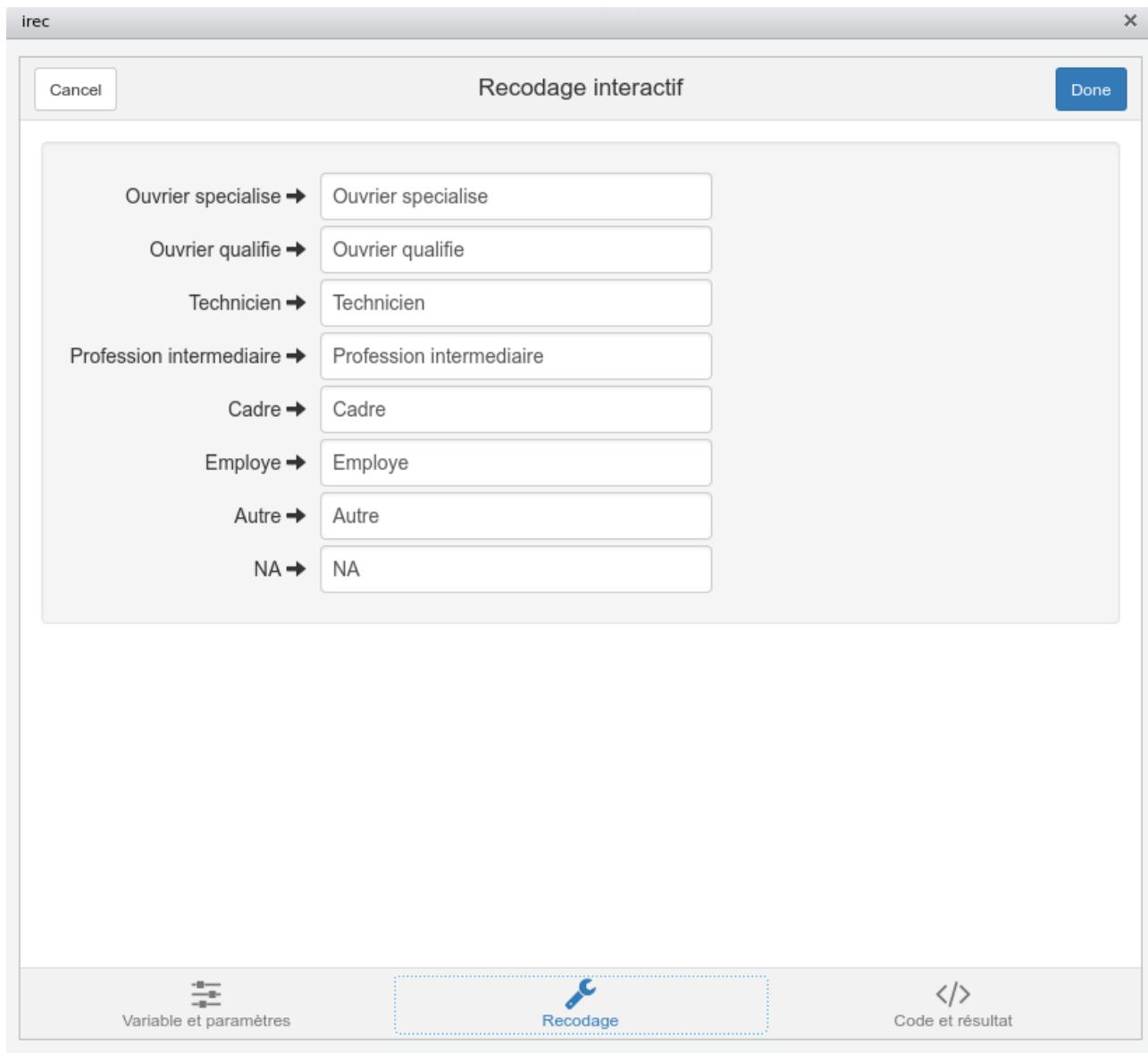
```
hdv2003$qualif_rec <- fct_lump(hdv2003$qualif)

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Ouvrier qualifie            292 14.6 20.1
#> Profession intermediaire 160  8.0 11.0
#> Cadre                         260 13.0 17.9
#> Employe                      594 29.7 41.0
#> Other                          144  7.2  9.9
#> NA                            550 27.5   NA
```

9.3.2.1 Interface graphique de recodage

L’extension `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 `irec()` en lui passant comme paramètre la variable à recoder.

Figure 9.1: Interface graphique de `irec`

L'interface se compose de trois onglets : l'onglet *Variable et paramètres* vous permet de sélectionner la variable à recoder, le nom de la nouvelle variable et d'autres paramètres, l'onglet *Recodages* vous permet de saisir les nouvelles valeurs des modalités, et l'onglet *Code et résultat* affiche le code R correspondant ainsi qu'un tableau permettant de vérifier les résultats.

Une fois votre recodage terminé, cliquez sur le bouton *Done* et le code R sera inséré dans votre script R ou affiché dans la console.



Attention, cette interface est prévue pour ne pas modifier vos données. C'est donc à vous d'exécuter le code généré pour que le recodage soit réellement effectif.

9.3.3 Ordonner les modalités d'une variable qualitative

L'avantage des facteurs (par rapport aux vecteurs de type `character`) est que leurs modalités peuvent être ordonnées, ce qui peut faciliter la lecture de tableaux ou graphiques.

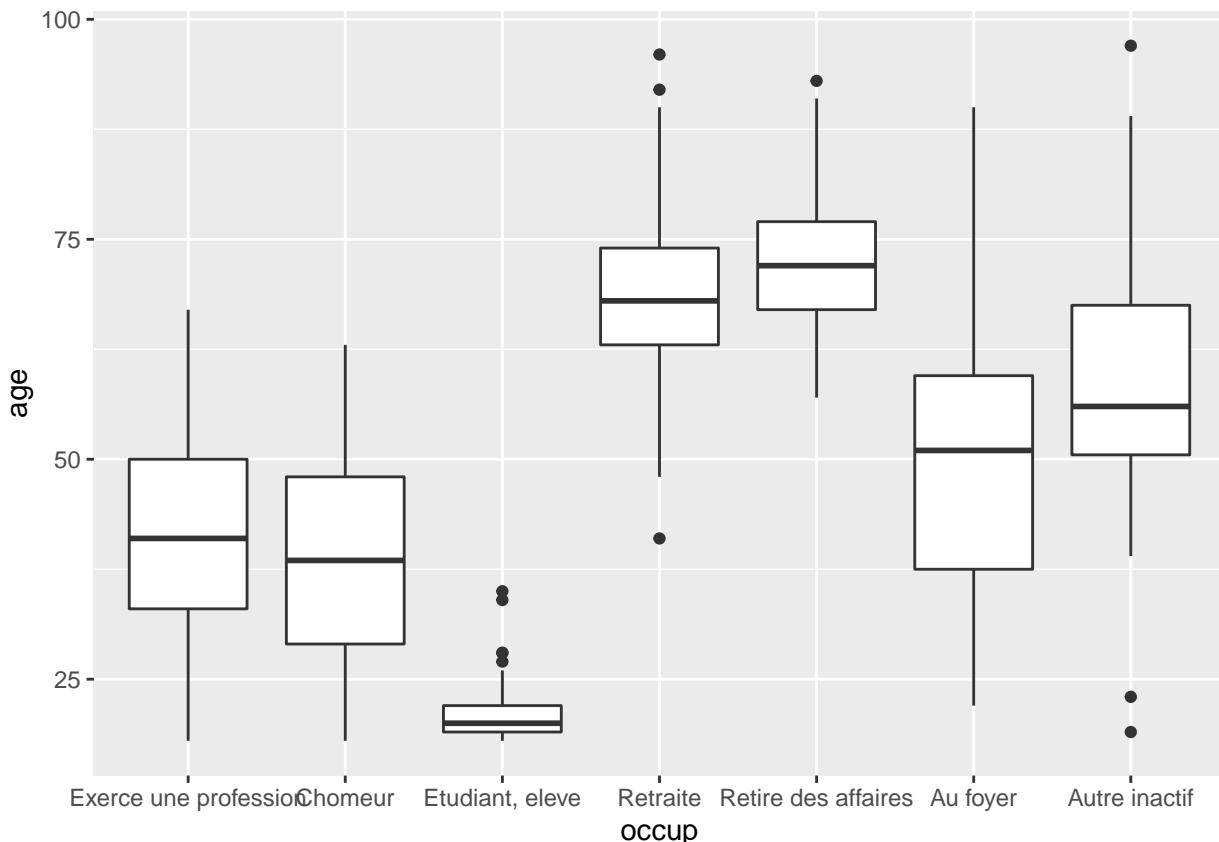
On peut ordonner les modalités d'un facteur manuellement, par exemple avec la fonction `fct_relevel()` de l'extension `forcats`.

```
hdv2003$qualif_rec <- fct_relevel(
  hdv2003$qualif,
  "Cadre", "Profession intermediaire", "Technicien",
  "Employe", "Ouvrier qualifie", "Ouvrier specialise",
  "Autre"
)

freq(hdv2003$qualif_rec)
#>                               n   % val%
#> Cadre                      260 13.0 17.9
#> Profession intermediaire 160  8.0 11.0
#> Technicien                  86  4.3  5.9
#> Employe                     594 29.7 41.0
#> Ouvrier qualifie           292 14.6 20.1
#> Ouvrier specialise          0   0.0  0.0
#> Autre                       58  2.9  4.0
#> NA                          550 27.5  NA
```

Une autre possibilité est d'ordonner les modalités d'un facteur selon les valeurs d'une autre variable. Par exemple, si on représente le boxplot de la répartition de l'âge selon le statut d'occupation :

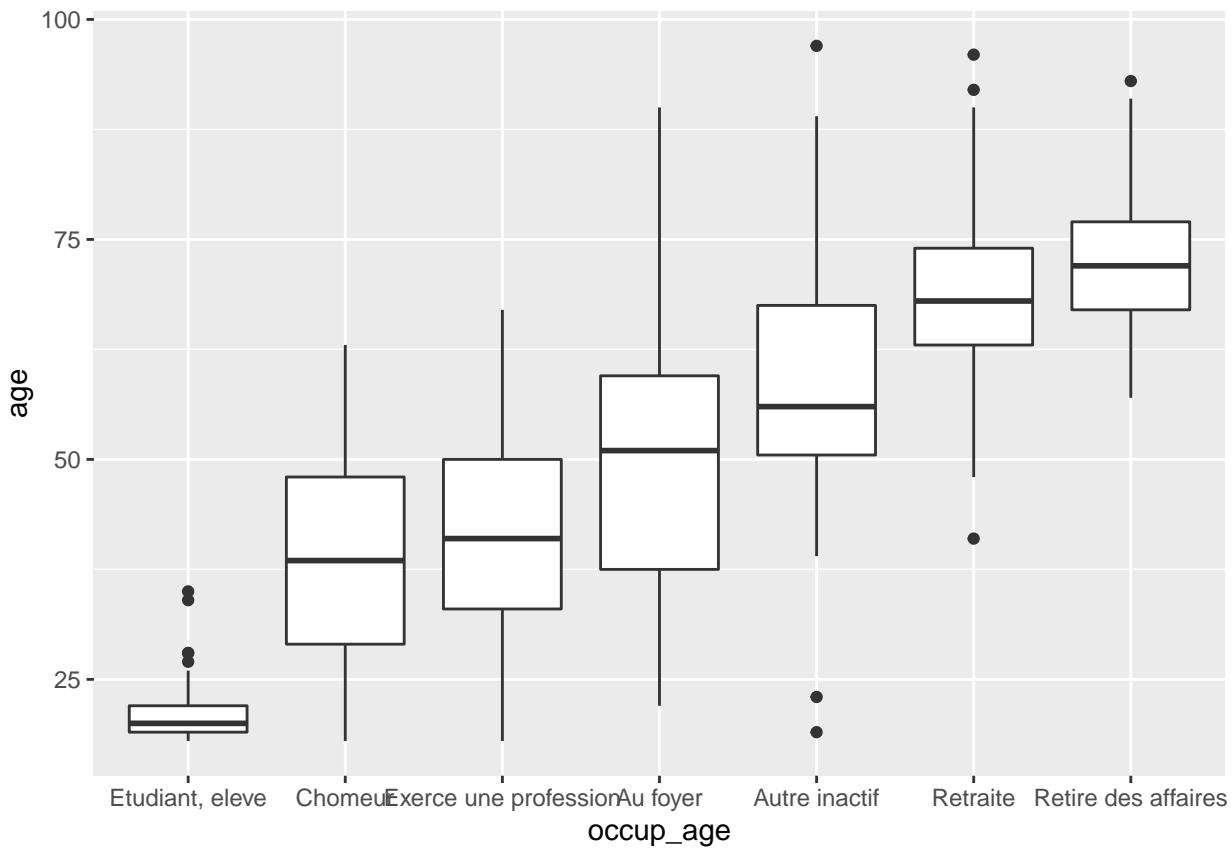
```
library(ggplot2)
ggplot(hdv2003) +
  geom_boxplot(aes(x = occup, y = age))
```



Le graphique pourrait être plus lisible si les modalités étaient triées par âge médian croissant. On peut dans ce cas utiliser la fonction `fct_reorder`. Celle-ci prend 3 arguments : le facteur à réordonner, la variable dont les valeurs doivent être utilisées pour ce réordonnement, et enfin une fonction à appliquer à cette deuxième variable.

```
hdv2003$occup_age <- fct_reorder(hdv2003$occup, hdv2003$age, median)

ggplot(hdv2003) +
  geom_boxplot(aes(x = occup_age, y = age))
```

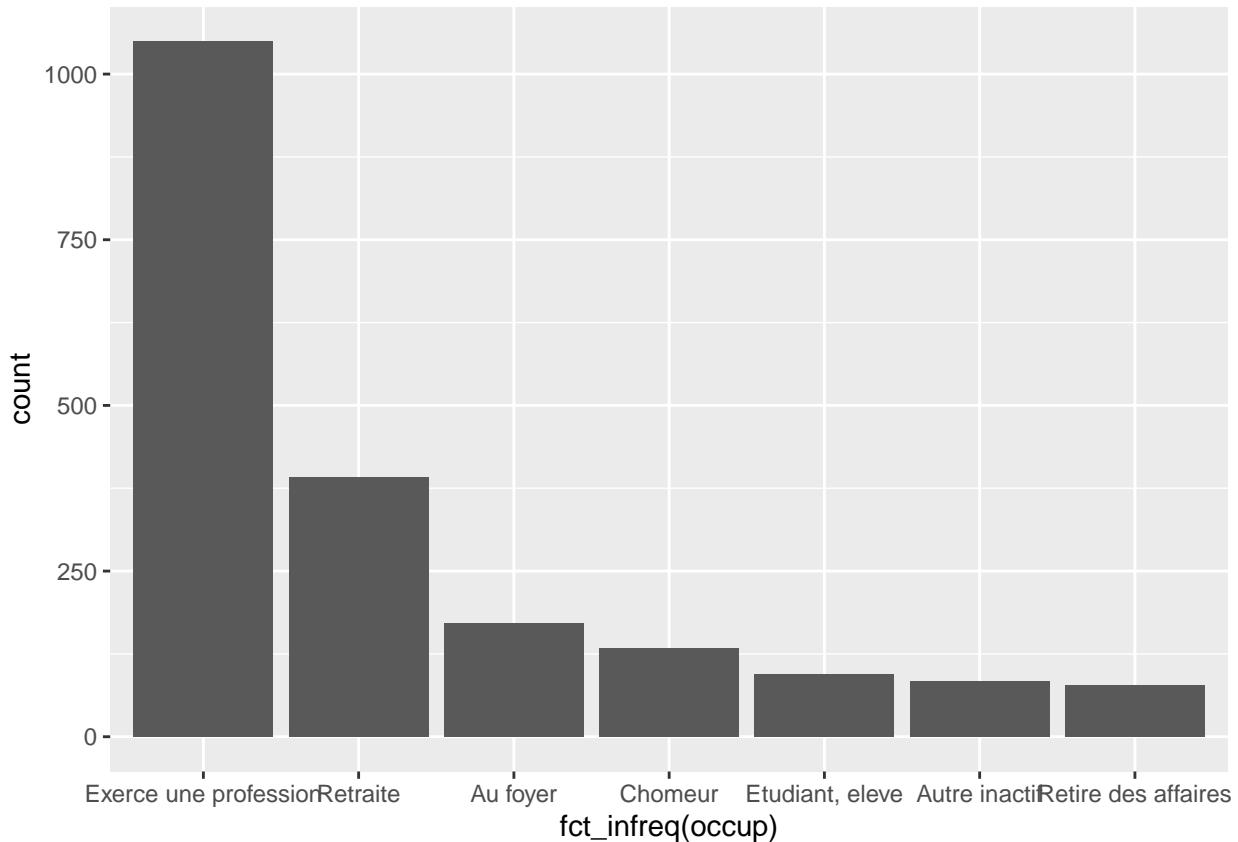


On peut aussi effectuer le réordonnancement directement dans l'appel à ggplot2, sans créer de nouvelle variable.

```
ggplot(hdv2003) +
  geom_boxplot(
    aes(
      x = fct_reorder(occup, age, median),
      y = age
    )
  )
```

Lorsqu'on effectue un diagramme en barres avec `geom_bar`, on peut aussi réordonner les modalités selon leurs effectifs à l'aide de `fct_infreq`.

```
ggplot(hdv2003) +
  geom_bar(aes(x = fct_infreq(occup)))
```



9.3.3.1 Interface graphique

`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 `iorder()` en lui passant comme paramètre le facteur à réordonner.

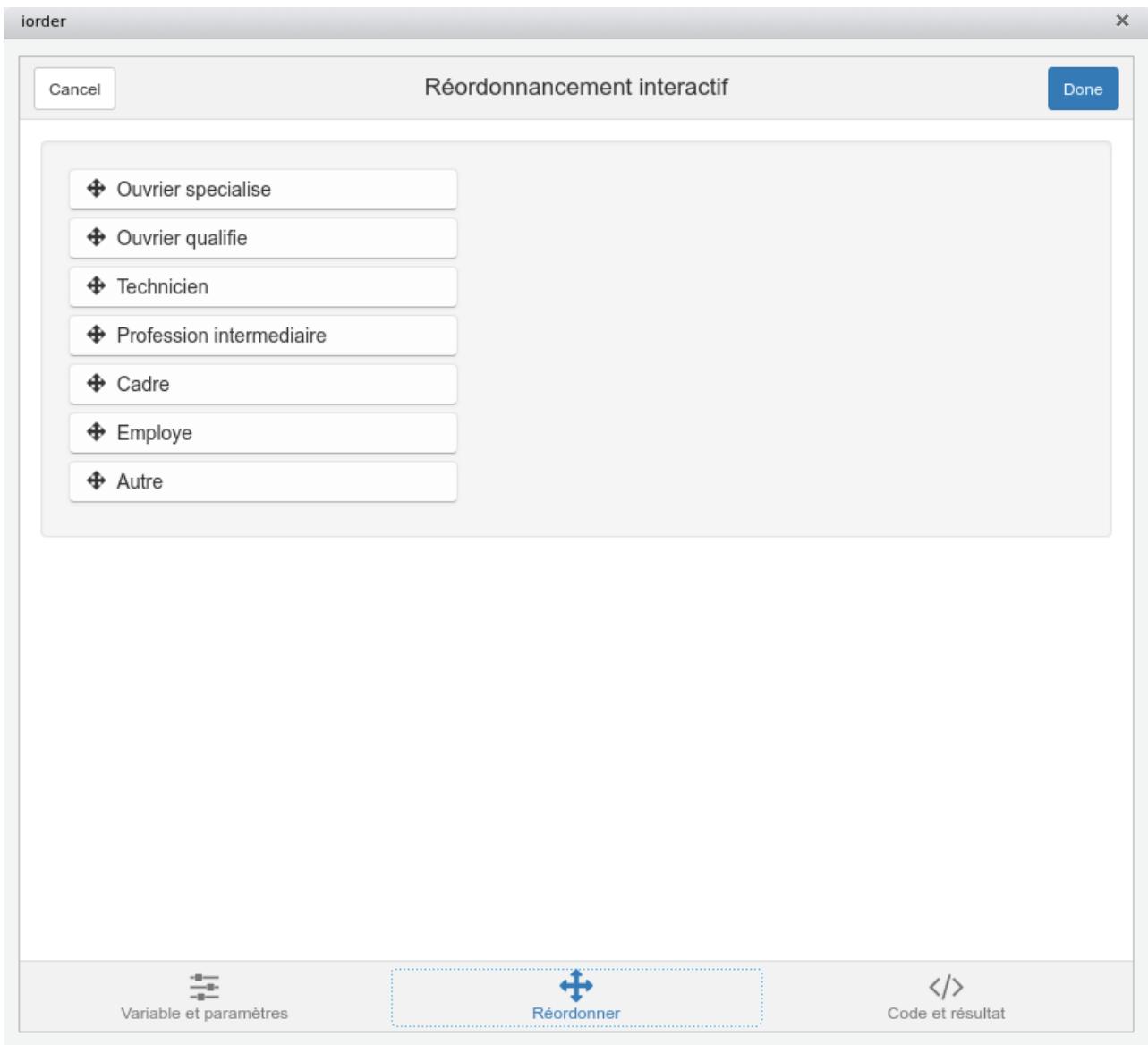


Figure 9.2: Interface graphique de `iorder`

Le fonctionnement de l'interface est similaire à celui de [l'interface de recodage](#). Vous pouvez réordonner les modalités en les faisant glisser avec la souris, puis récupérer et exécuter le code R généré.

9.4 Combiner plusieurs variables

Parfois, on veut créer une nouvelle variable en partant des valeurs d'une ou plusieurs autres variables. Dans ce cas on peut utiliser les fonctions `ifelse` pour les cas les plus simples, ou `case_when` pour les cas plus complexes. Cette dernière fonction est incluse dans l'extension `dplyr`, qu'il faut donc avoir chargé précédemment.

9.4.1 `ifelse`

`ifelse` prend trois arguments : un test, une valeur à renvoyer si le test est vrai, et une valeur à renvoyer si le test est faux.

Voici un exemple simple :

```
v <- c(12, 14, 8, 16)
ifelse(v > 10, "Supérieur à 10", "Inférieur à 10")
#> [1] "Supérieur à 10" "Supérieur à 10" "Inférieur à 10" "Supérieur à 10"
```

La fonction permet d'utiliser des tests combinant plusieurs variables. Par exemple, imaginons qu'on souhaite créer une nouvelle variable indiquant les hommes de plus de 60 ans :

```
hdv2003$statut <- ifelse(
  hdv2003$sex == "Homme" & hdv2003$age > 60,
  "Homme de plus de 60 ans",
  "Autre"
)

freq(hdv2003$statut)
#>           n   % val%
#> Autre      1778 88.9 88.9
#> Homme de plus de 60 ans 222 11.1 11.1
```

9.4.2 case_when

`case_when` est une généralisation du `ifelse` qui permet d'indiquer plusieurs tests et leurs valeurs associées.

Imaginons qu'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$statut <- case_when(
  hdv2003$age > 60 & hdv2003$sex == "Homme" ~ "Homme de plus de 60 ans",
  hdv2003$age > 60 & hdv2003$sex == "Femme" ~ "Femme de plus de 60 ans",
  TRUE ~ "Autre"
)

freq(hdv2003$statut)
#>           n   % val%
#> Autre      1512 75.6 75.6
#> Femme de plus de 60 ans 266 13.3 13.3
#> Homme de plus de 60 ans 222 11.1 11.1
```

`case_when` prend en arguments une série d'instructions sous la forme `condition ~ valeur`. Il les exécute une par une, et dès qu'une `condition` est vraie, il renvoie la `valeur` associée.

La dernière clause `TRUE ~ "Autre"` permet d'assigner une valeur à toutes les lignes pour lesquelles aucune des conditions précédentes n'est vraie.



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.

Pour illustrer cet avertissement, on pourra noter que le recodage suivant ne fonctionne pas :

```
hdv2003$statut <- case_when(
  hdv2003$sexe == "Homme" ~ "Homme",
  hdv2003$sexe == "Homme" & hdv2003$age > 60 ~ "Homme de plus de 60 ans",
  TRUE ~ "Autre"
)

freq(hdv2003$statut)
#>           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, et on n'obtiendra donc 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$statut <- case_when(
  hdv2003$sexe == "Homme" & hdv2003$age > 60 ~ "Homme de plus de 60 ans",
  hdv2003$sexe == "Homme" ~ "Homme",
  TRUE ~ "Autre"
)

freq(hdv2003$statut)
#>           n   % val%
#> Autre    1101 55.0 55.0
#> Homme   677 33.9 33.9
#> Homme de plus de 60 ans 222 11.1 11.1
```

9.5 Découper une variable numérique en classes

Une autre opération courante consiste à découper une variable numérique en classes. Par exemple, on voudra transformer une variable `revenu` contenant le revenu mensuel en une variable qualitative avec des catégories *Moins de 500 euros*, *501-1000 euros*, etc.

On utilise pour cela la fonction `cut()` :

```
hdv2003$agecl <- cut(hdv2003$age, breaks = 5)

freq(hdv2003$agecl)
#>           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
```

Si on donne un nombre entier à l'argument `breaks`, un nombre correspondant de classes d'amplitudes égales sont automatiquement calculées. Comme il est souvent préférable d'avoir des limites “arrondies”, on peut aussi spécifier ces dernières manuellement en passant un vecteur à `breaks`.

```

hdv2003$agecl <- cut(
  hdv2003$age,
  breaks = c(18, 25, 35, 45, 55, 65, 97),
  include.lowest = TRUE
)

freq(hdv2003$agecl)
#>      n    % val%
#> [18,25] 191  9.6  9.6
#> (25,35] 338 16.9 16.9
#> (35,45] 390 19.5 19.5
#> (45,55] 414 20.7 20.7
#> (55,65] 305 15.2 15.2
#> (65,97] 362 18.1 18.1

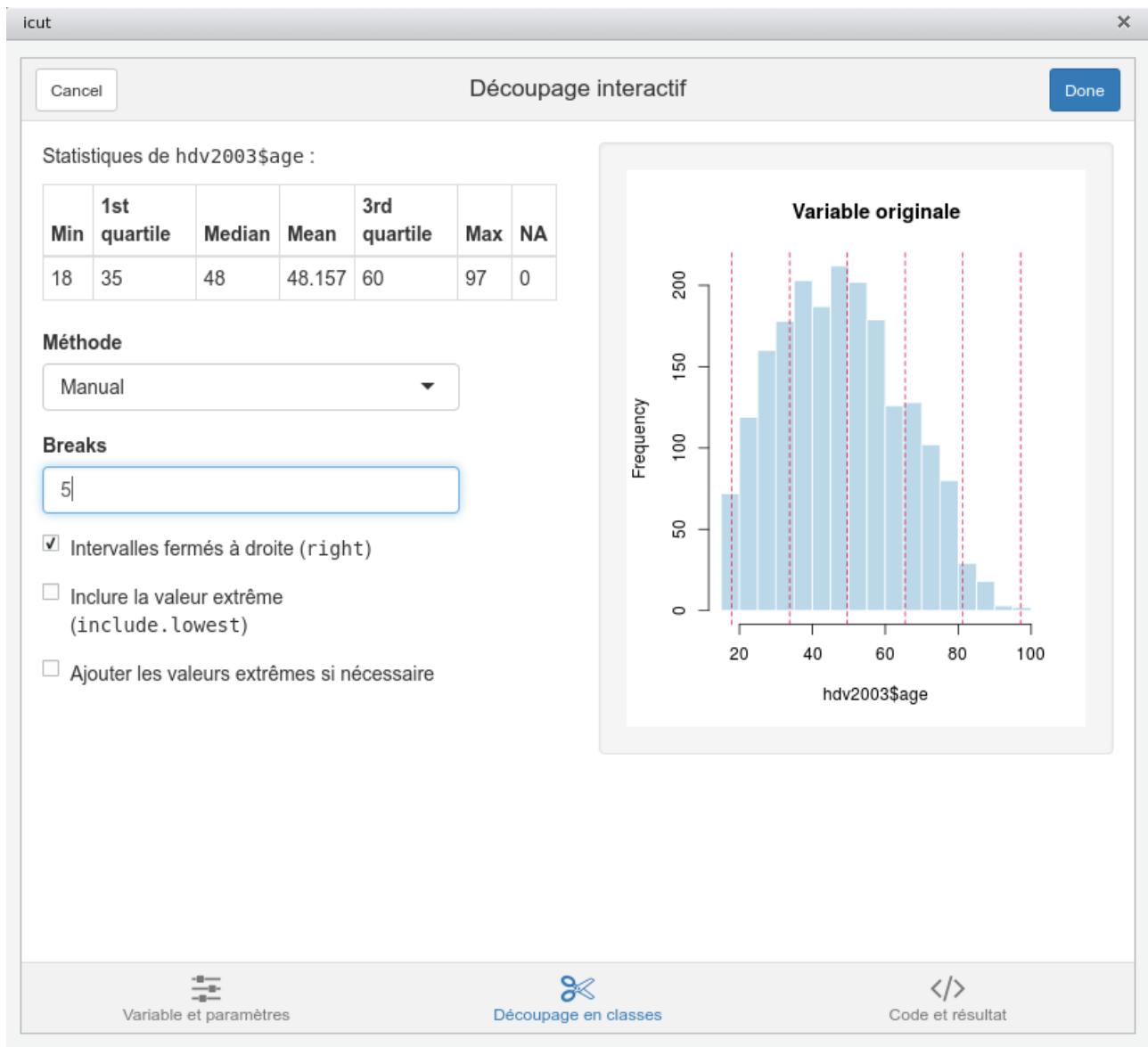
```

Ici on a été obligé d'ajouter l'argument `include.lowest = TRUE` car sinon la valeur 18 n'aurait pas été incluse, et on aurait eu des valeurs manquantes.

9.5.1 Interface graphique

Comme l'utilisation des arguments de `cut` n'est pas toujours très intuitive, l'extension `questionr` propose une interface graphique facilitant cette opération de découpage en classes d'une variable numérique.

Pour lancer cette interface, sous RStudio ouvrir le menu *Addins* et sélectionner *Numeric range dividing*, ou exécuter la fonction `icut()` dans la console en lui passant comme argument la variable quantitative à découper.

Figure 9.3: Interface graphique de *icut*

Vous pouvez alors choisir la variable à découper dans l'onglet *Variable et paramètres*, indiquer les limites de vos classes ainsi que quelques options complémentaires dans l'onglet *Découpage en classes*, et vérifier le résultat dans l'onglet *Code et résultat*. Une fois le résultat satisfaisant, cliquez sur *Done* : si vous êtes sous RStudio le code généré sera directement inséré dans votre script actuel à l'emplacement du curseur. Sinon, ce code sera affiché dans la console et vous pourrez le copier/coller pour l'inclure dans votre script.



Attention, cette interface est prévue pour ne pas modifier vos données. C'est donc à vous d'exécuter le code généré pour que le découpage soit réellement effectif.

9.6 Exercices

9.6.1 Préparation

Pour la plupart de ces exercices, on a besoin des extensions `forcats` et `questionr`, et du jeu de données d'exemple `hdv2003`.

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

9.6.2 Vecteurs et tests

Exercice 1.1

Construire le vecteur suivant :

```
x <- c("12", "3.5", "421", "2,4")
```

Et le convertir en vecteur numérique. Que remarquez-vous ?

Exercice 1.2

Construire le vecteur suivant :

```
x <- c(1, 20, 21, 15.5, 14, 12, 8)
```

- Écrire le test qui indique si les éléments du vecteur sont strictement supérieurs à 15.
- Utiliser ce test pour extraire du vecteur les éléments correspondants.

Exercice 1.3

Le code suivant génère un vecteur de 1000 nombres aléatoires compris entre 0 et 10 :

```
x <- runif(1000, 0, 10)
```

Combien d'éléments de ce vecteur sont compris entre 2 et 4 ?

9.6.3 Recodages de variable qualitative

Exercice 2.1

Construire un vecteur **f** à l'aide du code suivant :

```
f <- c("Jan", "Jan", "Fev", "Juil")
```

Recoder le vecteur à l'aide de la fonction **fct_recode** pour obtenir le résultat suivant :

```
#> [1] Janvier Janvier Février Juillet
#> Levels: Février Janvier Juillet
```

Exercice 2.2

À l'aide de l'interface graphique de **questionr**, recoder la variable **relig** du jeu de données **hdv2003** pour obtenir le tri à plat suivant (il se peut que l'ordre des modalités dans le tri à plat soit différent) :

```
#>          n   % val%
#> Pratiquant      708 35.4 35.4
#> Appartenance     760 38.0 38.0
#> Ni croyance ni appartenance 399 20.0 20.0
#> Rejet            93  4.7  4.7
#> NSP              40  2.0  2.0
```

Exercice 2.3

À l'aide de l'interface graphique de `questionr`, recoder la variable `nivetud` pour obtenir le tri à plat suivant (il se peut que l'ordre des modalités dans le tri à plat soit différent) :

```
#>          n   % val%
#> N'a jamais fait d'etudes      39  2.0  2.1
#> Études primaires             427 21.3 22.6
#> 1er cycle                     204 10.2 10.8
#> 2eme cycle                   183  9.2  9.7
#> Enseignement technique ou professionnel 594 29.7 31.5
#> Enseignement superieur        441 22.0 23.4
#> NA                            112  5.6  NA
```

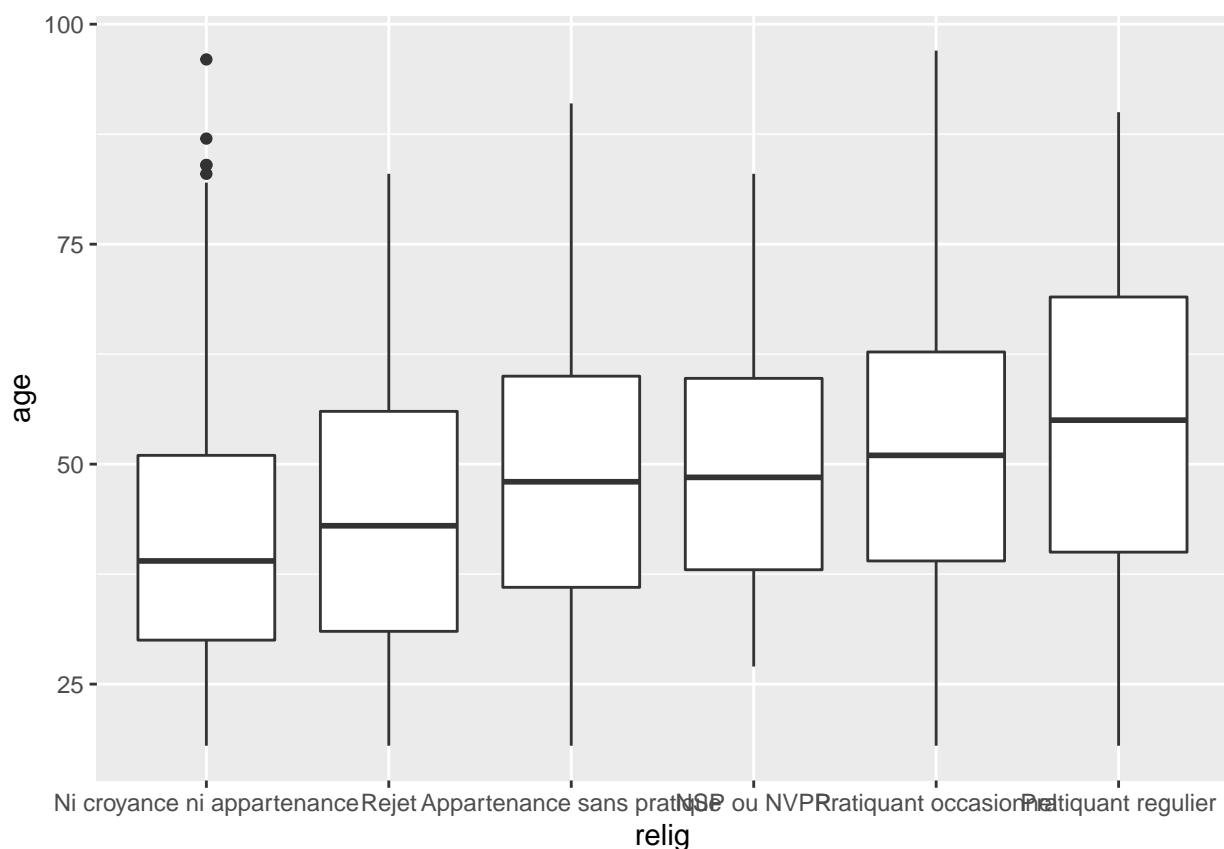
Toujours à l'aide de l'interface graphique, réordonner les modalités de cette variable recodée pour obtenir le tri à plat suivant :

```
#>          n   % val%
#> Enseignement superieur        441 22.0 23.4
#> Enseignement technique ou professionnel 594 29.7 31.5
#> 2eme cycle                   183  9.2  9.7
#> 1er cycle                     204 10.2 10.8
#> Études primaires             427 21.3 22.6
#> N'a jamais fait d'etudes      39  2.0  2.1
#> NA                            112  5.6  NA
```

Exercice 2.4

À l'aide de la fonction `fct_reorder`, trier les modalités de la variable `relig` du jeu de données `hdv2003` selon leur âge médian.

Vérifier en générant le boxplot suivant :



9.6.4 Combiner plusieurs variables

Exercice 3.1

À l'aide de la fonction `ifelse`, créer une nouvelle variable `cinema_bd` permettant d'identifier les personnes qui vont au cinéma et déclarent lire des bandes dessinées.

Vous devriez obtenir le tri à plat suivant pour cette nouvelle variable :

```
#>          n    % val%
#> Autre      1971 98.6 98.6
#> Cinéma et BD   29  1.5  1.5
```

Exercice 3.2

À l'aide de la fonction `case_when`, créer une nouvelle variable ayant les modalités suivantes :

- Homme ayant plus de 2 frères et soeurs
- Femme ayant plus de 2 frères et soeurs
- Autre

Vous devriez obtenir le tri à plat suivant :

```
#>          n    % val%
#> Autre      1001 50.0 50.0
#> Femme ayant plus de 2 frères et soeurs  546 27.3 27.3
#> Homme ayant plus de 2 frères et soeurs  453 22.7 22.7
```

Exercice 3.3

À l'aide de la fonction `case_when`, créer une nouvelle variable ayant les modalités suivantes :

- Homme de plus de 30 ans
- Homme de plus de 40 ans satisfait par son travail
- Femme pratiquant le sport ou le bricolage
- Autre

Vous devriez obtenir le tri à plat suivant :

```
#>           n   % val%
#> Autre          714 35.7 35.7
#> Femme pratiquant le sport ou le bricolage 549 27.5 27.5
#> Homme de plus de 30 ans            610 30.5 30.5
#> Homme de plus de 40 ans satisfait par son travail 127  6.3  6.3
```

9.6.5 Découper une variable numérique**Exercice 4.1**

Dans le jeu de données `hdv2003`, découper la variable `heures.tv` en classes de manière à obtenir au final le tri à plat suivant :

```
#>           n   % val%
#> [0,1]    684 34.2 34.3
#> (1,2]    535 26.8 26.8
#> (2,4]    594 29.7 29.8
#> (4,6]    138  6.9  6.9
#> (6,12]   44  2.2  2.2
#> NA        5  0.2   NA
```


Chapitre 10

Manipuler les données avec `dplyr`

`dplyr` est une extension facilitant le traitement et la manipulation de données contenues dans une ou plusieurs tables. Elle propose une syntaxe claire et cohérente, sous formes de verbes, pour la plupart des opérations de ce type.

`dplyr` part du principe que les données sont organisées selon le modèle des *tidy data* (voir la section 6.3). Les fonctions de l'extension peuvent s'appliquer à des tableaux de type `data.frame` ou `tibble`, et elles retournent systématiquement un `tibble` (voir la section 6.4).

Le code présent dans ce document nécessite d'avoir installé la version 1.0 de `dplyr` (ou plus récente).

10.1 Préparation

`dplyr` fait partie du cœur du *tidyverse*, elle est donc chargée automatiquement avec :

```
library(tidyverse)
```

On peut également la charger individuellement.

```
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 installé). 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 :

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

On va charger les trois tables du jeu de données :

```
library(nycflights13)
## Chargement des trois tables
data(flights)
data(airports)
data(airlines)
```

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

10.2 Les verbes de `dplyr`

La manipulation de données avec `dplyr` se fait en utilisant un nombre réduit de verbes, qui correspondent chacun à une action différente appliquée à un tableau de données.

10.2.1 `slice`

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

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

```
slice(airports, 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 on veut sélectionner les 5 premières lignes :

```
slice(airports, 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-
```

10.2.2 `filter`

`filter` sélectionne des lignes d'une table selon une condition. On lui passe en paramètre un test, et seules les lignes pour lesquelles ce test renvoie `TRUE` (vrai) sont conservées. Pour plus d'informations sur les tests et leur syntaxe, voir la section 9.2.

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

```
filter(flights, month == 1)
#> # A tibble: 27,004 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>           <int>    <dbl>    <int>           <int>
#> 1 2013     1     1     517       515         2     830         819
#> 2 2013     1     1     533       529         4     850         830
#> 3 2013     1     1     542       540         2     923         850
#> 4 2013     1     1     544       545        -1    1004        1022
#> 5 2013     1     1     554       600        -6    812         837
#> 6 2013     1     1     554       558        -4    740         728
#> 7 2013     1     1     555       600        -5    913         854
#> 8 2013     1     1     557       600        -3    709         723
#> 9 2013     1     1     557       600        -3    838         846
#> 10 2013    1     1     558       600        -2    753         745
```

```
#> # ... with 26,994 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
filter(flights, dep_delay >= 10 & dep_delay <= 15)
#> # A tibble: 14,919 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <int>     <dbl>    <int>        <int>
#> 1 2013     1     1      611       600      11     945        931
#> 2 2013     1     1      623       610      13     920        915
#> 3 2013     1     1      743       730      13    1107       1100
#> 4 2013     1     1      743       730      13    1059       1056
#> 5 2013     1     1      851       840      11    1215       1206
#> 6 2013     1     1      912       900      12    1241       1220
#> 7 2013     1     1      914       900      14    1058       1043
#> 8 2013     1     1      920       905      15    1039       1025
#> 9 2013     1     1     1011      1001      10    1133       1128
#> 10 2013    1     1     1112      1100      12    1440       1438
#> # ... with 14,909 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Si on passe plusieurs arguments à `filter`, celui-ci rajoute automatiquement une condition *et* entre les conditions. La commande précédente peut donc être écrite de la manière suivante, avec le même résultat :

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

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

```
filter(flights, distance == max(distance))
#> # A tibble: 342 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <int>     <dbl>    <int>        <int>
#> 1 2013     1     1      857       900      -3     1516       1530
#> 2 2013     1     2      909       900       9     1525       1530
#> 3 2013     1     3      914       900      14     1504       1530
#> 4 2013     1     4      900       900       0     1516       1530
#> 5 2013     1     5      858       900      -2     1519       1530
#> 6 2013     1     6     1019       900      79     1558       1530
#> 7 2013     1     7     1042       900     102     1620       1530
#> 8 2013     1     8      901       900       1     1504       1530
#> 9 2013     1     9      641       900     1301     1242       1530
#> 10 2013    1    10      859       900      -1     1449       1530
#> # ... with 332 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

10.2.3 `select` et `rename`

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

```
select(airports, 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 précédé le nom d'un `-`, la colonne est éliminée plutôt que sélectionnée :

```
select(airports, -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
```

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

```
select(flights, 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` incluses¹.

```
select(flights, year: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
```

`select` propose de nombreuses autres possibilités de sélection qui sont décrites dans [la documentation de l'extension `tidyselect`](#).

Une variante de `select` est `rename`², qui permet de renommer 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 :

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

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

```
tmp <- rename(
  flights,
  "retard départ" = dep_delay,
  "retard arrivée" = arr_delay
)
select(tmp, `retard départ`, `retard arrivée`)
#> # A tibble: 336,776 x 2
```

¹À 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.

²Il est également possible de renommer des colonnes directement avec `select`, avec la même syntaxe que pour `rename`.

```
#> `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
```

10.2.4 arrange

`arrange` réordonne les lignes d'un tableau selon une ou plusieurs colonnes.

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

```
arrange(flights, dep_delay)  

#> # A tibble: 336,776 x 19  

#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time  

#>   <int> <int> <int> <int> <int> <dbl> <int> <int>  

#> 1 2013 12 7 2040 2123 -43 40 2352  

#> 2 2013 2 3 2022 2055 -33 2240 2338  

#> 3 2013 11 10 1408 1440 -32 1549 1559  

#> 4 2013 1 11 1900 1930 -30 2233 2243  

#> 5 2013 1 29 1703 1730 -27 1947 1957  

#> 6 2013 8 9 729 755 -26 1002 955  

#> 7 2013 10 23 1907 1932 -25 2143 2143  

#> 8 2013 3 30 2030 2055 -25 2213 2250  

#> 9 2013 3 2 1431 1455 -24 1601 1631  

#> 10 2013 5 5 934 958 -24 1225 1309  

#> # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,  

#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,  

#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
arrange(flights, month, dep_delay)  

#> # A tibble: 336,776 x 19  

#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time  

#>   <int> <int> <int> <int> <int> <dbl> <int> <int>  

#> 1 2013 1 11 1900 1930 -30 2233 2243  

#> 2 2013 1 29 1703 1730 -27 1947 1957  

#> 3 2013 1 12 1354 1416 -22 1606 1650  

#> 4 2013 1 21 2137 2159 -22 2232 2316  

#> 5 2013 1 20 704 725 -21 1025 1035  

#> 6 2013 1 12 2050 2110 -20 2310 2355  

#> 7 2013 1 12 2134 2154 -20 4 50  

#> 8 2013 1 14 2050 2110 -20 2329 2355  

#> 9 2013 1 4 2140 2159 -19 2241 2316  

#> 10 2013 1 11 1947 2005 -18 2209 2230  

#> # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
```

```
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
arrange(flights, desc(dep_delay))
#> # A tibble: 336,776 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>           <int>     <dbl>    <int>           <int>
#> 1 2013     1     9      641        900 1301 1242 1530
#> 2 2013     6    15     1432       1935 1137 1607 2120
#> 3 2013     1    10     1121       1635 1126 1239 1810
#> 4 2013     9    20     1139       1845 1014 1457 2210
#> 5 2013     7    22      845       1600 1005 1044 1815
#> 6 2013     4    10     1100       1900  960 1342 2211
#> 7 2013     3    17     2321       810   911 135 1020
#> 8 2013     6    27      959       1900  899 1236 2226
#> 9 2013     7    22     2257       759   898 121 1026
#> 10 2013    12     5      756       1700  896 1058 2020
#> # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

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

```
tmp <- arrange(flights, desc(dep_delay))
slice(tmp, 1:3)
#> # A tibble: 3 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>           <int>     <dbl>    <int>           <int>
#> 1 2013     1     9      641        900 1301 1242 1530
#> 2 2013     6    15     1432       1935 1137 1607 2120
#> 3 2013     1    10     1121       1635 1126 1239 1810
#> # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

10.2.5 `mutate`

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

Par exemple, la table `flights` contient la durée du vol en minutes.. Si on veut créer une nouvelle variable `duree_h` avec cette durée en heures, on peut faire :

```
flights <- mutate(flights, duree_h = air_time / 60)

select(flights, air_time, duree_h)
#> # A tibble: 336,776 x 2
#>   air_time duree_h
#>   <dbl>    <dbl>
#> 1      227  3.78
#> 2      227  3.78
```

```
#> 3      160  2.67
#> 4      183  3.05
#> 5      116  1.93
#> 6      150  2.5
#> 7      158  2.63
#> 8      53   0.883
#> 9      140  2.33
#> 10     138  2.3
#> # ... with 336,766 more rows
```

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

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

select(flights, air_time, duree_h, distance, distance_km, vitesse)
#> # A tibble: 336,776 x 5
#>   air_time    duree_h  distance  distance_km  vitesse
#>   <dbl>     <dbl>     <dbl>      <dbl>     <dbl>
#> 1     227     3.78     1400     2253.     596.
#> 2     227     3.78     1416     2279.     602.
#> 3     160     2.67     1089     1753.     657.
#> 4     183     3.05     1576     2536.     832.
#> 5     116     1.93      762     1226.     634.
#> 6     150     2.5       719     1157.     463.
#> 7     158     2.63     1065     1714.     651.
#> 8      53     0.883     229      369.     417.
#> 9     140     2.33     944     1519.     651.
#> 10    138     2.3      733     1180.     513.
#> # ... with 336,766 more rows
```

À noter que `mutate` est évidemment parfaitement compatible avec les fonctions vues dans le chapitre 9 sur les recodages : `fct_recode`, `ifelse`, `case_when`...

L'avantage d'utiliser `mutate` est double. D'abord il permet d'éviter d'avoir à saisir le nom du tableau de données dans les conditions d'un `ifelse` ou d'un `case_when` :

```
flights <- mutate(
  flights,
  type_retard = case_when(
    dep_delay > 0 & arr_delay > 0 ~ "Retard départ et arrivée",
    dep_delay > 0 & arr_delay <= 0 ~ "Retard départ",
    dep_delay <= 0 & arr_delay > 0 ~ "Retard arrivée",
    TRUE ~ "Aucun retard"
  )
)
```

Ensuite, il permet aussi d'intégrer ces recodages dans un *pipeline* de traitement de données, concept présenté dans la section suivante.

10.3 Enchaîner les opérations avec le *pipe*

Quand on manipule un tableau de données, il est très fréquent d'enchaîner plusieurs opérations. On va par exemple extraire une sous-population avec `filter`, sélectionner des colonnes avec `select` puis trier selon une variable avec `arrange`, etc.

Quand on veut enchaîner des opérations, on peut le faire de différentes manières. La première est d'effectuer toutes les opérations en une fois en les “emboîtant” :

```
arrange(select(filter(flights, dest == "LAX"), dep_delay, arr_delay), dep_delay)
```

Cette notation a plusieurs inconvénients :

- elle est peu lisible
- les opérations apparaissent dans l'ordre inverse de leur réalisation. Ici on effectue d'abord le `filter`, puis le `select`, puis le `arrange`, alors qu'à la lecture du code c'est le `arrange` qui apparaît en premier.
- Il est difficile de voir quel paramètre se rapporte à quelle fonction

Une autre manière de faire est d'effectuer les opérations les unes après les autres, en stockant les résultats intermédiaires dans un objet temporaire :

```
tmp <- filter(flights, dest == "LAX")
tmp <- select(tmp, dep_delay, arr_delay)
arrange(tmp, dep_delay)
```

C'est nettement plus lisible, l'ordre des opérations est le bon, et les paramètres sont bien rattachés à leur fonction. Par contre, ça reste un peu “verbeux”, et on crée un objet temporaire `tmp` dont on n'a pas réellement besoin.

Pour simplifier et améliorer encore la lisibilité du code, on va utiliser un nouvel opérateur, baptisé *pipe*³. Le *pipe* se note `%>%`, et son fonctionnement est le suivant : si j'exécute `expr %>% f`, alors le résultat de l'expression `expr`, à gauche du *pipe*, sera passé comme premier argument à la fonction `f`, à droite du *pipe*, ce qui revient à exécuter `f(expr)`.

Ainsi les deux expressions suivantes sont rigoureusement équivalentes :

```
filter(flights, dest == "LAX")
```

```
flights %>% filter(dest == "LAX")
```

Ce qui est particulièrement intéressant, c'est qu'on va pouvoir enchaîner les *pipes*. Plutôt que d'écrire :

```
select(filter(flights, dest == "LAX"), dep_delay, arr_delay)
```

On va pouvoir faire :

³Le *pipe* a été introduit à l'origine par l'extension `magrittr`, et repris par `dplyr`

```
flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay)
```

À chaque fois, le résultat de ce qui se trouve à gauche du *pipe* est passé comme premier argument à ce qui se trouve à droite : on part de l'objet `flights`, qu'on passe comme premier argument à la fonction `filter`, puis on passe le résultat de ce `filter` comme premier argument du `select`.

Le résultat final est le même avec les deux syntaxes, mais avec le *pipe* l'ordre des opérations correspond à l'ordre naturel de leur exécution, et on n'a pas eu besoin de créer d'objet intermédiaire.

Si la liste des fonctions enchaînées est longue, on peut les répartir sur plusieurs lignes à condition que l'opérateur `%>%` soit en fin de ligne :

```
flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

 On appelle une suite d'instructions de ce type un *pipeline*.

Évidemment, il est naturel de vouloir récupérer le résultat final d'un *pipeline* pour le stocker dans un objet. On peut stocker le résultat du *pipeline* ci-dessus dans un nouveau tableau `delay_la` de la manière suivante :

```
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay) %>%
  arrange(dep_delay)
```

Dans ce cas, `delay_la` contiendra le tableau final, obtenu après application des trois instructions `filter`, `select` et `arrange`.

Cette notation n'est pas forcément très intuitive au départ : il faut bien comprendre que c'est le résultat final, une fois application de toutes les opérations du *pipeline*, qui est renvoyé et stocké dans l'objet en début de ligne.

Une manière de le comprendre peut être de voir que la notation suivante :

```
delay_la <- flights %>%
  filter(dest == "LAX") %>%
  select(dep_delay, arr_delay)
```

est équivalente à :

```
delay_la <- (flights %>% filter(dest == "LAX") %>% select(dep_delay, arr_delay))
```

 L'utilisation du *pipe* n'est pas obligatoire, mais elle rend les scripts plus lisibles et plus rapides à saisir. On l'utilisera donc dans ce qui suit.



Depuis la version 4.1, R propose un *pipe* “natif”, qui fonctionne partout, même si on n’utilise pas les extensions du *tidyverse*. Celui-ci est noté |>.

Il s’utilise de la même manière que %>% :

```
flights |> filter(dest == "LAX")
```

Ce *pipe* natif est à la fois un peu plus rapide et un peu moins souple. Par exemple, il est possible avec %>% d’appeler une fonction sans mettre de parenthèses :

```
df %>% View
```

Ce n’est pas possible d’omettre les parenthèses avec |>, on doit obligatoirement faire :

```
df |> View()
```

Dans la suite de ce document on privilégiera (pour l’instant) le *pipe* du *tidyverse* %>% , pour des raisons de compatibilité avec des versions de R moins récentes.

10.4 Opérations groupées

10.4.1 group_by

Un élément très important de **dplyr** est la fonction **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 22
#> # Groups:   month [12]
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <dbl>    <int>      <dbl>
#> 1 2013     1     1     517       515        2     830       819
#> 2 2013     1     1     533       529        4     850       830
#> 3 2013     1     1     542       540        2     923       850
#> 4 2013     1     1     544       545       -1    1004      1022
#> 5 2013     1     1     554       600       -6     812       837
#> 6 2013     1     1     554       558       -4     740       728
#> 7 2013     1     1     555       600       -5     913       854
#> 8 2013     1     1     557       600       -3     709       723
#> 9 2013     1     1     557       600       -3     838       846
#> 10 2013    1     1     558       600       -2     753       745
#> # ... with 336,766 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>
```

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 **slice**, **mutate** ou **summarise** vont en tenir compte lors de leurs opérations.

Par exemple, si on applique **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 22
#> # Groups:   month [12]
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
```

```
#>      <int> <int> <int> <int>      <int>    <dbl> <int>      <int>
#> 1 2013     1     1   517      515      2    830     819
#> 2 2013     2     1   456      500     -4    652     648
#> 3 2013     3     1     4   2159     125    318      56
#> 4 2013     4     1   454      500     -6    636     640
#> 5 2013     5     1     9   1655     434    308    2020
#> 6 2013     6     1     2   2359      3    341     350
#> 7 2013     7     1     1   2029     212    236    2359
#> 8 2013     8     1    12   2130     162    257      14
#> 9 2013     9     1     9   2359     10    343     340
#> 10 2013    10     1   447      500    -13    614     648
#> 11 2013    11     1     5   2359      6    352     345
#> 12 2013    12     1   13   2359     14    446     445
#> # ... with 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> # tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> # hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> # distance_km <dbl>, vitesse <dbl>
```

Idem pour `mutate` : les opérations appliquées lors du calcul des valeurs des nouvelles colonnes sont appliquées groupe de lignes par groupe de lignes. Dans l'exemple suivant, on ajoute une nouvelle colonne qui contient le retard moyen *pour chaque compagnie aérienne*. Cette valeur est donc différente d'une compagnie à une autre, mais identique pour tous les vols d'une même compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(mean_delay_carrier = mean(dep_delay, na.rm = TRUE)) %>%
  select(dep_delay, mean_delay_carrier)
#> Adding missing grouping variables: `carrier`
#> # A tibble: 336,776 x 3
#> # Groups:   carrier [16]
#>   carrier dep_delay mean_delay_carrier
#>   <chr>        <dbl>            <dbl>
#> 1 UA             2            12.1
#> 2 UA             4            12.1
#> 3 AA             2            8.59
#> 4 B6            -1            13.0
#> 5 DL            -6            9.26
#> 6 UA            -4            12.1
#> 7 B6            -5            13.0
#> 8 EV            -3            20.0
#> 9 B6            -3            13.0
#> 10 AA            -2            8.59
#> # ... with 336,766 more rows
```

Ceci peut permettre, par exemple, de déterminer si un retard donné est supérieur ou inférieur au retard médian de la compagnie :

```
flights %>%
  group_by(carrier) %>%
  mutate(
    median_delay = median(dep_delay, na.rm = TRUE),
    delay_carrier = ifelse(
      dep_delay > median_delay,
      "Supérieur",
      "Inférieur ou égal"
```

```

    )
) %>%
  select(dep_delay, median_delay, delay_carrier)
#> Adding missing grouping variables: `carrier`
#> # A tibble: 336,776 x 4
#> # Groups:   carrier [16]
#>   carrier dep_delay median_delay delay_carrier
#>   <chr>      <dbl>        <dbl> <chr>
#> 1 UA          2            0 Supérieur
#> 2 UA          4            0 Supérieur
#> 3 AA          2           -3 Supérieur
#> 4 B6         -1           -1 Inférieur ou égal
#> 5 DL          -6           -2 Inférieur ou égal
#> 6 UA          -4           0 Inférieur ou égal
#> 7 B6          -5           -1 Inférieur ou égal
#> 8 EV          -3           -1 Inférieur ou égal
#> 9 B6          -3           -1 Inférieur ou égal
#> 10 AA         -2           -3 Supérieur
#> # ... with 336,766 more rows

```

`group_by` peut aussi être utile avec `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 22
#> # Groups:   month [12]
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int> <int>           <int>     <dbl> <int>           <int>
#> 1 2013     1     9     641            900     1301     1242           1530
#> 2 2013     10    14    2042            900     702     2255           1127
#> 3 2013     11     3     603            1645     798     829            1913
#> 4 2013     12     5     756            1700     896     1058           2020
#> 5 2013     2     10    2243            830     853     100            1106
#> 6 2013     3     17    2321            810     911     135            1020
#> 7 2013     4     10    1100            1900     960     1342           2211
#> 8 2013     5     3    1133            2055     878     1250           2215
#> 9 2013     6     15    1432            1935     1137     1607           2120
#> 10 2013    7     22     845            1600     1005     1044           1815
#> 11 2013    8     8    2334            1454     520      120            1710
#> 12 2013    9     20    1139            1845     1014     1457           2210
#> # ... with 14 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>, duree_h <dbl>,
#> #   distance_km <dbl>, vitesse <dbl>

```



Attention : la clause `group_by` marche pour les verbes déjà vus précédemment, *sauf* pour `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 22
#> # Groups:   month [12]
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <int>    <dbl>    <int>        <int>
#> 1 2013     1     9      641         900    1301    1242        1530
#> 2 2013     6    15     1432        1935    1137    1607        2120
#> 3 2013     1    10     1121        1635    1126    1239        1810
#> 4 2013     9    20     1139        1845    1014    1457        2210
#> 5 2013     7    22      845        1600    1005    1044        1815
#> 6 2013     4    10     1100        1900     960    1342        2211
#> 7 2013     3    17     2321        810     911    135         1020
#> 8 2013     6    27      959        1900     899    1236        2226
#> 9 2013     7    22     2257        759     898    121         1026
#> 10 2013    12     5      756        1700     896    1058        2020
#> # ... with 336,766 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>

```

```

flights %>%
  group_by(month) %>%
  arrange(desc(dep_delay), .by_group = TRUE)
#> # A tibble: 336,776 x 22
#> # Groups:   month [12]
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <int>    <dbl>    <int>        <int>
#> 1 2013     1     9      641         900    1301    1242        1530
#> 2 2013     1    10     1121        1635    1126    1239        1810
#> 3 2013     1     1      848        1835     853    1001        1950
#> 4 2013     1    13     1809        810     599    2054        1042
#> 5 2013     1    16     1622        800     502    1911        1054
#> 6 2013     1    23     1551        753     478    1812        1006
#> 7 2013     1    10     1525        900     385    1713        1039
#> 8 2013     1     1     2343        1724     379     314        1938
#> 9 2013     1     2     2131        1512     379    2340        1741
#> 10 2013    1     7     2021        1415     366    2332        1724
#> # ... with 336,766 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>

```

10.4.2 summarise et count

`summarise` permet d'agrégier les lignes du tableau en effectuant une opération “résumée” sur une ou plusieurs colonnes. Par exemple, si 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 `group_by`, puisqu'elle permet du coup d'agréger et résumer les lignes du tableau groupe par groupe. Si 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

```

`summarise` dispose d'un opérateur spécial, `n()`, qui retourne le nombre de lignes du groupe. Ainsi si on veut le nombre de vols par destination, on peut utiliser :

```

flights %>%
  group_by(dest) %>%
  summarise(nb = n())
#> # A tibble: 105 x 2
#>   dest     nb
#>   <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

```

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

À noter que quand on veut compter le nombre de lignes par groupe, on peut utiliser directement la fonction `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
```

Lorsque la variable de groupage est un facteur et que certaines valeurs du facteur ne sont pas présentes dans le tableau, l'argument `.drop = FALSE` de `group_by` permet de conserver ces niveaux dans le résultat d'une opération groupée.

Par exemple, si on transforme la variable `origin` en facteur pour conserver la liste de ses modalités, et qu'on ne garde que les vols à destination de San Francisco (code SFO) :

```
ff <- flights %>%
  mutate(origin = factor(origin)) %>%
  filter(dest == "SFO")
```

Par défaut, si on compte le nombre de vols selon l'aéroport de départ, La Guardia n'apparaît pas car il ne compte aucun vol :

```
ff %>%
  group_by(origin) %>%
  summarise(n = n())
#> # A tibble: 2 x 2
#>   origin     n
#>   <fct> <int>
#> 1 EWR     5127
#> 2 JFK     8204
```

Si on souhaite faire apparaître cette information dans la sortie du `summarise`, on peut ajouter l'argument `.drop = FALSE` au `group_by` :

```
ff %>%
  group_by(origin, .drop = FALSE) %>%
  summarise(n = n())
#> # A tibble: 3 x 2
#>   origin     n
#>   <fct> <int>
#> 1 EWR     5127
#> 2 JFK     8204
#> 3 LGA       0
```

Cet argument fonctionne aussi avec `count` :

```
ff %>%
  count(origin, .drop = FALSE)
#> # A tibble: 3 x 2
#>   origin     n
#>   <fct> <int>
#> 1 EWR      5127
#> 2 JFK      8204
#> 3 LGA        0
```

10.4.3 Grouper selon plusieurs variables

On peut grouper selon plusieurs variables à la fois, il suffit de les indiquer dans la clause du `group_by`. Le pipeline suivant le nombre de vols pour chaque mois et pour chaque destination, et trie le résultat par nombre de vols décroissant :

```
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 utiliser `count` sur plusieurs variables. Les commandes suivantes comptent le nombre de vols pour chaque couple aéroport de départ / aéroport d'arrivée, et trie le résultat par nombre de vols décroissant. Ici la colonne qui contient le nombre de vols, créée par `count`, s'appelle `n` par défaut :

```
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 on souhaite déterminer le couple / aéroport de départ / aéroport d'arrivée ayant le plus grand nombre de vols selon le mois de l'année, on devra procéder en deux étapes :

- d'abord grouper selon mois, aéroports d'origine et d'arrivée pour calculer le nombre de vols
- puis grouper uniquement selon le mois pour sélectionner le mois 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 `group_by` suivi d'un `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` :

```
flights %>%
  group_by(month, origin, dest) %>%
  summarise(nb = n())
#> `summarise()` has grouped output by 'month', 'origin'. You can override using the `groups` argument
#> # A tibble: 2,313 x 4
#> # Groups: month, origin [36]
#>   month origin dest     nb
#>   <int> <chr> <chr> <int>
#> 1     1  EWR   ALB      64
#> 2     1  EWR   ATL     362
#> 3     1  EWR   AUS      51
#> 4     1  EWR   AVL       2
#> 5     1  EWR   BDL     37
```

```
#> 6   1 EWR    BNA    111
#> 7   1 EWR    BOS    430
#> 8   1 EWR    BQN    31
#> 9   1 EWR    BTV    100
#> 10  1 EWR    BUF    119
#> # ... with 2,303 more rows
```

R nous le signale d'ailleurs via un message d'avertissement : `summarise()` has grouped output by 'month', 'origin'.

Ce dégroupage progressif 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 "dégroupier" un tableau à l'aide de `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
```

On peut aussi spécifier précisément le comportement de dégroupage de `summarise` en lui fournissant un argument supplémentaire `.groups` qui peut prendre notamment les valeurs suivantes :

- "drop_last" : dégroupe seulement de la dernière variable de groupage
- "drop" : dégroupe totalement le tableau résultat (équivaut à l'application d'un `ungroup`)
- "keep" : conserve toutes les variables de groupage

Ce concept de dégroupage successif peut être un peu déroutant de prime abord. Il est donc utile de faire attention aux avertissements affichés par ces opérations, et il ne faut pas hésiter à ajouter un `ungroup` en fin de pipeline si on sait qu'on ne souhaite pas utiliser les groupes encore existants par la suite.

À noter que la fonction `count`, de son côté, renvoie un tableau non groupé :

```
flights %>%
  count(month, dest)
#> # A tibble: 1,113 x 3
#>   month dest     n
#>   <int> <chr> <int>
#> 1     1 ALB     64
#> 2     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
```

10.5 Autres fonctions utiles

`dplyr` contient beaucoup d'autres fonctions utiles pour la manipulation de données.

10.5.1 `slice_sample`

Ce verbe permet de sélectionner aléatoirement un nombre de lignes (avec l'argument `n`) ou une fraction des lignes (avec l'argument `prop`) d'un tableau.

Ainsi si 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 MMH  Mammoth Yosemite Airport  37.6 -119.   7128    -8 A  America/Los~
#> 2 CDB  Cold Bay            55.2 -163.     96    -9 A  America/Anc~
#> 3 FMY  Page Fld           26.6 -81.9     17    -5 A  America/New~
#> 4 CKF  Crisp County Cordele Airport 32.0 -83.8    310    -5 A  America/New~
#> 5 NNL  Nondalton Airport    60.0 -155.   262    -9 A  America/Anc~
```

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

```
flights %>% slice_sample(prop = 0.1)
#> # A tibble: 33,677 x 22
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>      <int>
#> 1 2013     1    13    1339        1345       -6    1449      1458
#> 2 2013     5     3    1253        1300       -7    1421      1410
#> 3 2013    10    14    2035        2045      -10    2146      2225
#> 4 2013    10    14     958        1000      -2    1105      1114
#> 5 2013     2     9    1237        1240      -3    1414      1444
#> 6 2013     7    18    2356        2220      96     58      2335
#> 7 2013     8     8     41        2225      136     138      2330
#> 8 2013     1    12    1541        1541       0    1742      1720
#> 9 2013     3    21    1913        1855      18    2301      2255
#> 10 2013    3     2    1455        1455       0    1802      1757
#> # ... with 33,667 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>
```

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

10.5.2 `slice_head`, `slice_tail`, `slice_min`, `slice_max`

`slice_head` et `slice_tail` permettent de sélectionner les premières ou dernière lignes d'un tableau. On peut indiquer le nombre (avec `n`) ou la proportion (avec `prop`) de lignes qu'on souhaite.

Ainsi si on veut afficher les trois premières lignes d'`airport` :

```
airports %>% slice_head(n = 3)
#> # A tibble: 3 x 8
#>   faa   name           lat   lon   alt   tz dst tzone
#>   <chr> <chr>         <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~
```

Si on veut afficher les dernières 20% des lignes de `airlines` :

```
airlines %>% slice_tail(prop = 0.2)
#> # A tibble: 3 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 VX      Virgin America
#> 2 WN      Southwest Airlines Co.
#> 3 YV      Mesa Airlines Inc.
```

`slice_min` et `slice_max` prennent en argument supplémentaire une variable du tableau et affichent `n` ou `prop` lignes du tableau ayant les valeurs les plus ou les moins élevées de cette variable.

Ainsi si on veut les 1% de lignes de `flights` ayant les plus fortes valeurs de `dep_delay` :

```
flights %>% slice_max(dep_delay, prop = 0.01)
#> # A tibble: 3,387 x 22
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>    <dbl>    <int>          <int>
#> 1 2013     1     9      641        900 1301 1242        1530
#> 2 2013     6    15     1432       1935 1137 1607        2120
#> 3 2013     1    10     1121       1635 1126 1239        1810
#> 4 2013     9    20     1139       1845 1014 1457        2210
#> 5 2013     7    22      845       1600 1005 1044        1815
#> 6 2013     4    10     1100       1900  960 1342        2211
#> 7 2013     3    17     2321       810   911 135         1020
#> 8 2013     6    27      959       1900  899 1236        2226
#> 9 2013     7    22     2257       759   898 121         1026
#> 10 2013    12     5      756       1700  896 1058        2020
#> # ... with 3,377 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>
```

Si on veut les 2 aéroports avec l'altitude la plus basse :

```
airports %>% slice_min(alt, n = 2)
#> # A tibble: 2 x 8
#>   faa      name      lat    lon    alt    tz dst tzone
#>   <chr>    <chr>    <dbl> <dbl> <dbl> <chr> <chr>
#> 1 IPL      Imperial Co 32.8 -116.  -54   -8 A   America/Los_Angeles
#> 2 NJK      El Centro Naf 32.8 -116.  -42   -8 A   America/Los_Angeles
```

On notera que le tableau renvoyé par `slice_min` et `slice_max` est trié selon la variable d'intérêt.

10.5.3 lead et lag

`lead` et `lag` permettent de décaler les observations d'une variable d'un cran vers l'arrière (pour `lead`) ou vers l'avant (pour `lag`).

```
lead(1:5)
#> [1] 2 3 4 5 NA
lag(1:5)
#> [1] NA 1 2 3 4
```

Ceci peut être utile pour des données de type “séries temporelles”. Par exemple, on peut facilement calculer l'écart entre le retard au départ de chaque vol et celui du vol précédent :

```
flights %>%
  mutate(
    dep_delay_prev = lag(dep_delay),
    dep_delay_diff = dep_delay - dep_delay_prev
  ) %>%
  select(dep_delay_prev, dep_delay, dep_delay_diff)
#> # A tibble: 336,776 x 3
#>   dep_delay_prev dep_delay dep_delay_diff
#>   <dbl>        <dbl>        <dbl>
```

```
#> 1      NA      2      NA
#> 2      2      4      2
#> 3      4      2     -2
#> 4      2     -1     -3
#> 5     -1     -6     -5
#> 6     -6     -4      2
#> 7     -4     -5     -1
#> 8     -5     -3      2
#> 9     -3     -3      0
#> 10    -3     -2      1
#> # ... with 336,766 more rows
```

10.5.4 distinct et n_distinct

`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, `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 22
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>        <int>     <dbl>    <int>        <int>
#> 1 2013     1     1      517         515       2     830        819
#> 2 2013     1     2      42          2359      43     518        442
#> 3 2013     1     3      32          2359      33     504        442
#> 4 2013     1     4      25          2359      26     505        442
#> 5 2013     1     5      14          2359      15     503        445
#> 6 2013     1     6      16          2359      17     451        442
#> 7 2013     1     7      49          2359      50     531        444
#> 8 2013     1     8      454         500      -6     625        648
#> 9 2013     1     9      2          2359      3     432        444
#> 10 2013    1    10      3          2359      4     426        437
#> # ... with 355 more rows, and 14 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   duree_h <dbl>, distance_km <dbl>, vitesse <dbl>
```

La fonction `n_distinct`, elle, renvoie le nombre de valeurs distinctes d'un vecteur. On peut notamment l'utiliser dans un `summarise`.

Dans l'exemple qui suit on calcule, pour les trois aéroports de départ de la table `flights` le nombre de valeurs distinctes de l'aéroport d'arrivée :

```
flights %>%
  group_by(origin) %>%
  summarise(n_dest = n_distinct(dest))
#> # A tibble: 3 x 2
#>   origin n_dest
#>   <chr>   <int>
#> 1 EWR      86
#> 2 JFK      70
#> 3 LGA      68
```

10.5.5 `relocate`

`relocate` peut être utilisé pour réordonner les colonnes d'une table. Par défaut, si on lui passe un ou plusieurs noms de colonnes, `relocate` les place en début de tableau.

```
airports %>% relocate(lat, lon)
#> # A tibble: 1,458 x 8
#>   lat    lon faa   name           alt   tz dst tzone
#>   <dbl> <dbl> <chr> <chr>      <dbl> <dbl> <chr> <chr>
#> 1 41.1 -80.6 04G Lansdowne Airport      1044  -5 A America/-
#> 2 32.5 -85.7 06A Moton Field Municipal Airport  264   -6 A America/-
#> 3 42.0 -88.1 06C Schaumburg Regional      801   -6 A America/-
#> 4 41.4 -74.4 06N Randall Airport        523   -5 A America/-
#> 5 31.1 -81.4 09J Jekyll Island Airport      11   -5 A America/-
#> 6 36.4 -82.2 049 Elizabethton Municipal Airport 1593   -5 A America/-
#> 7 41.5 -84.5 066 Williams County Airport      730   -5 A America/-
```

```
#> 8 42.9 -76.8 0G7 Finger Lakes Regional Airport 492 -5 A America/~
#> 9 39.8 -76.6 0P2 Shoestring Aviation Airfield 1000 -5 U America/~
#> 10 48.1 -123. 0S9 Jefferson County Intl 108 -8 A America/~
#> # ... with 1,448 more rows
```

Les arguments supplémentaires `.before` et `.after` permettent de préciser à quel endroit déplacer la ou les colonnes indiquées.

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

10.6 Tables multiples

Le jeu de données `nycflights13` est un exemple de données réparties en plusieurs tables. Ici on en a trois : les informations sur les vols dans `flights`, celles sur les aéroports dans `airports` et celles sur les compagnies aériennes dans `airlines`.

`dplyr` propose différentes fonctions permettant de travailler avec des données structurées de cette manière.

10.6.1 Concaténation : `bind_rows` et `bind_cols`

Les fonctions `bind_rows` et `bind_cols` permettent d'ajouter des lignes (respectivement des colonnes) à une table à partir d'une ou plusieurs autres tables.

L'exemple suivant (certes très artificiel) montre l'utilisation de `bind_rows`. On commence par créer trois tableaux `t1`, `t2` et `t3` :

```
t1 <- airports %>%
  select(faa, name, lat, lon) %>%
  slice(1:2)
t1
#> # A tibble: 2 x 4
#>   faa     name      lat    lon
#>   <chr>  <chr>    <dbl> <dbl>
#> 1 04G   Lansdowne Airport 41.1  -80.6
#> 2 06A   Moton Field Municipal Airport 32.5  -85.7
```

```
t2 <- airports %>%
  select(faa, name, lat, lon) %>%
  slice(5:6)

t2
#> # A tibble: 2 x 4
#>   faa     name           lat   lon
#>   <chr>   <chr>        <dbl> <dbl>
#> 1 09J    Jekyll Island Airport  31.1 -81.4
#> 2 0A9    Elizabethton Municipal Airport 36.4 -82.2
```

```
t3 <- airports %>%
  select(faa, name) %>%
  slice(100:101)

t3
#> # A tibble: 2 x 2
#>   faa     name
#>   <chr>   <chr>
#> 1 ADW    Andrews Afb
#> 2 AET    Allakaket Airport
```

On concaténe ensuite les trois tables avec `bind_rows` :

```
bind_rows(t1, t2, t3)
#> # A tibble: 6 x 4
#>   faa     name           lat   lon
#>   <chr>   <chr>        <dbl> <dbl>
#> 1 04G    Lansdowne Airport  41.1 -80.6
#> 2 06A    Moton Field Municipal Airport 32.5 -85.7
#> 3 09J    Jekyll Island Airport  31.1 -81.4
#> 4 0A9    Elizabethton Municipal Airport 36.4 -82.2
#> 5 ADW    Andrews Afb       NA     NA
#> 6 AET    Allakaket Airport    NA     NA
```

On remarquera que si des colonnes sont manquantes pour certaines tables, comme les colonnes `lat` et `lon` de `t3`, des `NA` sont automatiquement insérées.

Il peut être utile, quand on concatène des lignes, de garder une trace du tableau d'origine de chacune des lignes dans le tableau final. C'est possible grâce à l'argument `.id` de `bind_rows`. On passe à cet argument le nom d'une colonne qui contiendra l'indicateur d'origine des lignes :

```
bind_rows(t1, t2, t3, .id = "source")
#> # A tibble: 6 x 5
#>   source faa     name           lat   lon
#>   <chr>  <chr>   <chr>        <dbl> <dbl>
#> 1 1      04G    Lansdowne Airport  41.1 -80.6
#> 2 1      06A    Moton Field Municipal Airport 32.5 -85.7
#> 3 2      09J    Jekyll Island Airport  31.1 -81.4
#> 4 2      0A9    Elizabethton Municipal Airport 36.4 -82.2
#> 5 3      ADW    Andrews Afb       NA     NA
#> 6 3      AET    Allakaket Airport    NA     NA
```

Par défaut la colonne `.id` ne contient qu'un nombre, différent pour chaque tableau. On peut lui spécifier des valeurs plus explicites en “nommant” les tables dans `bind_rows` de la manière suivante :

```
bind_rows(table1 = t1, table2 = t2, table3 = t3, .id = "source")
#> # A tibble: 6 x 5
#>   source faa    name          lat    lon
#>   <chr>  <chr> <chr>        <dbl> <dbl>
#> 1 table1 04G  Lansdowne Airport  41.1 -80.6
#> 2 table1 06A  Moton Field Municipal Airport 32.5 -85.7
#> 3 table2 09J  Jekyll Island Airport 31.1 -81.4
#> 4 table2 0A9  Elizabethton Municipal Airport 36.4 -82.2
#> 5 table3 ADW  Andrews Afb       NA     NA
#> 6 table3 AET  Allakaket Airport    NA     NA
```

`bind_cols` permet de concaténer des colonnes et fonctionne de manière similaire :

```
t1 <- flights %>% slice(1:5) %>% select(dep_delay, dep_time)
t2 <- flights %>% slice(1:5) %>% select(origin, dest)
t3 <- flights %>% slice(1:5) %>% select(arr_delay, arr_time)
bind_cols(t1, t2, t3)
#> # A tibble: 5 x 6
#>   dep_delay dep_time origin dest arr_delay arr_time
#>   <dbl>      <int> <chr> <chr>    <dbl>      <int>
#> 1         2      517 EWR  IAH      11       830
#> 2         4      533 LGA  IAH      20       850
#> 3         2      542 JFK  MIA      33       923
#> 4        -1      544 JFK  BQN     -18      1004
#> 5        -6      554 LGA  ATL     -25       812
```

À noter que `bind_cols` associe les lignes uniquement *par position*. Les lignes des différents tableaux associés doivent donc correspondre (et leur nombre doit être identique). Pour associer des tables *par valeur*, on doit utiliser des jointures.

10.6.2 Jointures

10.6.2.1 Clés implicites

Très souvent, les données relatives à une analyse sont réparties dans plusieurs tables différentes. Dans notre exemple, on peut voir que la table `flights` contient le code de la compagnie aérienne du vol dans la variable `carrier` :

```
flights %>% select(carrier)
#> # A tibble: 336,776 x 1
#>   carrier
#>   <chr>
#> 1 UA
#> 2 UA
#> 3 AA
#> 4 B6
#> 5 DL
#> 6 UA
#> 7 B6
#> 8 EV
```

```
#> 9 B6
#> 10 AA
#> # ... with 336,766 more rows
```

Et que par ailleurs la table `airlines` contient une information supplémentaire relative à ces compagnies, à savoir le nom complet.

```
airlines
#> # A tibble: 16 x 2
#>   carrier name
#>   <chr>   <chr>
#> 1 9E      Endeavor Air Inc.
#> 2 AA      American Airlines Inc.
#> 3 AS      Alaska Airlines Inc.
#> 4 B6      JetBlue Airways
#> 5 DL      Delta Air Lines Inc.
#> 6 EV      ExpressJet Airlines Inc.
#> 7 F9      Frontier Airlines Inc.
#> 8 FL      AirTran Airways Corporation
#> 9 HA      Hawaiian Airlines Inc.
#> 10 MQ     Envoy Air
#> 11 OO     SkyWest Airlines Inc.
#> 12 UA     United Air Lines Inc.
#> 13 US     US Airways Inc.
#> 14 VX     Virgin America
#> 15 WN     Southwest Airlines Co.
#> 16 YV     Mesa Airlines Inc.
```

Il est donc naturel de vouloir associer les deux, ici pour ajouter les noms complets des compagnies à la table `flights`. Pour cela on va effectuer une *jointure* : les lignes d'une table seront associées à une autre en se basant non pas sur leur position, mais sur les valeurs d'une ou plusieurs colonnes. Ces colonnes sont appelées des *clés*.

Pour faire une jointure de ce type, on va utiliser la fonction `left_join` :

```
left_join(flights, airlines)
```

Pour faciliter la lecture, on va afficher seulement certaines colonnes du résultat :

```
left_join(flights, airlines) %>%
  select(month, day, carrier, name)
#> Joining, by = "carrier"
#> # A tibble: 336,776 x 4
#>   month   day   carrier name
#>   <int> <int> <chr>   <chr>
#> 1     1     1  UA      United Air Lines Inc.
#> 2     1     1  UA      United Air Lines Inc.
#> 3     1     1  AA      American Airlines Inc.
#> 4     1     1  B6      JetBlue Airways
#> 5     1     1  DL      Delta Air Lines Inc.
#> 6     1     1  UA      United Air Lines Inc.
#> 7     1     1  B6      JetBlue Airways
#> 8     1     1  EV      ExpressJet Airlines Inc.
#> 9     1     1  B6      JetBlue Airways
#> 10    1     1  AA      American Airlines Inc.
#> # ... with 336,766 more rows
```

On voit que la table résultat est bien la fusion des deux tables d'origine selon les valeurs des deux colonnes clés `carrier`. On est parti de la table `flights`, et pour chaque ligne de celle-ci on a ajouté les colonnes de `airlines` pour lesquelles la valeur de `carrier` est la même. On a donc bien une nouvelle colonne `name` dans notre table résultat, avec le nom complet de la compagnie aérienne.



À noter qu'on peut tout à fait utiliser le *pipe* avec les fonctions de jointure :

```
flights %>% left_join(airlines).
```

Nous sommes ici dans le cas le plus simple concernant les clés de jointure : les deux clés sont uniques et portent le même nom dans les deux tables. Par défaut, si on ne lui spécifie pas explicitement les clés, `dplyr` fusionne en utilisant l'ensemble des colonnes communes aux deux tables. On peut d'ailleurs voir dans cet exemple qu'un message a été affiché précisant que la jointure s'est bien faite sur la variable `carrier`.

10.6.2.2 Clés explicites

La table `airports`, contient des informations supplémentaires sur les aéroports : nom complet, altitude, position géographique, etc. Chaque aéroport est identifié par un code contenu dans la colonne `faa`.

Si on regarde la table `flights`, on voit que le code d'identification des aéroports apparaît à deux endroits différents : pour l'aéroport de départ dans la colonne `origin`, et pour celui d'arrivée dans la colonne `dest`. On a donc deux clés de jointure possibles, et qui portent un nom différent de la clé de `airports`.

On va commencer par fusionner les données concernant l'aéroport de départ. Pour simplifier l'affichage des résultats, on va se contenter d'un sous-ensemble des deux tables :

```
flights_ex <- flights %>% select(month, day, origin, dest)
airports_ex <- airports %>% select(faa, alt, name)
```

Si on se contente d'un `left_join` comme à l'étape précédente, on obtient un message d'erreur car aucune colonne commune ne peut être identifiée comme clé de jointure :

```
flights_ex %>% left_join(airports_ex)
#> Error: `by` must be supplied when `x` and `y` have no common variables.
#> i use by = character() to perform a cross-join.
```

On doit donc spécifier explicitement les clés avec l'argument `by` de `left_join`. Ici la clé est nommée `origin` dans la première table, et `faa` dans la seconde. La syntaxe est donc la suivante :

```
flights_ex %>%
  left_join(airports_ex, by = c("origin" = "faa"))
#> # A tibble: 336,776 x 6
#>   month   day origin dest    alt name
#>   <int> <int> <chr>  <chr> <dbl> <chr>
#> 1     1     1 EWR    IAH      18 Newark Liberty Intl
#> 2     1     1 LGA    IAH      22 La Guardia
#> 3     1     1 JFK    MIA      13 John F Kennedy Intl
#> 4     1     1 JFK    BQN      13 John F Kennedy Intl
#> 5     1     1 LGA    ATL      22 La Guardia
#> 6     1     1 EWR    ORD      18 Newark Liberty Intl
#> 7     1     1 EWR    FLL      18 Newark Liberty Intl
#> 8     1     1 LGA    IAD      22 La Guardia
#> 9     1     1 JFK    MCO      13 John F Kennedy Intl
#> 10    1     1 LGA    ORD      22 La Guardia
#> # ... with 336,766 more rows
```

On constate que les deux nouvelles colonnes `name` et `alt` contiennent bien les données correspondant à l'aéroport de départ.

On va stocker le résultat de cette jointure dans la table `flights_ex` :

```
flights_ex <- flights_ex %>%
  left_join(airports_ex, by = c("origin" = "faa"))
```

Supposons qu'on souhaite maintenant fusionner à nouveau les informations de la table `airports`, mais cette fois pour les aéroports d'arrivée de notre nouvelle table `flights_ex`. Les deux clés sont donc désormais `dest` dans la première table, et `faa` dans la deuxième. La syntaxe est donc la suivante :

```
flights_ex %>%
  left_join(airports_ex, by = c("dest" = "faa"))
#> # A tibble: 336,776 x 8
#>   month   day origin dest alt.x name.x          alt.y name.y
#>   <int> <int> <chr>  <chr> <dbl> <chr>          <dbl> <chr>
#> 1     1     1 EWR    IAH      18 Newark Liberty Intl  97 George Bush Interco-
#> 2     1     1 LGA    IAH      22 La Guardia        97 George Bush Interco-
#> 3     1     1 JFK    MIA      13 John F Kennedy Intl  8 Miami Intl
#> 4     1     1 JFK    BQN      13 John F Kennedy Intl NA <NA>
#> 5     1     1 LGA    ATL      22 La Guardia        1026 Hartsfield Jackson ~
#> 6     1     1 EWR    ORD      18 Newark Liberty Intl  668 Chicago O'hare Intl
#> 7     1     1 EWR    FLL      18 Newark Liberty Intl  9 Fort Lauderdale Hol-
#> 8     1     1 LGA    IAD      22 La Guardia        313 Washington Dulles I~
#> 9     1     1 JFK    MCO      13 John F Kennedy Intl  96 Orlando Intl
#> 10    1     1 LGA    ORD      22 La Guardia        668 Chicago O'hare Intl
#> # ... with 336,766 more rows
```

Cela fonctionne, les informations de l'aéroport d'arrivée ont bien été ajoutées, mais on constate que les colonnes ont été renommées. En effet, ici les deux tables fusionnées contenaient toutes les deux des colonnes `name` et `alt`. Comme on ne peut pas avoir deux colonnes avec le même nom dans un tableau, `dplyr` a renommé les colonnes de la première table en `name.x` et `alt.x`, et celles de la deuxième en `name.y` et `alt.y`.

C'est pratique, mais pas forcément très parlant. On pourrait renommer manuellement les colonnes avec `rename` avant de faire la jointure pour avoir des intitulés plus explicites, mais on peut aussi utiliser l'argument `suffix` de `left_join`, qui permet d'indiquer les suffixes à ajouter aux colonnes.

```
flights_ex %>%
  left_join(
    airports_ex,
    by = c("dest" = "faa"),
    suffix = c("_depart", "_arrivee")
)
#> # A tibble: 336,776 x 8
#>   month   day origin dest alt_depart name_depart alt_arrivee name_arrivee
#>   <int> <int> <chr>  <chr> <dbl> <chr>          <dbl> <chr>
#> 1     1     1 EWR    IAH      18 Newark Liber~  97 George Bush In~
#> 2     1     1 LGA    IAH      22 La Guardia  97 George Bush In~
#> 3     1     1 JFK    MIA      13 John F Kenne~  8 Miami Intl
#> 4     1     1 JFK    BQN      13 John F Kenne~ NA <NA>
#> 5     1     1 LGA    ATL      22 La Guardia  1026 Hartsfield Jac~
#> 6     1     1 EWR    ORD      18 Newark Liber~  668 Chicago O'hare ~
#> 7     1     1 EWR    FLL      18 Newark Liber~  9 Fort Lauderdale ~
#> 8     1     1 LGA    IAD      22 La Guardia  313 Washington Dul~
```

```
#> 9   1   1 JFK  MCO      13 John F Kenne~      96 Orlando Intl
#> 10  1   1 LGA  ORD      22 La Guardia     668 Chicago Ohare ~
#> # ... with 336,766 more rows
```

On obtient ainsi directement des noms de colonnes nettement plus clairs.

10.6.3 Types de jointures

Jusqu'à présent nous avons utilisé la fonction `left_join`, mais il existe plusieurs types de jointures.

Partons de deux tables d'exemple, `personnes` et `voitures` :

```
personnes <- tibble(
  nom = c("Sylvie", "Sylvie", "Monique", "Gunter", "Rayan", "Rayan"),
  voiture = c("Twingo", "Ferrari", "Scenic", "Lada", "Twingo", "Clio")
)
```

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Monique	Scenic
Gunter	Lada
Rayan	Twingo
Rayan	Clio

```
voitures <- tibble(
  voiture = c("Twingo", "Ferrari", "Clio", "Lada", "208"),
  vitesse = c("140", "280", "160", "85", "160")
)
```

voiture	vitesse
Twingo	140
Ferrari	280
Clio	160
Lada	85
208	160

10.6.3.1 left_join

Si on fait un `left_join` de `voitures` sur `personnes` :

```
personnes %>% left_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

On voit que chaque ligne de **personnes** est bien présente, et qu'on lui a ajouté une ligne de **voitures** correspondante si elle existe. Dans le cas du **Scenic**, il n'y a avait pas de ligne dans **voitures**, donc **vitesse** a été mise à **NA**. Dans le cas de **208**, présente dans **voitures** mais pas dans **personnes**, la ligne n'apparaît pas.

Si on fait un **left_join** cette fois de **personnes** sur **voitures**, c'est l'inverse :

```
voitures %>% left_join(personnes)
```

```
#> Joining, by = "voiture"
```

voiture	vitesse	nom
Twingo	140	Sylvie
Twingo	140	Rayan
Ferrari	280	Sylvie
Clio	160	Rayan
Lada	85	Gunter
208	160	NA

La ligne **208** est là, mais **nom** est à **NA**. Par contre **Monique** est absente. Et on remarquera que la ligne **Twingo**, présente deux fois dans **personnes**, a été dupliquée pour être associée aux deux lignes de données de **Sylvie** et **Rayan**.

En résumé, quand on fait un **left_join(x, y)**, toutes les lignes de **x** sont présentes, et dupliquées si nécessaire quand elles apparaissent plusieurs fois dans **y**. Les lignes de **y** non présentes dans **x** disparaissent. Les lignes de **x** non présentes dans **y** se voient attribuer des **NA** pour les nouvelles colonnes.

Intuitivement, on pourrait considérer que **left_join(x, y)** signifie “ramener l'information de la table **y** sur la table **x**”.

En général, **left_join** sera le type de jointures le plus fréquemment utilisé.

10.6.3.2 right_join

La jointure **right_join** est l'exacte symétrique de **left_join**, c'est-à-dire que **right_join(x, y)** est équivalent à **left_join(y, x)** :

```
personnes %>% right_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160
NA	208	160

10.6.3.3 inner_join

Dans le cas de **inner_join(x, y)**, seules les lignes présentes à la fois dans **x** et **y** sont conservées (et si nécessaire dupliquées) dans la table résultat :

```
personnes %>% inner_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160

Ici la ligne 208 est absente, ainsi que la ligne Monique, qui dans le cas d'un `left_join` avait été conservée et s'était vue attribuer une `vitesse` à NA.

10.6.3.4 full_join

Dans le cas de `full_join(x, y)`, toutes les lignes de `x` et toutes les lignes de `y` sont conservées (avec des NA ajoutés si nécessaire) même si elles sont absentes de l'autre table :

```
personnes %>% full_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture	vitesse
Sylvie	Twingo	140
Sylvie	Ferrari	280
Monique	Scenic	NA
Gunter	Lada	85
Rayan	Twingo	140
Rayan	Clio	160
NA	208	160

10.6.3.5 semi_join et anti_join

`semi_join` et `anti_join` sont des jointures *filtrantes*, c'est-à-dire qu'elles sélectionnent les lignes de `x` sans ajouter les colonnes de `y`.

Ainsi, `semi_join` ne conservera que les lignes de `x` pour lesquelles une ligne de `y` existe également, et supprimera les autres. Dans notre exemple, la ligne `Monique` est donc supprimée :

```
personnes %>% semi_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture
Sylvie	Twingo
Sylvie	Ferrari
Gunter	Lada
Rayan	Twingo
Rayan	Clio

Un `anti_join` fait l'inverse, il ne conserve que les lignes de `x` absentes de `y`. Dans notre exemple, on ne garde donc que la ligne Monique :

```
personnes %>% anti_join(voitures)
```

```
#> Joining, by = "voiture"
```

nom	voiture
Monique	Scenic

10.7 Ressources

Toutes les ressources ci-dessous sont en anglais...

Le livre *R for data science*, librement accessible en ligne, contient plusieurs chapitres très complets sur la manipulation des données, notamment :

- [Data transformation](#) pour les manipulations
- [Relational data](#) pour les tables multiples

Le [site de l'extension](#) comprend une [liste des fonctions](#) et les pages d'aide associées, mais aussi une [introduction](#) au package et plusieurs articles dont un spécifiquement sur les [jointures](#).

Enfin, une “antisèche” très synthétique est également accessible depuis RStudio, en allant dans le menu *Help* puis *Cheatsheets* et *Data Transformation with dplyr*.

10.8 Exercices

On commence par charger les extensions et les données nécessaires.

```
library(tidyverse)
library(nycflights13)
data(flights)
data(airports)
data(airlines)
```

10.8.1 Les verbes de base de *dplyr*

Exercice 1.1

Sélectionner les lignes 100 à 105 du tableau des vols (`flights`).

```
#> # A tibble: 6 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
#> 1 2013     1     1     752         759      -7     955         959
#> 2 2013     1     1     753         755      -2    1056        1110
#> 3 2013     1     1     754         759      -5    1039        1041
#> 4 2013     1     1     754         755      -1    1103        1030
#> 5 2013     1     1     758         800      -2    1053        1054
#> 6 2013     1     1     759         800      -1    1057        1127
#> # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercice 1.2

Sélectionnez les vols du mois de juillet (variable `month`).

```
#> # A tibble: 29,425 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
#> 1 2013     7     1       1         2029      212     236         2359
#> 2 2013     7     1       2         2359       3     344         344
#> 3 2013     7     1      29         2245     104     151          1
#> 4 2013     7     1      43         2130     193     322          14
#> 5 2013     7     1      44         2150     174     300         100
#> 6 2013     7     1      46         2051     235     304         2358
#> 7 2013     7     1      48         2001     287     308        2305
#> 8 2013     7     1      58         2155     183     335         43
#> 9 2013     7     1     100         2146     194     327         30
#> 10 2013    7     1     100         2245     135     337         135
#> # ... with 29,415 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Sélectionnez les vols avec un retard à l'arrivée (variable `arr_delay`) compris entre 5 et 15 minutes.

```
#> # A tibble: 36,392 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>     <int>           <int>     <dbl>     <int>           <int>
#> 1 2013     1     1     517         515      2     830         819
#> 2 2013     1     1     554         558     -4     740         728
#> 3 2013     1     1     558         600     -2     753         745
#> 4 2013     1     1     558         600     -2     924         917
#> 5 2013     1     1     600         600      0     837         825
#> 6 2013     1     1     611         600     11     945         931
#> 7 2013     1     1     623         610     13     920         915
#> 8 2013     1     1     624         630     -6     840         830
#> 9 2013     1     1     629         630     -1     824         810
#> 10 2013    1     1     632         608     24     740         728
#> # ... with 36,382 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Sélectionnez les vols des compagnies Delta, United et American (codes DL, UA et AA de la variable `carrier`).

```
#> # A tibble: 139,504 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1 2013     1     1      517        515       2     830        819
#> 2 2013     1     1      533        529       4     850        830
#> 3 2013     1     1      542        540       2     923        850
#> 4 2013     1     1      554        600      -6     812        837
#> 5 2013     1     1      554        558      -4     740        728
#> 6 2013     1     1      558        600      -2     753        745
#> 7 2013     1     1      558        600      -2     924        917
#> 8 2013     1     1      558        600      -2     923        937
#> 9 2013     1     1      559        600      -1     941        910
#> 10 2013    1     1      559        600      -1     854        902
#> # ... with 139,494 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercice 1.3

Triez la table flights par retard au départ décroissant.

```
#> # A tibble: 336,776 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1 2013     1     9      641        900      1301     1242        1530
#> 2 2013     6    15     1432       1935      1137     1607        2120
#> 3 2013     1    10     1121       1635      1126     1239        1810
#> 4 2013     9    20     1139       1845      1014     1457        2210
#> 5 2013     7    22      845       1600      1005     1044        1815
#> 6 2013     4    10     1100       1900      960      1342        2211
#> 7 2013     3    17     2321       810       911      135         1020
#> 8 2013     6    27      959       1900      899      1236        2226
#> 9 2013     7    22     2257       759       898      121         1026
#> 10 2013    12     5      756       1700      896      1058        2020
#> # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercice 1.4

Sélectionnez les colonnes name, lat et lon de la table airports

```
#> # A tibble: 1,458 x 3
#>   name                  lat    lon
#>   <chr>                <dbl> <dbl>
#> 1 Lansdowne Airport     41.1 -80.6
#> 2 Moton Field Municipal Airport 32.5 -85.7
#> 3 Schaumburg Regional   42.0 -88.1
#> 4 Randall Airport       41.4 -74.4
#> 5 Jekyll Island Airport 31.1 -81.4
#> 6 Elizabethton Municipal Airport 36.4 -82.2
#> 7 Williams County Airport 41.5 -84.5
#> 8 Finger Lakes Regional Airport 42.9 -76.8
#> 9 Shoestring Aviation Airfield 39.8 -76.6
```

```
#> 10 Jefferson County Intl          48.1 -123.
#> # ... with 1,448 more rows
```

Sélectionnez toutes les colonnes de la table `airports` sauf les colonnes `tz` et `tzone`

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

Toujours dans la table `airports`, renommez la colonne `lat` en `latitude` et `lon` en `longitude`.

```
#> # A tibble: 1,458 x 8
#>   faa     name           latitude longitude    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~ 32.5    -85.7 264   -6 A America/Chic~
#> 3 06C Schaumburg Regional    42.0    -88.1 801   -6 A America/Chic~
#> 4 06N Randall Airport      41.4    -74.4 523   -5 A America/New_~
#> 5 09J Jekyll Island Airpo~ 31.1    -81.4 11    -5 A America/New_~
#> 6 0A9 Elizabethton Municipi~ 36.4    -82.2 1593  -5 A America/New_~
#> 7 0G6 Williams County Air~ 41.5    -84.5 730   -5 A America/New_~
#> 8 0G7 Finger Lakes Region~ 42.9    -76.8 492   -5 A America/New_~
#> 9 0P2 Shoestring Aviation~ 39.8    -76.6 1000  -5 U America/New_~
#> 10 0S9 Jefferson County In~ 48.1    -123. 108   -8 A America/Los_~
#> # ... with 1,448 more rows
```

Exercice 1.5

Dans la table `airports`, la colonne `alt` contient l'altitude de l'aéroport en pieds. Créer une nouvelle variable `alt_m` contenant l'altitude en mètres (on convertit des pieds en mètres en les divisant par 3.2808). Sélectionner dans la table obtenue uniquement les deux colonnes `alt` et `alt_m`.

```
#> # A tibble: 1,458 x 2
#>   alt  alt_m
#>   <dbl> <dbl>
#> 1 1044 318.
#> 2 264  80.5
#> 3 801  244.
#> 4 523  159.
#> 5 11   3.35
#> 6 1593 486.
```

```
#> 7 730 223.
#> 8 492 150.
#> 9 1000 305.
#> 10 108 32.9
#> # ... with 1,448 more rows
```

10.8.2 Enchaîner des opérations

Exercice 2.1

Réécrire le code de l'exercice précédent en utilisant le *pipe* `%>%`.

Exercice 2.2

En utilisant le *pipe*, sélectionnez les vols à destination de San Francico (code SFO de la variable `dest`) et triez-les selon le retard au départ décroissant (variable `dep_delay`).

```
#> # A tibble: 13,331 x 19
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1 2013     9    20      1139          1845     1014      1457          2210
#> 2 2013     7     7      2123          1030      653       17           1345
#> 3 2013     7     7      2059          1030      629       106          1350
#> 4 2013     7     6      149          1600      589       456          1935
#> 5 2013     7    10      133          1800      453       455          2130
#> 6 2013     7    10      2342          1630      432       312          1959
#> 7 2013     7     7      2204          1525      399       107          1823
#> 8 2013     7     7      2306          1630      396       250          1959
#> 9 2013     6    23      1833          1200      393       NA           1507
#> 10 2013    7    10      2232          1609      383       138          1928
#> # ... with 13,321 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercice 2.3

Sélectionnez les vols des mois de septembre et octobre, conservez les colonnes `dest` et `dep_delay`, créez une nouvelle variable `retard_h` contenant le retard au départ en heures, triez selon `retard_h` par ordre décroissant et conservez uniquement les 5 premières lignes.

```
#> # A tibble: 5 x 3
#>   dest  dep_delay retard_h
#>   <chr>    <dbl>    <dbl>
#> 1 SFO      1014    16.9
#> 2 ATL       702    11.7
#> 3 DTW       696    11.6
#> 4 ATL       602    10.0
#> 5 MSP       593    9.88
```

10.8.3 `group_by` et `summarise`

Exercice 3.1

Affichez le nombre de vols par mois.

```
#> # A tibble: 12 x 2
#>   month     n
#>   <int> <int>
#> 1     1 27004
#> 2     2 24951
#> 3     3 28834
#> 4     4 28330
#> 5     5 28796
#> 6     6 28243
#> 7     7 29425
#> 8     8 29327
#> 9     9 27574
#> 10    10 28889
#> 11    11 27268
#> 12    12 28135
```

Triez la table résultatat selon le nombre de vols croissant.

```
#> # A tibble: 12 x 2
#>   month     n
#>   <int> <int>
#> 1     2 24951
#> 2     1 27004
#> 3    11 27268
#> 4     9 27574
#> 5    12 28135
#> 6     6 28243
#> 7     4 28330
#> 8     5 28796
#> 9     3 28834
#> 10    10 28889
#> 11    8 29327
#> 12    7 29425
```

Exercice 3.2

Calculer la distance moyenne des vols selon l'aéroport de départ (variable `origin`).

```
#> # A tibble: 3 x 2
#>   origin distance_moyenne
#>   <chr>          <dbl>
#> 1 EWR            1057.
#> 2 JFK            1266.
#> 3 LGA             780.
```

Exercice 3.3

Calculer le nombre de vols à destination de Los Angeles (code `LAX`) pour chaque mois de l'année.

```
#> # A tibble: 12 x 2
#>   month     n
```

```
#>      <int> <int>
#> 1     1 1159
#> 2     2 1030
#> 3     3 1178
#> 4     4 1382
#> 5     5 1453
#> 6     6 1430
#> 7     7 1500
#> 8     8 1505
#> 9     9 1384
#> 10   10 1409
#> 11   11 1336
#> 12   12 1408
```

Exercice 3.4

Calculer le nombre de vols selon le mois et la destination.

```
#> # 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
```

Ne conserver, pour chaque mois, que la destination avec le nombre maximal de vols.

```
#> # A tibble: 12 x 3
#> # Groups:   month [12]
#>   month dest    n
#>   <int> <chr> <int>
#> 1     1 ATL   1396
#> 2     2 ATL   1267
#> 3     3 ATL   1448
#> 4     4 ATL   1490
#> 5     5 ORD   1582
#> 6     6 ORD   1547
#> 7     7 ORD   1573
#> 8     8 ORD   1604
#> 9     9 ORD   1582
#> 10   10 ORD  1604
#> 11   11 ATL   1384
#> 12   12 ATL   1463
```

Exercice 3.5

Calculer le nombre de vols selon le mois. Ajouter une colonne comportant le pourcentage de vols annuels réalisés par mois.

```
#> # A tibble: 12 x 3
#>   month     n pourcentage
#>   <int> <int>      <dbl>
#> 1     1 27004      8.02
#> 2     2 24951      7.41
#> 3     3 28834      8.56
#> 4     4 28330      8.41
#> 5     5 28796      8.55
#> 6     6 28243      8.39
#> 7     7 29425      8.74
#> 8     8 29327      8.71
#> 9     9 27574      8.19
#> 10   10 28889      8.58
#> 11   11 27268      8.10
#> 12   12 28135      8.35
```

Exercice 3.6

Calculer, pour chaque destination et chaque mois, le retard moyen à l'arrivée. Pour chaque mois, trier les destinations selon ce retard moyen décroissant, et (toujours pour chaque mois) ne conserver que les trois destinations avec le retard le plus important.

```
#> `summarise()` has grouped output by 'month'. You can override using the `groups` argument.
#> # A tibble: 36 x 3
#> # Groups:   month [12]
#>   month dest  retard_moyen
#>   <int> <chr>    <dbl>
#> 1     1 TUL       68.1
#> 2     1 OKC       57.7
#> 3     1 CAE       55.9
#> 4     2 DSM       48.2
#> 5     2 TUL       33.5
#> 6     2 GSP       32.9
#> 7     3 DSM       60.6
#> 8     3 CAE       46.9
#> 9     3 PVD       44.3
#> 10    4 CAE       71.3
#> # ... with 26 more rows
```

10.8.4 Jointures

Exercice 4.1

Faire la jointure de la table `airlines` sur la table `flights` à l'aide de `left_join`.

```
#> # A tibble: 336,776 x 20
#>   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>      <dbl>    <int>          <int>
#> 1 2013     1     1      517            515        2     830          819
#> 2 2013     1     1      533            529        4     850          830
#> 3 2013     1     1      542            540        2     923          850
#> 4 2013     1     1      544            545       -1    1004         1022
#> 5 2013     1     1      554            600       -6     812          837
```

```
#> 6 2013 1 1 554 558 -4 740 728
#> 7 2013 1 1 555 600 -5 913 854
#> 8 2013 1 1 557 600 -3 709 723
#> 9 2013 1 1 557 600 -3 838 846
#> 10 2013 1 1 558 600 -2 753 745
#> # ... with 336,766 more rows, and 12 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   name <chr>
```

Exercice 4.2

À partir de la table résultatat de l'exercice précédent, calculer le retard moyen au départ pour chaque compagnie, et trier selon ce retard décroissant.

```
#> # A tibble: 16 x 2
#>   name           retard_moyen
#>   <chr>          <dbl>
#> 1 Frontier Airlines Inc.    20.2
#> 2 ExpressJet Airlines Inc.  20.0
#> 3 Mesa Airlines Inc.       19.0
#> 4 AirTran Airways Corporation 18.7
#> 5 Southwest Airlines Co.   17.7
#> 6 Endeavor Air Inc.        16.7
#> 7 JetBlue Airways         13.0
#> 8 Virgin America          12.9
#> 9 SkyWest Airlines Inc.   12.6
#> 10 United Air Lines Inc.  12.1
#> 11 Envoy Air              10.6
#> 12 Delta Air Lines Inc.   9.26
#> 13 American Airlines Inc. 8.59
#> 14 Alaska Airlines Inc.   5.80
#> 15 Hawaiian Airlines Inc. 4.90
#> 16 US Airways Inc.        3.78
```

Exercice 4.3

Faire la jointure de la table `airports` sur la table `flights` en utilisant comme clé le code de l'aéroport de destination.

```
#> # A tibble: 336,776 x 26
#>   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int> <int>          <int>     <dbl>      <int>          <int>
#> 1 2013 1 1 517 515 2 830 819
#> 2 2013 1 1 533 529 4 850 830
#> 3 2013 1 1 542 540 2 923 850
#> 4 2013 1 1 544 545 -1 1004 1022
#> 5 2013 1 1 554 600 -6 812 837
#> 6 2013 1 1 554 558 -4 740 728
#> 7 2013 1 1 555 600 -5 913 854
#> 8 2013 1 1 557 600 -3 709 723
#> 9 2013 1 1 557 600 -3 838 846
#> 10 2013 1 1 558 600 -2 753 745
#> # ... with 336,766 more rows, and 18 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
```

```
#> #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
#> #   name <chr>, lat <dbl>, lon <dbl>, alt <dbl>, tz <dbl>, dst <chr>,
#> #   tzone <chr>
```

À partir de cette table, afficher pour chaque mois le nom de l'aéroport de destination ayant eu le plus petit nombre de vol.

```
#> # A tibble: 14 x 3
#> # Groups: month [12]
#>   month name          n
#>   <int> <chr>       <int>
#> 1     1 Key West Intl      1
#> 2     2 Jackson Hole Airport    3
#> 3     3 Bangor Intl        2
#> 4     4 Key West Intl      1
#> 5     4 Myrtle Beach Intl    1
#> 6     5 Columbia Metropolitan 9
#> 7     6 Myrtle Beach Intl    1
#> 8     7 La Guardia        1
#> 9     8 South Bend Rgnl    1
#> 10    9 South Bend Rgnl    5
#> 11    10 Albany Intl       1
#> 12    10 South Bend Rgnl    1
#> 13    11 Blue Grass       1
#> 14    12 South Bend Rgnl    1
```

Exercice 4.4

Créer une table indiquant, pour chaque trajet, le nom de l'aéroport de départ et celui de l'aéroport d'arrivée.

```
#> # A tibble: 336,776 x 2
#>   orig_name      dest_name
#>   <chr>           <chr>
#> 1 Newark Liberty Intl George Bush Intercontinental
#> 2 La Guardia      George Bush Intercontinental
#> 3 John F Kennedy Intl Miami Intl
#> 4 John F Kennedy Intl <NA>
#> 5 La Guardia      Hartsfield Jackson Atlanta Intl
#> 6 Newark Liberty Intl Chicago Ohare Intl
#> 7 Newark Liberty Intl Fort Lauderdale Hollywood Intl
#> 8 La Guardia      Washington Dulles Intl
#> 9 John F Kennedy Intl Orlando Intl
#> 10 La Guardia     Chicago Ohare Intl
#> # ... with 336,766 more rows
```

10.8.5 Bonus

Exercice 5.1

Calculer le nombre de vols selon l'aéroport de destination, et fusionnez la table `airports` sur le résultat avec `left_join`. Stocker le résultat final dans un objet nommé `flights_dest`.

Créez une carte interactive des résultats avec le package `leaflet` et le code suivant :

```
library(leaflet)
leaflet(data = flights_dest) %>%
  addTiles %>%
  addCircles(lng = ~lon, lat = ~lat, radius = ~n * 10, popup = ~name)
#> Warning in validateCoords(lng, lat, funcName): Data contains 4 rows with either
#> missing or invalid lat/lon values and will be ignored
```

Chapitre 11

Manipuler du texte avec `stringr`

Les fonctions de `forcats` vues précédemment permettent de modifier des modalités d'une variables qualitative globalement. Mais parfois on a besoin de manipuler le contenu même du texte d'une variable de type chaîne de caractères : combiner, rechercher, remplacer...

On va utiliser ici les fonctions de l'extension `stringr`. Celle-ci fait partie du cœur du *tidyverse*, elle est donc automatiquement chargée avec :

```
library(tidyverse)
```



`stringr` est en fait une interface simplifiée aux fonctions d'une autre extension, `stringi`. Si les fonctions de `stringr` ne sont pas suffisantes ou si vous manipulez beaucoup de chaînes de caractères, n'hésitez pas à vous reporter à la documentation de `stringi`.

Dans ce qui suit on va utiliser le court tableau d'exemple `d` suivant :

```
d <- tibble(  
  nom = c("Mr Félicien Machin", "Mme Raymonde Bidule", "M. Martial Truc", "Mme Huguette Chose"),  
  adresse = c("3 rue des Fleurs", "47 ave de la Libération", "12 rue du 17 octobre 1961", "221 avenue de la Libération"),  
  ville = c("Nouméa", "Marseille", "Vénissieux", "Marseille"))
```

nom	adresse	ville
Mr Félicien Machin	3 rue des Fleurs	Nouméa
Mme Raymonde Bidule	47 ave de la Libération	Marseille
M. Martial Truc	12 rue du 17 octobre 1961	Vénissieux
Mme Huguette Chose	221 avenue de la Libération	Marseille

11.1 Expressions régulières

Les fonctions présentées ci-dessous sont pour la plupart prévues pour fonctionner avec des *expressions régulières*. Celles-ci constituent un mini-langage, qui peut paraître assez cryptique, mais qui est très puissant pour spécifier des motifs de chaînes de caractères.

Elles permettent par exemple de sélectionner le dernier mot avant la fin d'une chaîne, l'ensemble des suites alphanumériques commençant par une majuscule, des nombres de 3 ou 4 chiffres situés en début de chaîne, et beaucoup beaucoup d'autres choses encore bien plus complexes.

Pour donner un exemple concret, l'expression régulière suivante permet de détecter une adresse de courrier électronique¹ :

```
[\w\d+._]+@[\\w\\d.-]+\.\[a-zA-Z]{2,}
```

Par souci de simplicité, dans ce qui suit les exemples seront donnés autant que possible avec de simples chaînes de caractères, sans expression régulière. Mais si vous pensez manipuler des données textuelles, il peut être très utile de s'intéresser à cette syntaxe.

11.2 Concaténer des chaînes

La première opération de base consiste à concaténer des chaînes de caractères entre elles. On peut le faire avec la fonction `paste`.

Par exemple, si on veut concaténer l'adresse et la ville :

```
paste(d$adresse, d$ville)
#> [1] "3 rue des Fleurs Nouméa"
#> [2] "47 ave de la Libération Marseille"
#> [3] "12 rue du 17 octobre 1961 Vénissieux"
#> [4] "221 avenue de la Libération Marseille"
```

Par défaut, `paste` concatène en ajoutant un espace entre les différentes chaînes. On peut spécifier un autre séparateur avec son argument `sep` :

```
paste(d$adresse, d$ville, sep = " - ")
#> [1] "3 rue des Fleurs - Nouméa"
#> [2] "47 ave de la Libération - Marseille"
#> [3] "12 rue du 17 octobre 1961 - Vénissieux"
#> [4] "221 avenue de la Libération - Marseille"
```

Il existe une variante, `paste0`, qui concatène sans mettre de séparateur, et qui est légèrement plus rapide :

```
paste0(d$adresse, d$ville)
#> [1] "3 rue des FleursNouméa"
#> [2] "47 ave de la LibérationMarseille"
#> [3] "12 rue du 17 octobre 1961Vénissieux"
#> [4] "221 avenue de la LibérationMarseille"
```



À noter que `paste` et `paste0` sont des fonctions R de base. L'équivalent pour `stringr` se nomme `str_c`.

Parfois on cherche à concaténer les différents éléments d'un vecteur non pas avec ceux d'un autre vecteur, comme on l'a fait précédemment, mais *entre eux*. Dans ce cas `paste` seule ne fera rien :

```
paste(d$ville)
#> [1] "Nouméa"      "Marseille"   "Vénissieux"  "Marseille"
```

¹Il s'agit en fait d'une version très simplifiée, la “véritable” expression permettant de tester si une adresse mail est valide fait plus de 80 lignes...

Il faut lui ajouter un argument `collapse`, avec comme valeur la chaîne à utiliser pour concaténer les éléments :

```
paste(d$ville, collapse = ", ")
#> [1] "Nouméa, Marseille, Vénissieux, Marseille"
```

11.3 Convertir en majuscules / minuscules

Les fonctions `str_to_lower`, `str_to_upper` et `str_to_title` permettent respectivement de mettre en minuscules, mettre en majuscules, ou de capitaliser les éléments d'un vecteur de chaînes de caractères :

```
str_to_lower(d$nom)
#> [1] "mr félicien machin"  "mme raymonde bidule" "m. martial truc"
#> [4] "mme huguette chose"
```

```
str_to_upper(d$nom)
#> [1] "MR FÉLICIEN MACHIN"  "MME RAYMONDE BIDULE" "M. MARTIAL TRUC"
#> [4] "MME HUGUETTE CHOSE"
```

```
str_to_title(d$nom)
#> [1] "Mr Félicien Machin"  "Mme Raymonde Bidule" "M. Martial Truc"
#> [4] "Mme Huguette Chose"
```

11.4 Découper des chaînes

La fonction `str_split` permet de “découper” une chaîne de caractère en fonction d'un délimiteur. On passe la chaîne en premier argument, et le délimiteur en second :

```
str_split("un-deux-trois", "-")
#> [[1]]
#> [1] "un"     "deux"   "trois"
```

On peut appliquer la fonction à un vecteur, dans ce cas le résultat sera une liste :

```
str_split(d$nom, " ")
#> [[1]]
#> [1] "Mr"        "Félicien"  "Machin"
#>
#> [[2]]
#> [1] "Mme"       "Raymonde"  "Bidule"
#>
#> [[3]]
#> [1] "M."        "Martial"   "Truc"
#>
#> [[4]]
#> [1] "Mme"       "Huguette"  "Chose"
```

Ou un tableau (plus précisément une matrice) si on ajoute `simplify = TRUE`.

```
str_split(d$nom, " ", simplify = TRUE)
#> [1] [2] [3]
#> [1,] "Mr" "Félicien" "Machin"
#> [2,] "Mme" "Raymonde" "Bidule"
#> [3,] "M." "Martial" "Truc"
#> [4,] "Mme" "Huguette" "Chose"
```

Si on souhaite créer de nouvelles colonnes dans un tableau de données en découplant une colonne de type texte, on pourra utiliser la fonction `separate` de l'extension `tidyR`. Celle-ci est expliquée section 12.3.3.

Voici juste un exemple de son utilisation :

```
library(tidyR)
d %>% separate(nom, c("genre", "prenom", "nom"), sep = " ")
#> # A tibble: 4 x 5
#>   genre prenom nom      adresse           ville
#>   <chr>  <chr>  <chr>    <chr>            <chr>
#> 1 Mr     Félicien Machin 3 rue des Fleurs Nouméa
#> 2 Mme    Raymonde Bidule 47 ave de la Libération Marseille
#> 3 M.     Martial Truc   12 rue du 17 octobre 1961 Vénissieux
#> 4 Mme    Huguette Chose  221 avenue de la Libération Marseille
```

11.5 Extraire des sous-chaînes par position

La fonction `str_sub` permet d'extraire des sous-chaînes par position, en indiquant simplement les positions des premier et dernier caractères :

```
str_sub(d$ville, 1, 3)
#> [1] "Nou" "Mar" "Vén" "Mar"
```

11.6 Déetecter des motifs

`str_detect` permet de détecter la présence d'un motif parmi les éléments d'un vecteur. Par exemple, si on souhaite identifier toutes les adresses contenant "Libération" :

```
str_detect(d$adresse, "Libération")
#> [1] FALSE TRUE FALSE TRUE
```

`str_detect` renvoie un vecteur de valeurs logiques et peut donc être utilisée, par exemple, avec le verbe `filter` de `dplyr` pour extraire des sous-populations.

Une variante, `str_count`, compte le nombre d'occurrences d'une chaîne pour chaque élément d'un vecteur :

```
str_count(d$ville, "s")
#> [1] 0 1 2 1
```



Attention, les fonctions de `stringr` étant prévues pour fonctionner avec des expressions régulières, certains caractères n'auront pas le sens habituel dans la chaîne indiquant le motif à rechercher. Par exemple, le `.` ne sera pas un point mais le symbole représentant “n’importe quel caractère”.

La section sur les modificateurs de motifs explique comment utiliser des chaînes “classiques” au lieu d’expressions régulières.

On peut aussi utiliser `str_subset` pour ne garder d’un vecteur que les éléments correspondant au motif :

```
str_subset(d$adresse, "Libération")
#> [1] "47 ave de la Libération"      "221 avenue de la Libération"
```

11.7 Extraire des motifs

`str_extract` permet d’extraire les valeurs correspondant à un motif. Si on lui passe comme motif une chaîne de caractère, cela aura peu d’intérêt :

```
str_extract(d$adresse, "Libération")
#> [1] NA          "Libération" NA      "Libération"
```

C’est tout de suite plus intéressant si on utilise des expressions régulières. Par exemple la commande suivante permet d’isoler les numéros de rue.

```
str_extract(d$adresse, "^\\d+")
#> [1] "3"    "47"   "12"   "221"
```

`str_extract` ne récupère que la première occurrence du motif. Si on veut toutes les extraire on peut utiliser `str_extract_all`. Ainsi, si on veut extraire l’ensemble des nombres présents dans les adresses :

```
str_extract_all(d$adresse, "\\d+")
#> [[1]]
#> [1] "3"
#>
#> [[2]]
#> [1] "47"
#>
#> [[3]]
#> [1] "12"   "17"   "1961"
#>
#> [[4]]
#> [1] "221"
```



Si on veut faire de l’extraction de groupes dans des expressions régulières (identifiés avec des parenthèses), on pourra utiliser `str_match`.

À noter que si on souhaite extraire des valeurs d'une colonne texte d'un tableau de données pour créer de nouvelles variables, on pourra utiliser la fonction `extract` de l'extension `tidyR`, décrite section 12.3.6.

Par exemple :

```
library(tidyR)
d %>% extract(adresse, "type_rue", "^\d+ (.*) ", remove = FALSE)
#> # A tibble: 4 x 4
#>   nom             adresse          type_rue ville
#>   <chr>           <chr>           <chr>     <chr>
#> 1 Mr Félicien Machin 3 rue des Fleurs    rue      Nouméa
#> 2 Mme Raymonde Bidule 47 ave de la Libération    ave      Marseille
#> 3 M. Martial Truc   12 rue du 17 octobre 1961    rue      Vénissieux
#> 4 Mme Huguette Chose 221 avenue de la Libération avenue Marseille
```

11.8 Remplacer des motifs

La fonction `str_replace_all` permet de remplacer une chaîne ou un motif par une autre.

Par exemple, on peut remplacer les occurrence de “Mr” par “M.” dans notre variable `nom` :

```
str_replace_all(d$nom, "Mr", "M.")
#> [1] "M. Félicien Machin"  "Mme Raymonde Bidule" "M. Martial Truc"
#> [4] "Mme Huguette Chose"
```

On peut également spécifier plusieurs remplacements en une seule fois :

```
str_replace_all(
  d$adresse,
  c("avenue" = "Avenue", "ave" = "Avenue", "rue" = "Rue")
)
#> [1] "3 Rue des Fleurs"           "47 Avenue de la Libération"
#> [3] "12 Rue du 17 octobre 1961" "221 Avenue de la Libération"
```

11.9 Modificateurs de motifs

Par défaut, les motifs passés aux fonctions comme `str_detect`, `str_extract` ou `str_replace` sont des expressions régulières classiques.

On peut spécifier qu'un motif n'est pas une expression régulière mais une chaîne de caractères normale en lui appliquant la fonction `fixed`. Par exemple, si on veut compter le nombre de points dans les noms de notre tableau, le paramétrage par défaut ne fonctionnera pas car dans une expression régulière le `.` est un symbole signifiant “n’importe quel caractère” :

```
str_count(d$nom, ".")
#> [1] 18 19 15 18
```

Il faut donc spécifier que notre point est bien un point avec `fixed` :

```
str_count(d$nom, fixed("."))  
#> [1] 0 0 1 0
```

On peut aussi modifier le comportement des expressions régulières à l'aide de la fonction `regex`. On peut ainsi rendre les motifs insensibles à la casse avec `ignore_case` :

```
str_detect(d$nom, "mme")  
#> [1] FALSE FALSE FALSE FALSE
```

```
str_detect(d$nom, regex("mme", ignore_case = TRUE))  
#> [1] FALSE TRUE FALSE TRUE
```

On peut également permettre aux regex d'être multilignes avec l'option `multiline = TRUE`, etc.

11.10 Ressources

L'ouvrage *R for Data Science*, accessible en ligne, contient [un chapitre entier](#) sur les chaînes de caractères et les expressions régulières (en anglais).

Le [site officiel de stringr](#) contient une [liste des fonctions](#) et les pages d'aide associées, ainsi qu'un [article dédié aux expressions régulières](#).

Pour des besoins plus pointus, on pourra aussi utiliser [l'extension stringi](#) sur laquelle est elle-même basée `stringr`.

11.11 Exercices

Dans ces exercices on utilise un tableau `d`, généré par le code suivant :

```
d <- tibble(  
  nom = c("M. rené Bézigue", "Mme Paulette fouchin", "Mme yvonne duluc", "M. Jean-Yves Pernoud"),  
  naissance = c("18/04/1937 Vesoul", "En 1947 à Grenoble (38)", "Le 5 mars 1931 à Bar-le-Duc", "Marseille"),  
  profession = c("Ouvrier agric", "ouvrière qualifiée", "Institutrice", "Exploitant agric"))
```

nom	naissance	profession
M. rené Bézigue	18/04/1937 Vesoul	Ouvrier agric
Mme Paulette fouchin	En 1947 à Grenoble (38)	ouvrière qualifiée
Mme yvonne duluc	Le 5 mars 1931 à Bar-le-Duc	Institutrice
M. Jean-Yves Pernoud	Marseille, juin 1938	Exploitant agric

Exercice 1

Capitalisez les noms des personnes avec `str_to_title` :

```
#> [1] "M. René Bézigue"      "Mme Paulette Fouchin" "Mme Yvonne Duluc"  
#> [4] "M. Jean-Yves Pernoud"
```

Exercice 2

Dans la variable `profession`, remplacer toutes les occurrences de l'abréviation "agric" par "agricole" :

```
#> [1] "Ouvrier agricole"    "ouvrière qualifiée"  "Institutrice"
#> [4] "Exploitant agricole"
```

Exercice 3

À l'aide de `str_detect`, identifier les personnes de catégorie professionnelle "Ouvrier". Indication : pensez au modificateur `ignore_case`.

```
#> [1] TRUE  TRUE FALSE FALSE
```

Exercice 4

À l'aide de `case_when` et de `str_detect`, créer une nouvelle variable `sexe` identifiant le sexe de chaque personne en fonction de la présence de M. ou de Mme dans son nom.

```
#> # A tibble: 4 x 4
#>   nom              naissance      profession      sexe
#>   <chr>            <chr>          <chr>          <chr>
#> 1 M. rené Bézigue  18/04/1937 Vesoul        Ouvrier agric  Homme
#> 2 Mme Paulette fouchin En 1947 à Grenoble (38)  ouvrière qualifiée Femme
#> 3 Mme yvonne duluc   Le 5 mars 1931 à Bar-le-Duc Institutrice  Femme
#> 4 M. Jean-Yves Pernoud Marseille, juin 1938     Exploitant agric  Homme
```

Exercice 5

Extraire l'année de naissance de chaque individu avec `str_extract`. Vous pouvez utiliser le regex "`\d\d\d\d`" qui permet d'identifier les nombres de quatre chiffres.

Vous devez obtenir le vecteur suivant :

```
#> [1] "1937" "1947" "1931" "1938"
```

À l'aide de la fonction `extract` de l'extension `tidyverse` et du regex précédent, créez une nouvelle variable `annee` dans le tableau, qui contient l'année de naissance (pour plus d'informations sur `extract`, voir la section 12.3.6).

```
#> # A tibble: 4 x 4
#>   nom              naissance      annee  profession
#>   <chr>            <chr>          <dbl>  <chr>
#> 1 M. rené Bézigue  18/04/1937 Vesoul        1937  Ouvrier agric
#> 2 Mme Paulette fouchin En 1947 à Grenoble (38)  1947  ouvrière qualifiée
#> 3 Mme yvonne duluc   Le 5 mars 1931 à Bar-le-Duc 1931  Institutrice
#> 4 M. Jean-Yves Pernoud Marseille, juin 1938     1938  Exploitant agric
```

Chapitre 12

Mettre en ordre avec `tidyverse`

12.1 Tidy data

Comme indiqué dans la section [6.3](#), les extensions du *tidyverse* comme `dplyr` ou `ggplot2` partent du principe que les données sont “bien rangées” sous forme de *tidy data*.

Prenons un exemple avec les données suivantes, qui indique la population de trois pays pour quatre années différentes :

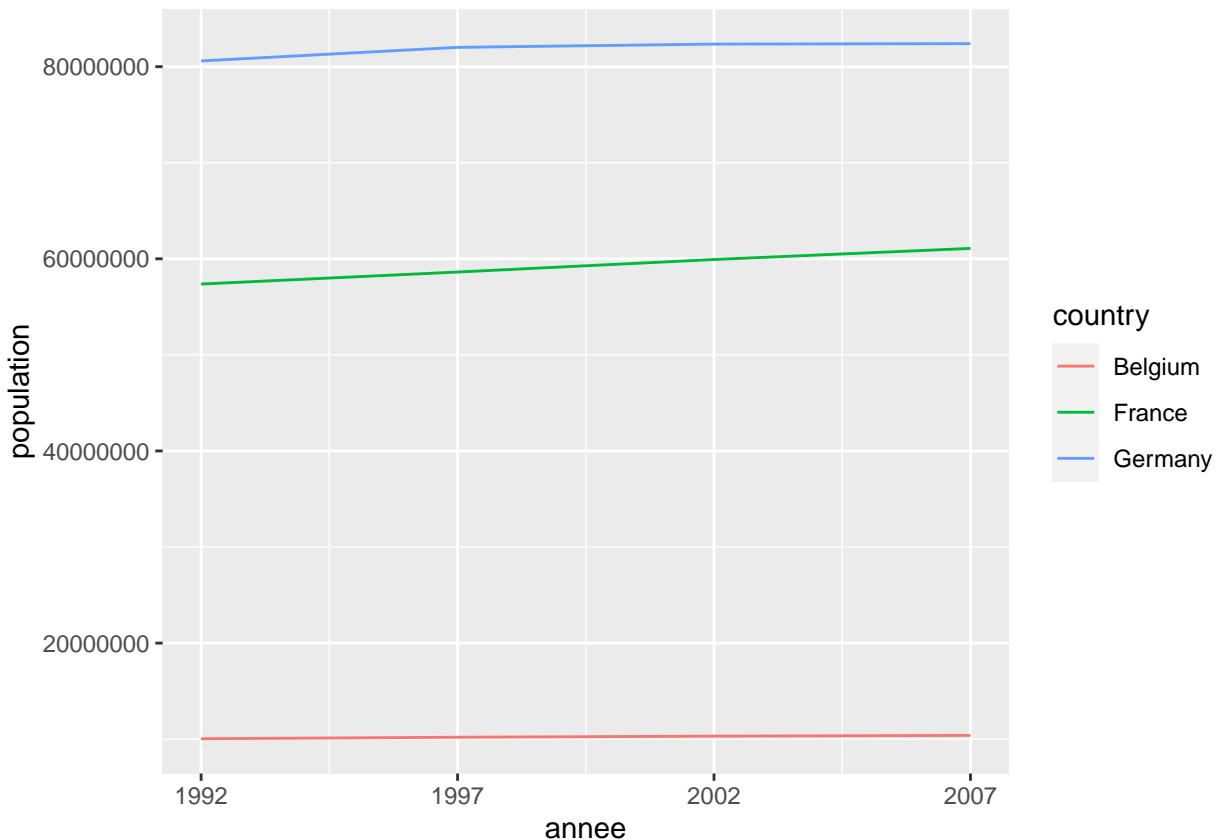
country	1992	1997	2002	2007
Belgium	10045622	10199787	10311970	10392226
France	57374179	58623428	59925035	61083916
Germany	80597764	82011073	82350671	82400996

Imaginons qu’on souhaite représenter avec `ggplot2` l’évolution de la population pour chaque pays sous forme de lignes : c’est impossible avec les données sous ce format. On a besoin d’arranger le tableau de la manière suivante :

country	annee	population
Belgium	1992	10045622
Belgium	1997	10199787
Belgium	2002	10311970
Belgium	2007	10392226
France	1992	57374179
France	1997	58623428
France	2002	59925035
France	2007	61083916
Germany	1992	80597764
Germany	1997	82011073
Germany	2002	82350671
Germany	2007	82400996

C’est seulement avec les données dans ce format qu’on peut réaliser le graphique :

```
ggplot(d) +
  geom_line(aes(x = annee, y = population, color = country)) +
  scale_x_continuous(breaks = unique(d$annee))
```



C'est la même chose pour `dplyr`, par exemple si on voulait calculer la population minimale pour chaque pays avec `summarise` :

```
d %>%
  group_by(country) %>%
  summarise(pop_min = min(population))
#> # A tibble: 3 x 2
#>   country  pop_min
#>   <fct>    <int>
#> 1 Belgium 10045622
#> 2 France  57374179
#> 3 Germany 80597764
```

12.2 Trois règles pour des données bien rangées

Le concept de *tidy data* repose sur trois règles interdépendantes. Des données sont considérées comme *tidy* si :

1. chaque ligne correspond à une observation
2. chaque colonne correspond à une variable
3. chaque valeur est présente dans une unique case de la table ou, de manière équivalente, des unités d'observations différentes sont présentes dans des tables différentes

Ces règles ne sont pas forcément très intuitives. De plus, il y a une infinité de manières pour un tableau de données de ne pas être *tidy*.

Prenons par exemple les règles 1 et 2 et le tableau de notre premier exemple :

country	1992	1997	2002	2007
Belgium	10045622	10199787	10311970	10392226
France	57374179	58623428	59925035	61083916
Germany	80597764	82011073	82350671	82400996

Pourquoi ce tableau n'est pas *tidy* ? Parce que si on essaie d'identifier les variables mesurées dans le tableau, il y en a trois : le pays, l'année et la population. Or elles ne correspondent pas aux colonnes de la table. C'est le cas par contre pour la table transformée :

country	annee	population
Belgium	1992	10045622
Belgium	1997	10199787
Belgium	2002	10311970
Belgium	2007	10392226
France	1992	57374179
France	1997	58623428
France	2002	59925035
France	2007	61083916
Germany	1992	80597764
Germany	1997	82011073
Germany	2002	82350671
Germany	2007	82400996

On peut remarquer qu'en modifiant notre table pour satisfaire à la deuxième règle, on a aussi réglé la première : chaque ligne correspond désormais à une observation, en l'occurrence l'observation de trois pays à plusieurs moments dans le temps. Dans notre table d'origine, chaque ligne comportait en réalité quatre observations différentes.

Ce point permet d'illustrer le fait que les règles sont interdépendantes.

Autre exemple, généré depuis le jeu de données `nycflights13`, permettant cette fois d'illustrer la troisième règle :

year	month	day	dep_time	carrier	name
2013	1	1	517	UA	United Air Lines Inc.
2013	1	1	533	UA	United Air Lines Inc.
2013	1	1	542	AA	American Airlines Inc.
2013	1	1	554	UA	United Air Lines Inc.
2013	1	1	558	AA	American Airlines Inc.
2013	1	1	558	UA	United Air Lines Inc.
2013	1	1	558	UA	United Air Lines Inc.
2013	1	1	559	AA	American Airlines Inc.

Dans ce tableau on a bien une observation par ligne (un vol), et une variable par colonne. Mais on a une "infraction" à la troisième règle, qui est que chaque valeur doit être présente dans une unique case : si on regarde la colonne `name`, on a en effet une duplication de l'information concernant le nom des compagnies aériennes. Notre tableau mèle en fait deux types d'observations différents : des observations sur les vols, et des observations sur les compagnies aériennes.

Pour "arranger" ce tableau, il faut séparer les deux types d'observations en deux tables différentes :

year	month	day	dep_time	carrier
2013	1	1	517	UA
2013	1	1	533	UA
2013	1	1	542	AA
2013	1	1	554	UA
2013	1	1	558	AA
2013	1	1	558	UA
2013	1	1	558	UA
2013	1	1	559	AA

carrier	name
UA	United Air Lines Inc.
AA	American Airlines Inc.

On a désormais deux tables distinctes, l'information n'est pas dupliquée, et on peut facilement faire une jointure si on a besoin de récupérer l'information d'une table dans une autre.

12.3 Les verbes de `tidyR`

L'objectif de `tidyR` est de fournir des fonctions pour arranger ses données et les convertir dans un format *tidy*. Ces fonctions prennent la forme de verbes qui viennent compléter ceux de `dplyR` et s'intègrent parfaitement dans les séries de *pipes* (`%>%`), les *pipelines*, permettant d'enchaîner les opérations.

12.3.1 `pivot_longer` : transformer des colonnes en lignes

Prenons le tableau `d` suivant, qui liste la population de 6 pays en 2002 et 2007 :

country	2002	2007
Belgium	10311970	10392226
France	59925035	61083916
Germany	82350671	82400996
Italy	57926999	58147733
Spain	40152517	40448191
Switzerland	7361757	7554661

Dans ce tableau, une même variable (la population) est répartie sur plusieurs colonnes, chacune représentant une observation à un moment différent. On souhaite que la variable ne représente plus qu'une seule colonne, et que les observations soient réparties sur plusieurs lignes.

Pour cela on va utiliser la fonction `pivot_longer`¹ :

```
d %>% pivot_longer(c(`2002`, `2007`))
#> # A tibble: 12 x 3
#>   country     name   value
#>   <fct>      <chr>  <int>
#> 1 Belgium    2002  10311970
#> 2 Belgium    2007  10392226
#> 3 France     2002  59925035
#> 4 France     2007  61083916
#> 5 Germany    2002  82350671
#> 6 Germany    2007  82400996
#> 7 Italy      2002  57926999
#> 8 Italy      2007  58147733
#> 9 Spain      2002  40152517
#> 10 Spain     2007  40448191
#> 11 Switzerland 2002  7361757
#> 12 Switzerland 2007  7554661
```

La fonction `pivot_longer` prend comme premier argument la liste des colonnes à rassembler (ici on a mis 2002 et 2007 entre *backticks* (`2002`)) pour indiquer à `pivot_longer` qu'il s'agit d'un nom de colonne et pas d'un nombre). Ces colonnes peuvent être spécifiées avec la même syntaxe que celle de la fonction `select` de `dplyR`.

Par exemple, il est parfois plus rapide d'indiquer à `pivot_longer` les colonnes qu'on ne souhaite pas "rassembler". On peut le faire avec la syntaxe suivante :

```
d %>% pivot_longer(-country)
#> # A tibble: 12 x 3
#>   country     name   value
#>   <fct>      <chr>  <int>
#> 1 Belgium    2002  10311970
#> 2 Belgium    2007  10392226
#> 3 France     2002  59925035
#> 4 France     2007  61083916
```

¹`pivot_longer` et `pivot_wider` ont été introduites dans la version 1.0 de `tidyR`. Elles ont alors remplacé `gather` et `spread`.

```
#> 5 Germany 2002 82350671
#> 6 Germany 2007 82400996
#> 7 Italy 2002 57926999
#> 8 Italy 2007 58147733
#> 9 Spain 2002 40152517
#> 10 Spain 2007 40448191
#> 11 Switzerland 2002 7361757
#> 12 Switzerland 2007 7554661
```

Par défaut, les colonnes qui contiennent les noms des colonnes d'origine et leurs valeurs sont nommées `name` et `value`. Si cela ne convient pas, on peut indiquer les noms à utiliser via les arguments `names_to` et `values_to` :

```
d %>% pivot_longer(
  c(`2002`, `2007`),
  names_to = "annee",
  values_to = "population"
)
#> # A tibble: 12 x 3
#>   country     annee population
#>   <fct>     <chr>    <int>
#> 1 Belgium 2002 10311970
#> 2 Belgium 2007 10392226
#> 3 France 2002 59925035
#> 4 France 2007 61083916
#> 5 Germany 2002 82350671
#> 6 Germany 2007 82400996
#> 7 Italy 2002 57926999
#> 8 Italy 2007 58147733
#> 9 Spain 2002 40152517
#> 10 Spain 2007 40448191
#> 11 Switzerland 2002 7361757
#> 12 Switzerland 2007 7554661
```

Au final, le nom de `pivot_longer` s'explique par le fait qu'on fait “pivoter” notre tableau de départ d'un format “large” (avec plus de colonnes) vers un format “long” (avec plus de lignes).

12.3.2 `pivot_wider` : transformer des lignes en colonnes

La fonction `pivot_wider` est l'inverse de `pivot_longer`.

Soit le tableau `d` suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	lifeExp	79.441
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	lifeExp	79.590
France	Europe	2002	pop	59925035.000
France	Europe	2007	lifeExp	80.657
France	Europe	2007	pop	61083916.000

Ce tableau a le problème inverse du précédent : on a deux variables, `lifeExp` et `pop` qui, plutôt que d'être réparties en deux colonnes, sont réparties entre plusieurs lignes.

On va donc utiliser `pivot_wider` pour répartir ces lignes dans deux colonnes différentes :

```
d %>% pivot_wider(names_from = variable, values_from = value)
#> # A tibble: 4 x 5
#>   country continent  year lifeExp      pop
#>   <fct>    <fct>    <int>    <dbl>    <dbl>
#> 1 Belgium Europe    2002     78.3 10311970
#> 2 Belgium Europe    2007     79.4 10392226
#> 3 France  Europe    2002     79.6 59925035
#> 4 France  Europe    2007     80.7 61083916
```

`pivot_wider` prend deux arguments principaux :

- `names_from` indique la colonne contenant les noms des nouvelles variables à créer
- `values_from` indique la colonne contenant les valeurs de ces variables

Il peut arriver que certaines variables soient absentes pour certaines observations. Dans ce cas l'argument `values_fill` permet de spécifier la valeur à utiliser pour ces données manquantes (par défaut les valeurs absentes sont, logiquement, indiquées par des NA).

Exemple avec le tableau d suivant :

country	continent	year	variable	value
Belgium	Europe	2002	lifeExp	78.320
Belgium	Europe	2002	pop	10311970.000
Belgium	Europe	2007	lifeExp	79.441
Belgium	Europe	2007	pop	10392226.000
France	Europe	2002	lifeExp	79.590
France	Europe	2002	pop	59925035.000
France	Europe	2007	lifeExp	80.657
France	Europe	2007	pop	61083916.000
France	Europe	2002	density	94.000

```
d %>%
  pivot_wider(names_from = variable, values_from = value)
#> # A tibble: 4 x 6
#>   country continent  year lifeExp      pop density
#>   <chr>    <chr>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1 Belgium Europe    2002     78.3 10311970      NA
#> 2 Belgium Europe    2007     79.4 10392226      NA
#> 3 France  Europe    2002     79.6 59925035      94
#> 4 France  Europe    2007     80.7 61083916      NA
```

```
d %>%
  pivot_wider(
    names_from = variable, values_from = value,
    values_fill = list(value = 0)
  )
#> # A tibble: 4 x 6
#>   country continent  year lifeExp      pop density
#>   <chr>    <chr>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1 Belgium Europe    2002     78.3 10311970      0
#> 2 Belgium Europe    2007     79.4 10392226      0
#> 3 France  Europe    2002     79.6 59925035      94
#> 4 France  Europe    2007     80.7 61083916      0
```

Au final, le nom de `pivot_wider` s'explique par le fait qu'on fait “pivoter” notre tableau de départ d'un format “long” (avec plus de lignes) vers un format “large” (avec plus de colonnes).

12.3.3 `separate` : séparer une colonne en plusieurs colonnes

Parfois on a plusieurs informations réunies en une seule colonne et on souhaite les séparer. Soit le tableau d'exemple caricatural suivant, nommé `df` :

eleve	note
Félicien Machin	5/20
Raymonde Bidule	6/10
Martial Truc	87/100

`separate` permet de séparer la colonne `note` en deux nouvelles colonnes `note` et `note_sur` :

```
df %>% separate(note, c("note", "note_sur"))
#> # A tibble: 3 x 3
#>   eleve           note  note_sur
#>   <chr>          <chr> <chr>
#> 1 Félicien Machin 5     20
#> 2 Raymonde Bidule 6     10
#> 3 Martial Truc    87    100
```

`separate` prend deux arguments principaux, le nom de la colonne à séparer et un vecteur indiquant les noms des nouvelles variables à créer. Par défaut `separate` “sépare” au niveau des caractères non-alphanumérique (espace, symbole, etc.). On peut lui indiquer explicitement le caractère sur lequel séparer avec l'argument `sep` :

```
df %>% separate(eleve, c("prenom", "nom"), sep = " ")
#> # A tibble: 3 x 3
#>   prenom  nom   note
#>   <chr>   <chr> <chr>
#> 1 Félicien Machin 5/20
#> 2 Raymonde Bidule 6/10
#> 3 Martial Truc  87/100
```

12.3.4 `separate_rows` : séparer une colonne en plusieurs lignes

`separate_rows` est également utile quand plusieurs informations différentes ont été réunies dans une seule colonne. Contrairement à `separate`, elle ne répartit les différentes valeurs dans plusieurs colonnes, mais dans plusieurs lignes.

Prenons l'exemple suivant, où la colonne `notes` contient plusieurs notes séparées par des virgules :

eleve	classe	notes
Félicien Machin	6e	5,16,11
Raymonde Bidule	5e	15
Martial Truc	6e	11,17

Si on applique `separate_rows` à la colonne `notes`, chaque note se retrouve dans une ligne différente (et les autres colonnes sont dupliquées) :

```
separate_rows(df, notes)
#> # A tibble: 6 x 3
#>   eleve      classe notes
#>   <chr>      <chr>  <chr>
#> 1 Félicien Machin 6e    5
#> 2 Félicien Machin 6e    16
#> 3 Félicien Machin 6e    11
#> 4 Raymonde Bidule 5e    15
#> 5 Martial Truc   6e    11
#> 6 Martial Truc   6e    17
```

Par défaut `separate_rows` sépare les valeurs dès qu'elle trouve un caractère qui ne soit ni un chiffre ni une lettre, mais on peut spécifier le séparateur à l'aide de l'argument `sep` (qui accepte une chaîne de caractère ou même une expression régulière) :

```
separate_rows(df, notes, sep = ",")
```

12.3.5 `unite` : regrouper plusieurs colonnes en une seule

`unite` est l'opération inverse de `separate`. Elle permet de regrouper plusieurs colonnes en une seule. Imaginons qu'on obtient le tableau `d` suivant :

code_departement	code_commune	commune	pop_tot
01	004	Ambérieu-en-Bugey	14204
01	007	Ambronay	2763
01	014	Arbent	3356
01	024	Attignat	3196
01	025	Bâgé-Dommartin	4078
01	027	Balan	2513

On souhaite reconstruire une colonne `code_insee` qui indique le code INSEE de la commune, et qui s'obtient en concaténant le code du département et celui de la commune. On peut utiliser `unite` pour cela :

```
d %>% unite(code_insee, code_departement, code_commune)
#> # A tibble: 6 x 3
#>   code_insee  commune      pop_tot
#>   <chr>       <chr>        <dbl>
#> 1 01_004     Ambérieu-en-Bugey 14204
#> 2 01_007     Ambronay        2763
#> 3 01_014     Arbent          3356
#> 4 01_024     Attignat        3196
#> 5 01_025     Bâgé-Dommartin 4078.
#> 6 01_027     Balan           2513
```

Le résultat n'est pas idéal : par défaut `unite` ajoute un caractère `_` entre les deux valeurs concaténées, alors qu'on ne veut aucun séparateur. De plus, on souhaite conserver nos deux colonnes d'origine, qui peuvent nous être utiles. On peut résoudre ces deux problèmes à l'aide des arguments `sep` et `remove` :

```
d %>%
  unite(
    code_insee, code_departement, code_commune,
    sep = "", remove = FALSE
  )
#> # A tibble: 6 x 5
#>   code_insee  code_departement code_commune commune      pop_tot
#>   <chr>       <chr>        <chr>       <chr>        <dbl>
#> 1 01004      01            004         Ambérieu-en-Bugey 14204
#> 2 01007      01            007         Ambronay        2763
#> 3 01014      01            014         Arbent          3356
#> 4 01024      01            024         Attignat        3196
#> 5 01025      01            025         Bâgé-Dommartin 4078.
#> 6 01027      01            027         Balan           2513
```

12.3.6 `extract` : créer de nouvelles colonnes à partir d'une colonne de texte

`extract` permet de créer de nouvelles colonnes à partir de sous-chaînes d'une colonne de texte existante, identifiées par des groupes dans une expression régulière.

Par exemple, à partir du tableau suivant :

eleve	note
Félicien Machin	5/20
Raymonde Bidule	6/10
Martial Truc	87/100

On peut extraire les noms et prénoms dans deux nouvelles colonnes avec :

```
df %>%
  extract(
    eleve,
    c("prenom", "nom"),
    "^(.*)(.*)$"
  )
#> # A tibble: 3 x 3
#>   prenom nom   note
#>   <chr>   <chr> <chr>
#> 1 Félicien Machin 5/20
#> 2 Raymonde Bidule 6/10
#> 3 Martial Truc  87/100
```

On passe donc à `extract` trois arguments : la colonne d'où on doit extraire les valeurs, un vecteur avec les noms des nouvelles colonnes à créer, et une expression régulière comportant autant de groupes (identifiés par des parenthèses) que de nouvelles colonnes.

Par défaut la colonne d'origine n'est pas conservée dans la table résultat. On peut modifier ce comportement avec l'argument `remove = FALSE`. Ainsi, le code suivant extrait les initiales du prénom et du nom mais conserve la colonne d'origine :

```
df %>%
  extract(
    eleve,
    c("initiale_prenom", "initiale_nom"),
    "^(.).*(.).*$",
    remove = FALSE
  )
#> # A tibble: 3 x 4
#>   eleve      initiale_prenom initiale_nom note
#>   <chr>      <chr>          <chr>        <chr>
#> 1 Félicien F            M        5/20
#> 2 Raymonde R           B        6/10
#> 3 Martial T            T        87/100
```

12.3.7 `complete` : compléter des combinaisons de variables manquantes

Imaginons qu'on ait le tableau de résultats suivants :

eleve	matiere	note
Alain	Maths	16
Alain	Français	9
Barnabé	Maths	17
Chantal	Français	11

Les élèves Barnabé et Chantal n'ont pas de notes dans toutes les matières. Supposons que c'est parce qu'ils étaient absents et que leur note est en fait un 0. Si on veut calculer les moyennes des élèves, on doit compléter ces notes manquantes.

La fonction `complete` est prévue pour ce cas de figure : elle permet de compléter des combinaisons manquantes de valeurs de plusieurs colonnes.

On peut l'utiliser de cette manière :

```
df %>% complete(eleve, matiere)
#> # A tibble: 6 x 3
#>   eleve    matiere    note
#>   <chr>    <chr>     <dbl>
#> 1 Alain    Français    9
#> 2 Alain    Maths      16
#> 3 Barnabé Français   NA
#> 4 Barnabé Maths       17
#> 5 Chantal   Français   11
#> 6 Chantal   Maths      NA
```

On voit que les combinaisons manquante “Barnabé - Français” et “Chantal - Maths” ont bien été ajoutées par `complete`.

Par défaut les lignes insérées récupèrent des valeurs manquantes `NA` pour les colonnes restantes. On peut néanmoins choisir une autre valeur avec l'argument `fill`, qui prend la forme d'une liste nommée :

```
df %>% complete(eleve, matiere, fill = list(note = 0))
#> # A tibble: 6 x 3
#>   eleve    matiere    note
#>   <chr>    <chr>     <dbl>
#> 1 Alain    Français    9
#> 2 Alain    Maths      16
#> 3 Barnabé Français    0
#> 4 Barnabé Maths       17
#> 5 Chantal   Français   11
#> 6 Chantal   Maths      0
```

Parfois on ne souhaite pas inclure toutes les colonnes dans le calcul des combinaisons de valeurs. Par exemple, supposons qu'on rajoute dans notre tableau une colonne avec les identifiants de chaque élève :

id	eleve	matiere	note
1001001	Alain	Maths	16
1001001	Alain	Français	9
1001002	Barnabé	Maths	17
1001003	Chantal	Français	11

Si on applique `complete` comme précédemment, le résultat n'est pas bon car il contient toutes les combinaisons de `id`, `eleve` et `matiere`.

```
df %>% complete(id, eleve, matiere)
#> # A tibble: 18 x 4
#>   id     eleve    matiere    note
#>   <dbl>    <chr>    <chr>     <dbl>
#> 1 1001001 Alain    Français    9
#> 2 1001001 Alain    Maths      16
#> 3 1001001 Barnabé Français   NA
#> 4 1001001 Barnabé Maths      NA
```

```
#> 5 1001001 Chantal Français NA
#> 6 1001001 Chantal Maths NA
#> 7 1001002 Alain Français NA
#> 8 1001002 Alain Maths NA
#> 9 1001002 Barnabé Français NA
#> 10 1001002 Barnabé Maths 17
#> 11 1001002 Chantal Français NA
#> 12 1001002 Chantal Maths NA
#> 13 1001003 Alain Français NA
#> 14 1001003 Alain Maths NA
#> 15 1001003 Barnabé Français NA
#> 16 1001003 Barnabé Maths NA
#> 17 1001003 Chantal Français 11
#> 18 1001003 Chantal Maths NA
```

Dans ce cas, pour signifier à `complete` que `id` et `eleve` sont deux attributs d'un même individu et ne doivent pas être combinés entre eux, on doit les placer dans une fonction `nesting` :

```
df %>% complete(nesting(id, eleve), matiere)
#> # A tibble: 6 x 4
#>   id     eleve    matiere  note
#>   <dbl> <chr>    <chr>    <dbl>
#> 1 1001001 Alain    Français  9
#> 2 1001001 Alain    Maths     16
#> 3 1001002 Barnabé Français NA
#> 4 1001002 Barnabé Maths    17
#> 5 1001003 Chantal Français 11
#> 6 1001003 Chantal Maths   NA
```

12.4 Ressources

Chaque jeu de données est différent, et le travail de remise en forme est souvent long et plus ou moins compliqué. On n'a donné ici que les exemples les plus simples, et c'est souvent en combinant différentes opérations qu'on finit par obtenir le résultat souhaité.

Le livre *R for data science*, librement accessible en ligne, contient [un chapitre complet](#) sur la remise en forme des données.

L'article [Tidy data](#), publié en 2014 dans le *Journal of Statistical Software*, présente de manière détaillée le concept éponyme (mais il utilise des extensions désormais obsolètes qui ont depuis été remplacées par `dplyr` et `tidyverse`).

Le site de l'extension est accessible à l'adresse : <https://tidyverse.org/> et contient une liste des fonctions et les pages d'aide associées.

Chapitre 13

Diffuser et publier avec `rmarkdown`

L'extension `rmarkdown` permet de générer des documents de manière dynamique en mélangeant texte mis en forme et résultats produits par du code R. Les documents générés peuvent être au format HTML, PDF, Word, et bien d'autres¹. C'est donc un outil très pratique pour l'exportation, la communication et la diffusion de résultats d'analyse.

Le présent document a lui-même été généré à partir de fichiers R Markdown².

`rmarkdown` ne fait pas partie du *tidyverse*, mais elle est installée et chargée par défaut par RStudio³.

Voici un exemple de document R Markdown minimal :

```
---
```

```
title: "Test R Markdown"
```

```
--
```



```
*R Markdown* permet de mélanger :
```

- du texte libre mis en forme
- des blocs de code R


```
Les blocs de code sont exécutés et leur résultat affiché, par exemple :
```

```
```{r}
mean(mtcars$mpg)
````
```



```
## Graphiques
```



```
On peut également inclure des graphiques :
```

```
```{r}
plot(mtcars$hp, mtcars$mpg)
````
```

Ce document peut être “compilé” sous différents formats. Lors de cette étape, le texte est mis en forme, les blocs de code sont exécutés, leur résultat ajouté au document, et le tout est transformé dans un des différents formats possibles.

Voici le rendu du document précédent au format HTML :

¹On peut citer les formats `odt`, `rtf`, `Markdown`, etc.

²Plus précisément grâce à l'extension `bookdown` qui permet de générer des documents de type livre.

³Si vous n'utilisez pas ce dernier, l'extension peut être installée à part avec `install.packages("rmarkdown")` et chargée explicitement avec `library(rmarkdown)`.

Test R Markdown

R Markdown permet de mélanger :

- du texte libre mis en forme
- des blocs de code R

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Graphiques

On peut également inclure des graphiques :

```
plot(mtcars$hp, mtcars$mpg)
```

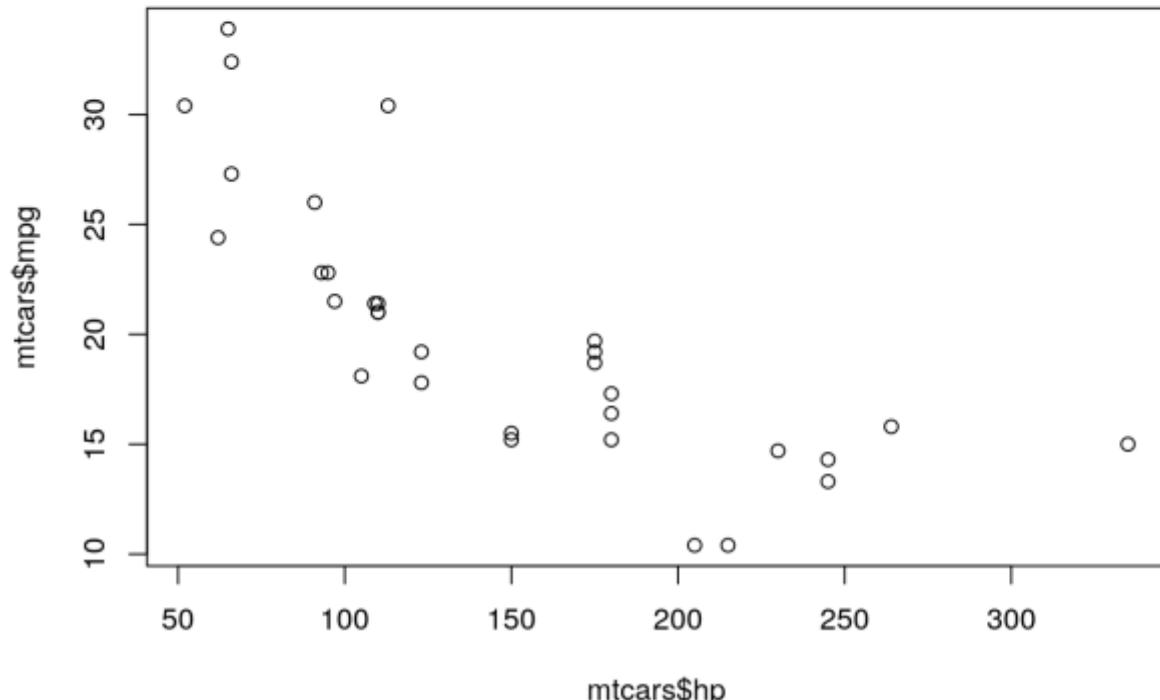


Figure 13.1: Rendu HTML

Le rendu du même document au format PDF :

Test R Markdown

R Markdown permet de mélanger :

- du texte libre mis en forme
- des blocs de code R

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Graphiques

On peut également inclure des graphiques :

```
plot(mtcars$hp, mtcars$mpg)
```

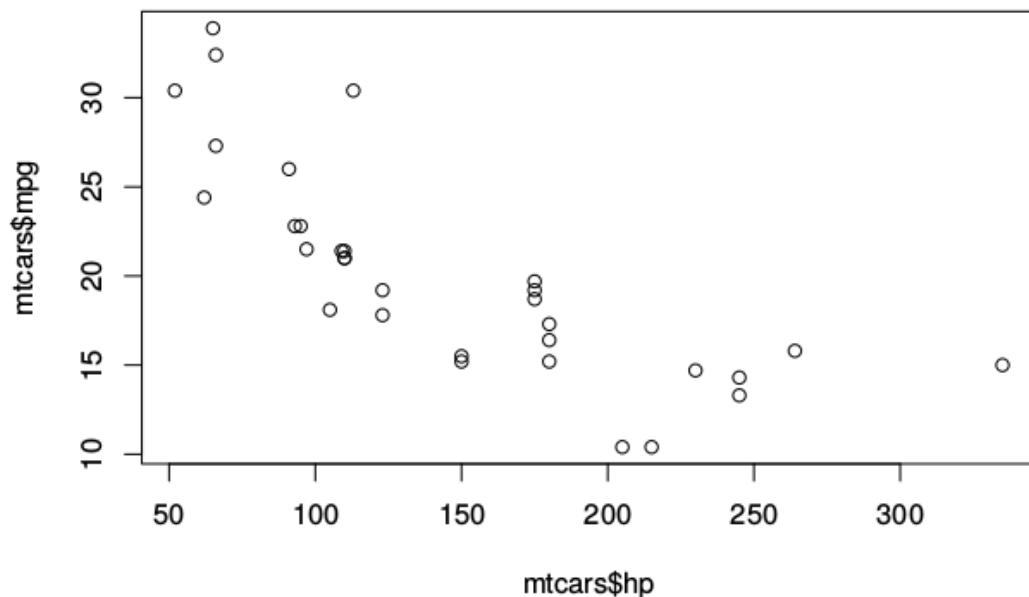


Figure 13.2: Rendu PDF

Et le rendu au format docx :

Test R Markdown

R Markdown permet de mélanger :

- du texte libre mis en forme
- des blocs de code R

Les blocs de code sont exécutés et leur résultat affiché, par exemple :

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Graphiques

On peut également inclure des graphiques :

```
plot(mtcars$hp, mtcars$mpg)
```

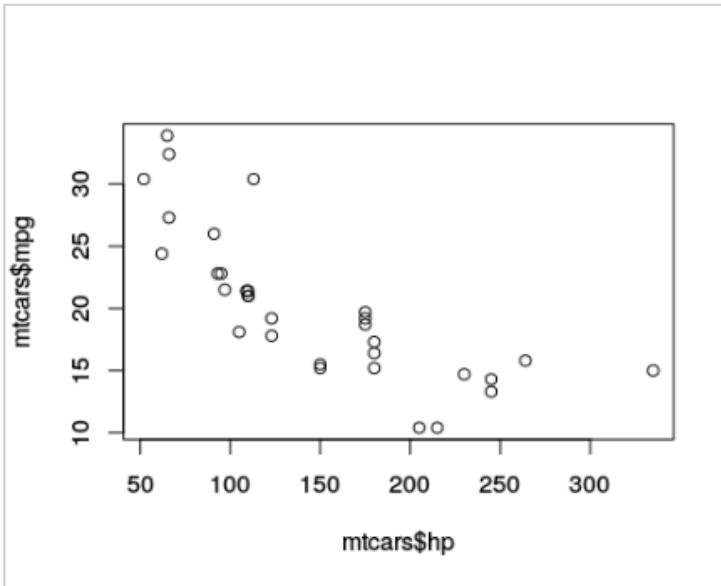


Figure 13.3: Rendu docx

Les avantages de ce système sont nombreux :

- le code et ses résultats ne sont pas séparés des commentaires qui leur sont associés
- le document final est reproductible
- le document peut être très facilement régénéré et mis à jour, par exemple si les données source ont été modifiées.

13.1 Créer un nouveau document

Un document R Markdown est un simple fichier texte enregistré avec l'extension .Rmd.

Sous RStudio, on peut créer un nouveau document en allant dans le menu *File* puis en choisissant *New file* puis *R Markdown....* La boîte de dialogue suivante s'affiche :

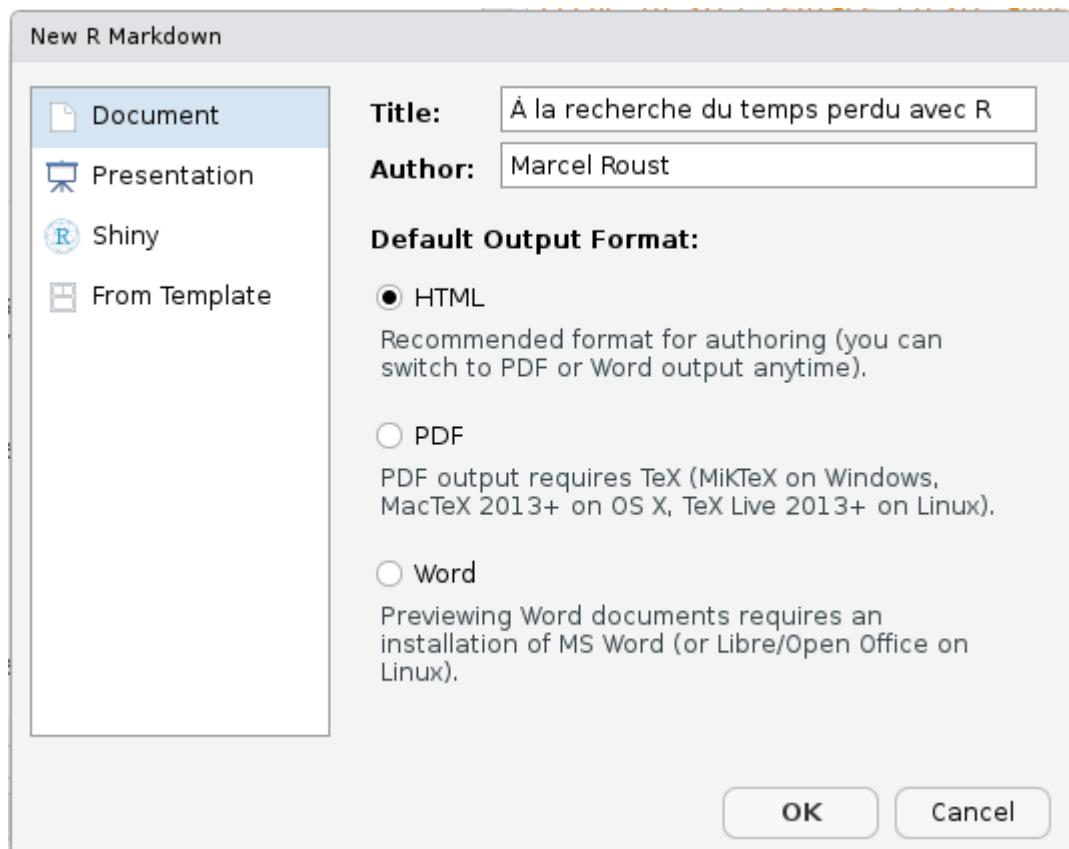


Figure 13.4: Création d'un document R Markdown

On peut indiquer le titre, l'auteur du document ainsi que le format de sortie par défaut (il est possible de modifier facilement ces éléments par la suite). Plutôt qu'un document classique, on verra section 13.6 qu'on peut aussi choisir de créer une présentation sous forme de slides (entrée *Presentation*) ou de créer un document à partir d'un modèle (Entrée *From Template*).

Un fichier comportant un contenu d'exemple s'affiche alors. Vous pouvez l'enregistrer où vous le souhaitez avec une extension `.Rmd`.

13.2 Éléments d'un document R Markdown

Un document R Markdown est donc un fichier texte qui ressemble à quelque chose comme ça :

```
---
title: "Titre"
author: "Prénom Nom"
date: "10 avril 2017"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## Introduction

Ceci est un document RMarkdown, qui mélange :
```

- du texte balisé selon la syntaxe Markdown
- des bouts de code R qui seront exécutés

Le code R se présente de la manière suivante :

```
```{r}
summary(cars)
```

## Graphiques
```

On peut aussi inclure des graphiques, par exemple :

```
```{r}
plot(pressure)
```
```

On va décomposer les différents éléments constitutifs de ce document.

13.2.1 En-tête (préambule)

La première partie du document est son *en-tête*. Il se situe en tout début de document, et est délimité par trois tirets (---) avant et après :

```
---
title: "Titre"
author: "Prénom Nom"
date: "10 avril 2017"
output: html_document
---
```

Cet en-tête contient les métadonnées du document, comme son titre, son auteur, sa date, plus tout un tas d'options possibles qui vont permettre de configurer ou personnaliser l'ensemble du document et son rendu. Ici, par exemple, la ligne `output: html_document` indique que le document généré doit être au format HTML.

13.2.2 Texte du document

Le corps du document est constitué de texte qui suit la syntaxe *Markdown*. Un fichier Markdown est un fichier texte contenant un balisage léger qui permet de définir des niveaux de titres ou de mettre en forme le texte. Par exemple, le texte suivant :

Ceci est du texte avec *de l'*italique** et **du **gras****.

On peut définir des listes à puces :

- premier élément
- deuxième élément

Génèrera le texte mis en forme suivant :

Ceci est du texte avec de l'*italique* et du **gras**.

On peut définir des listes à puces :

- premier élément
- deuxième élément

On voit que des mots placés entre des astérisques sont mis en italique, des lignes qui commencent par un tiret sont transformés en liste à puce, etc.

On peut définir des titres de différents niveaux en faisant débuter une ligne par un ou plusieurs # :

```
# Titre de niveau 1
## Titre de niveau 2
### Titre de niveau 3
```

Quand des titres ont été définis, si vous cliquez sur l'icône *Show document outline* totalement à droite de la barre d'outils associée au fichier R Markdown, une table des matières générée automatiquement à partir des titres s'affiche et vous permet de naviguer facilement dans le document :

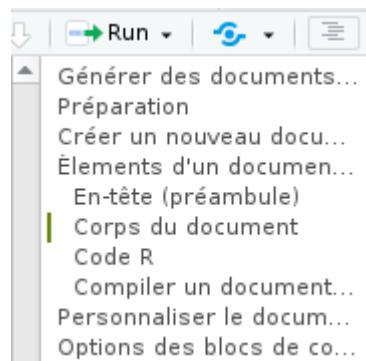


Figure 13.5: Table des matières dynamique

La syntaxe Markdown permet d'autres mises en forme, comme la possibilité d'insérer des liens ou des images. Par exemple, le code suivant :

```
[Exemple de lien](https://example.com)
```

Donnera le lien suivant :

[Exemple de lien](https://example.com)

Dans RStudio, le menu *Help* puis *Markdown quick reference* donne un aperçu plus complet de la syntaxe.

13.2.3 Blocs de code R

En plus du texte libre au format Markdown, un document R Markdown contient, comme son nom l'indique, du code R. Celui-ci est inclus dans des blocs (*chunks*) délimités par la syntaxe suivante :

```
```{r}
x <- 1:5
```
```

Comme cette suite de caractères n'est pas très simple à saisir, vous pouvez utiliser le menu *Insert* de RStudio et choisir *R*⁴, ou utiliser le raccourci clavier **Ctrl+Alt+i**.

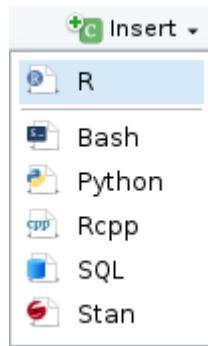


Figure 13.6: Menu d'insertion d'un bloc de code

Dans RStudio les blocs de code R sont en général affichés avec une couleur de fond légèrement différente pour les distinguer du reste du document.

Quand votre curseur se trouve dans un bloc, vous pouvez saisir le code R que vous souhaitez, l'exécuter, utiliser l'autocomplétion, exactement comme si vous vous trouviez dans un script R. Vous pouvez également exécuter l'ensemble du code contenu dans un bloc à l'aide du raccourci clavier **Ctrl+Maj+Entrée**.

Dans RStudio, par défaut, les résultats d'un bloc de code (texte, tableau ou graphique) s'affichent directement *dans* la fenêtre d'édition du document, permettant de les visualiser facilement et de les conserver le temps de la session⁵.

Lorsque le document est “compilé” au format HTML, PDF ou docx, chaque bloc est exécuté tour à tour, et le résultat inclus dans le document final, qu'il s'agisse de texte, d'un tableau ou d'un graphique. Les blocs sont liés entre eux, dans le sens où les données importées ou calculées dans un bloc sont accessibles aux blocs suivants. On peut donc aussi voir un document R Markdown comme un script R dans lequel on aurait intercalé du texte libre au format Markdown.



À noter qu'avant chaque compilation, une nouvelle session R est lancée, ne contenant aucun objet. Les premiers blocs de code d'un document sont donc souvent utilisés pour importer des données, exécuter des recodages, etc.

13.2.4 Compiler un document (*Knit*)

On peut à tout moment compiler, ou plutôt “tricoter” (*Knit*), un document R Markdown pour obtenir et visualiser le document généré. Pour cela, il suffit de cliquer sur le bouton *Knit* et de choisir le format de sortie voulu :

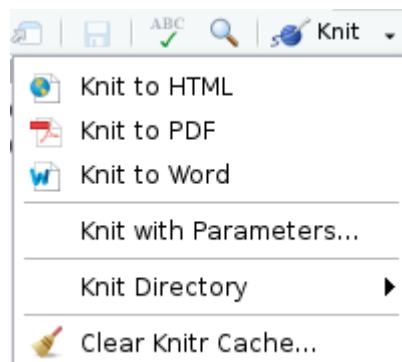


Figure 13.7: Menu *Knit*

⁴Il est possible d'inclure dans un document R Markdown des blocs de code d'autres langages

⁵Ce comportement peut être modifié en cliquant sur l'icône d'engrenage de la barre d'outils et en choisissant *Chunk Output in Console*

Vous pouvez aussi utiliser le raccourci **Ctrl+Maj+K** pour compiler le document dans le dernier format utilisé.



Pour la génération du format PDF, vous devez avoir une installation fonctionnelle de **LaTeX** sur votre système.

Si ça n'est pas le cas, l'extension **tinytex** de Yihui Xie vise à faciliter l'installation d'une distribution **LaTeX** minimale quel que soit le système d'exploitation de votre machine. Pour l'utiliser il vous faut d'abord installer l'extension avec `install.packages('tinytex')`, puis lancer la commande suivante dans la console (prévoir un téléchargement d'environ 200Mo) :

```
tinytex::install_tinytex()
```

Plus d'informations sur [le site de tinytex](#).

Un onglet *R Markdown* s'ouvre dans la même zone que l'onglet *Console* et indique la progression de la compilation, ainsi que les messages d'erreur éventuels. Si tout se passe bien, Le document devrait s'afficher soit dans une fenêtre *Viewer* de RStudio (pour la sortie HTML), soit dans le logiciel par défaut de votre ordinateur.

13.3 Personnaliser le document généré

La personnalisation du document généré se fait en modifiant des options dans le préambule du document. RStudio propose néanmoins une petite interface graphique permettant de changer ces options plus facilement. Pour cela, cliquez sur l'icône en forme d'engrenage à droite du bouton *Knit* et choisissez *Output Options...*

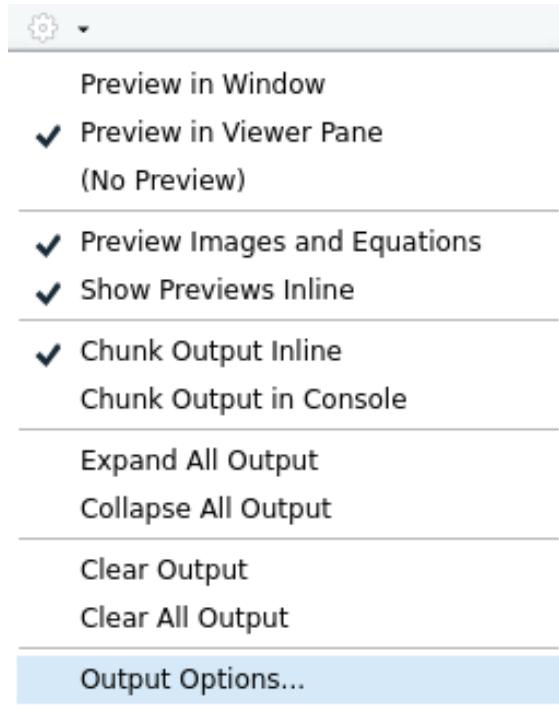


Figure 13.8: Options de sortie R Markdown

Une boîte de dialogue s'affiche vous permettant de sélectionner le format de sortie souhaité et, selon le format, différentes options :

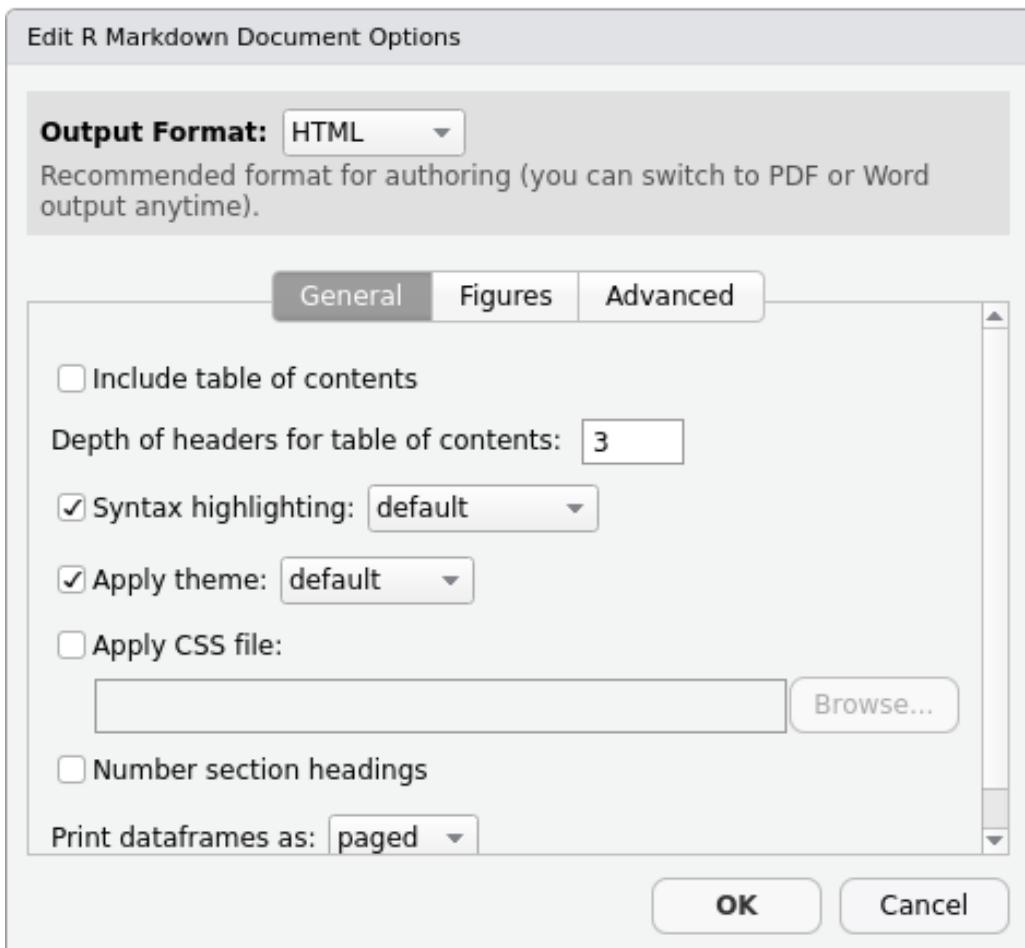


Figure 13.9: Dialogue d'options de sortie R Markdown

Pour le format HTML par exemple, l'onglet *General* vous permet de spécifier si vous voulez une table des matières, sa profondeur, les thèmes à appliquer pour le document et la coloration syntaxique des blocs R, etc. L'onglet *Figures* vous permet de changer les dimensions par défaut des graphiques générés.



Une option très intéressante pour les fichiers HTML, accessible via l'onglet *Advanced*, est l'entrée *Create standalone HTML document*. Si elle est cochée (ce qui est le cas par défaut), le document HTML généré contiendra en un seul fichier le code HTML mais aussi les images et toutes les autres ressources nécessaires à son affichage. Ceci permet de générer des fichiers (parfois assez volumineux) que vous pouvez transférer très facilement à quelqu'un par mail ou en le mettant en ligne quelque part. Si la case n'est pas cochée, les images et autres ressources sont placées dans un dossier à part.

Lorsque vous changez des options, RStudio va en fait modifier le préambule de votre document. Ainsi, si vous choisissez d'afficher une table des matières et de modifier le thème de coloration syntaxique, votre en-tête va devenir quelque chose comme :

```
---
title: "Test R Markdown"
output:
  html_document:
    highlight: kate
    toc: yes
---
```

Vous pouvez modifier les options directement en éditant le préambule.

À noter qu'il est possible de spécifier des options différentes selon les formats, par exemple :

```
---
title: "Test R Markdown"
output:
  html_document:
    highlight: kate
    toc: yes
  pdf_document:
    fig_caption: yes
    highlight: kate
---
```

La liste complète des options possibles est présente sur [le site de la documentation officielle](#) (très complet et bien fait) et sur l'antiseche et le guide de référence, accessibles depuis RStudio via le menu *Help* puis *Cheatsheets*.

13.4 Options des blocs de code R

Il est également possible de passer des options à chaque bloc de code R pour modifier son comportement.

On rappelle qu'un bloc de code se présente de la manière suivante :

```
```{r}
x <- 1:5
```
```

Les options d'un bloc de code sont à placer à l'intérieur des accolades **{r}**.

13.4.1 Nom du bloc

La première possibilité est de donner un *nom* au bloc. Celui-ci est indiqué directement après le **r** :

```
{r nom_du_bloc}
```

Il n'est pas obligatoire de nommer un bloc, mais cela peut être utile en cas d'erreur à la compilation, pour identifier le bloc ayant causé le problème. Attention, on ne peut pas avoir deux blocs avec le même nom.

13.4.2 Options

En plus d'un nom, on peut passer à un bloc une série d'options sous la forme **option = valeur**. Voici un exemple de bloc avec un nom et des options :

```
```{r mon_bloc, echo = FALSE, warning = TRUE}
x <- 1:5
```
```

Et un exemple de bloc non nommé avec des options :

```
```{r echo = FALSE, warning = FALSE}
x <- 1:5
```
```

Une des options utiles est l'option `echo`. Par défaut `echo` vaut `TRUE`, et le bloc de code R est inséré dans le document généré, de cette manière :

```
x <- 1:5
print(x)
#> [1] 1 2 3 4 5
```

Mais si on positionne l'option `echo=FALSE`, alors le code R n'est plus inséré dans le document, et seul le résultat est visible :

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

Voici une liste de quelques unes des options disponibles :

| Option | Valeurs | Description |
|----------------------|-------------------------------------|--|
| <code>echo</code> | <code>TRUE/FALSE</code> | Afficher ou non le code R dans le document |
| <code>eval</code> | <code>TRUE/FALSE</code> | Exécuter ou non le code R à la compilation |
| <code>include</code> | <code>TRUE/FALSE</code> | Inclure ou non le code R et ses résultats dans le document |
| <code>results</code> | <code>"hide"/"asis"/"markup"</code> | Résultats renvoyés par le bloc de code |
| <code>warning</code> | <code>TRUE/FALSE</code> | Afficher ou non les avertissements générés par le bloc |
| <code>message</code> | <code>TRUE/FALSE</code> | Afficher ou non les messages générés par le bloc |

Il existe de nombreuses autres options décrites notamment dans [guide de référence R Markdown](#) (PDF en anglais).

13.4.3 Modifier les options

Il est possible de modifier les options manuellement en éditant l'en-tête du bloc de code, mais on peut aussi utiliser une petite interface graphique proposée par RStudio. Pour cela, il suffit de cliquer sur l'icône d'engrenage située à droite sur la ligne de l'en-tête de chaque bloc :

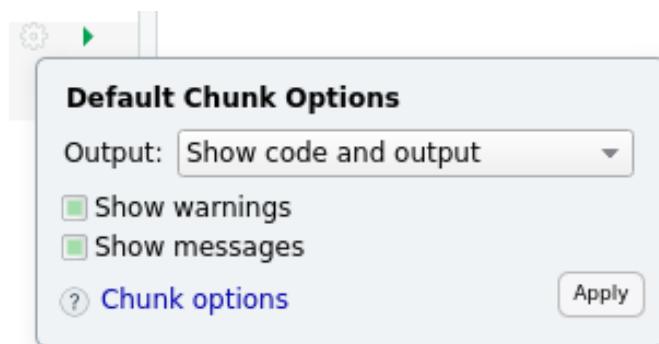


Figure 13.10: Menu d'options de bloc de code

Vous pouvez ensuite modifier les options les plus courantes, et cliquer sur *Apply* pour les appliquer.

13.4.4 Options globales

On peut vouloir appliquer une option à l'ensemble des blocs d'un document. Par exemple, on peut souhaiter par défaut ne pas afficher le code R de chaque bloc dans le document final.

On peut positionner une option globalement en utilisant la fonction `knitr::opts_chunk$set()`. Par exemple, insérer `knitr::opts_chunk$set(echo = FALSE)` dans un bloc de code positionnera l'option `echo = FALSE` par défaut pour tous les blocs suivants.

En général, on place toutes ces modifications globales dans un bloc spécial nommé `setup` et qui est le premier bloc du document :

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

```

 Par défaut RStudio exécute systématiquement le contenu du bloc `setup` avant d'exécuter celui d'un autre bloc.

Contrairement aux autres blocs de code, quand on utilise dans RStudio le menu des paramètres du bloc `setup` pour modifier ses options, celles-ci modifient non pas les options de ce bloc mais les options globales, en mettant à jour l'appel de la fonction `knitr::opts_chunk$set()`.

13.4.5 Mise en cache des résultats

Compiler un document R Markdown peut être long, car il faut à chaque fois exécuter l'ensemble des blocs de code R qui le constituent.

Pour accélérer cette opération, R Markdown utilise un système de *mise en cache* : les résultats de chaque bloc sont enregistrés dans un fichier et à la prochaine compilation, si le code et les options du bloc n'ont pas été modifiés, c'est le contenu du fichier de cache qui est utilisé, ce qui évite d'exécuter le code R.

 On peut activer ou désactiver la mise en cache des résultats pour chaque bloc de code avec l'option `cache = TRUE` ou `cache = FALSE`, et on peut aussi désactiver totalement la mise en cache pour le document en ajoutant `knitr::opts_chunk$set(cache = FALSE)` dans le premier bloc `setup`.

Ce système de cache peut poser problème par exemple si les données source changent : dans ce cas les résultats de certains blocs peuvent ne pas être mis à jour s'ils sont présents en cache. Dans ce cas, on peut vider le cache du document, ce qui forcera un recalcul de tous les blocs de code à la prochaine compilation. Pour cela, vous pouvez ouvrir le menu *Knit* et choisir *Clear Knitr Cache...* :

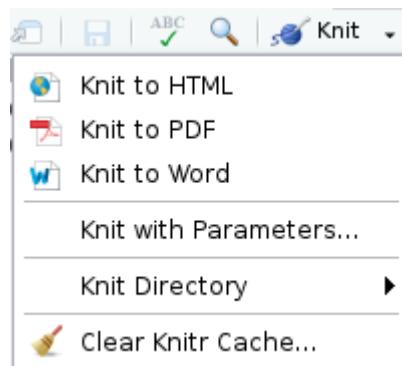


Figure 13.11: Menu *Knit*

13.5 Rendu des tableaux

13.5.1 Tableaux croisés

Par défaut, les tableaux issus de la fonction `table` sont affichés comme ils apparaissent dans la console de R, en texte brut :

```
library(questionr)
data(hdv2003)
tab <- lprop(table(hdv2003$qualif, hdv2003$sexe))
tab
#>
#>           Homme Femme Total
#> Ouvrier specialise    47.3  52.7 100.0
#> Ouvrier qualifie     78.4  21.6 100.0
#> Technicien            76.7  23.3 100.0
#> Profession intermediaire 55.0  45.0 100.0
#> Cadre                 55.8  44.2 100.0
#> Employe               16.2  83.8 100.0
#> Autre                  36.2  63.8 100.0
#> Ensemble                44.8  55.2 100.0
```

On peut améliorer leur présentation en utilisant la fonction `kable` de l'extension `knitr`. Celle-ci fournit un formatage adapté en fonction du format de sortie. On aura donc des tableaux “propres” que ce soit en HTML, PDF ou aux formats traitements de texte :

```
library(knitr)
kable(tab)
```

| | Homme | Femme | Total |
|--------------------------|----------|----------|-------|
| Ouvrier specialise | 47.29064 | 52.70936 | 100 |
| Ouvrier qualifie | 78.42466 | 21.57534 | 100 |
| Technicien | 76.74419 | 23.25581 | 100 |
| Profession intermediaire | 55.00000 | 45.00000 | 100 |
| Cadre | 55.76923 | 44.23077 | 100 |
| Employe | 16.16162 | 83.83838 | 100 |
| Autre | 36.20690 | 63.79310 | 100 |
| Ensemble | 44.82759 | 55.17241 | 100 |

Différents arguments permettent de modifier la sortie de `kable`. `digits`, par exemple, permet de spécifier le nombre de chiffres significatifs à afficher dans les colonnes de nombres :

```
kable(tab, digits = 1)
```

| | Homme | Femme | Total |
|--------------------------|-------|-------|-------|
| Ouvrier specialise | 47.3 | 52.7 | 100 |
| Ouvrier qualifie | 78.4 | 21.6 | 100 |
| Technicien | 76.7 | 23.3 | 100 |
| Profession intermediaire | 55.0 | 45.0 | 100 |
| Cadre | 55.8 | 44.2 | 100 |
| Employe | 16.2 | 83.8 | 100 |
| Autre | 36.2 | 63.8 | 100 |
| Ensemble | 44.8 | 55.2 | 100 |

13.5.2 Tableaux de données et tris à plat

En ce qui concerne les tableaux de données (tibble ou *data frame*), l'affichage HTML par défaut se contente d'un affichage texte comme dans la console, très peu lisible dès que le tableau dépasse une certaine taille.

Une alternative est d'utiliser la fonction `paged_table`, qui affiche une représentation HTML paginée du tableau :

| <code>id</code> | <code>...</code> | <code>sexe</code> | <code>nivetud</code> | <code>poids</code> | ▶ |
|-----------------|------------------|-------------------|--|--------------------|---|
| <int> | <int> | <fctr> | <fctr> | <dbl> | |
| 1 | 1 | 28 | Femme Enseignement supérieur y compris technique supérieur | 2634.39822 | |
| 2 | 2 | 23 | Femme NA | 9738.39578 | |
| 3 | 3 | 59 | Homme Dernière année d'études primaires | 3994.10246 | |
| 4 | 4 | 34 | Homme Enseignement supérieur y compris technique supérieur | 5731.66151 | |
| 5 | 5 | 71 | Femme Dernière année d'études primaires | 4329.09400 | |
| 6 | 6 | 35 | Femme Enseignement technique ou professionnel court | 8674.69938 | |
| 7 | 7 | 60 | Femme Dernière année d'études primaires | 6165.80349 | |
| 8 | 8 | 47 | Homme Enseignement technique ou professionnel court | 12891.64076 | |
| 9 | 9 | 20 | Femme NA | 7808.87206 | |
| 10 | 10 | 28 | Homme Enseignement technique ou professionnel long | 2277.16047 | |

1-10 of 2,000 rows | 1-6 of 21 columns Previous **1** 2 3 4 5 6 ... 200 Next

Figure 13.12: Rendu d'une table par `paged_table`

Une alternative est d'utiliser `kable`, comme précédemment pour les tableaux croisés, ou encore la fonction `datatable` de l'extension DT, qui propose encore davantage d'interactivité :

| Show 10 ▾ entries | | | | | | Search: <input type="text"/> |
|-------------------|--------------------|---------------------|--|----------------------|-----------------------|------------------------------|
| <code>id</code> ▾ | <code>age</code> ▾ | <code>sexe</code> ▾ | <code>nivetud</code> | <code>poids</code> ▾ | <code>occup</code> ▾ | ◀ |
| 1 | 1 | 28 | Femme Enseignement supérieur y compris technique supérieur | 2634.3982157 | Exerce une profession | |
| 2 | 2 | 23 | Femme | 9738.3957759 | Etudiant, élève | |
| 3 | 3 | 59 | Homme Dernière année d'études primaires | 3994.1024587 | Exerce une profession | |
| 4 | 4 | 34 | Homme Enseignement supérieur y compris technique supérieur | 5731.6615081 | Exerce une profession | |
| 5 | 5 | 71 | Femme Dernière année d'études primaires | 4329.0940022 | Retraite | |
| 6 | 6 | 35 | Femme Enseignement technique ou professionnel court | 8674.6993828 | Exerce une profession | |
| 7 | 7 | 60 | Femme Dernière année d'études primaires | 6165.8034861 | Au foyer | |
| 8 | 8 | 47 | Homme Enseignement technique ou professionnel court | 12891.640759 | Exerce une profession | |
| 9 | 9 | 20 | Femme | 7808.8720636 | Etudiant, élève | |
| 10 | 10 | 28 | Homme Enseignement technique ou professionnel long | 2277.160471 | Exerce une profession | |

Showing 1 to 10 of 2,000 entries Previous **1** 2 3 4 5 ... 200 Next

Figure 13.13: Rendu d'une table par DT::`datatable`

Dans tous les cas il est déconseillé d'afficher de cette manière un tableau de données de très grandes dimensions, car le fichier HTML résultant contiendrait l'ensemble des données et serait donc très volumineux.



On peut définir un mode d'affichage par défaut pour tous les tableaux de données en modifiant les *Output options* du format HTML (onglet *General*, *Print dataframes as*), ou en modifiant manuellement l'option `df_print` de l'entrée `html_document` dans le préambule.

À noter que les tableaux issus de la fonction `freq` de `questionr` s'affichent comme des tableaux de données (et non comme des tableaux croisés).

13.6 Modèles de documents

On a vu ici la production de documents “classiques”, mais R Markdown permet de créer bien d'autres choses.

Le site de documentation de l'extension propose [une galerie](#) des différentes sorties possibles. On peut ainsi créer des slides, des sites Web ou même des livres entiers, comme le présent document.

13.6.1 Slides

Un usage intéressant est la création de diaporamas pour des présentations sous forme de slides. Le principe reste toujours le même : on mélange texte au format Markdown et code R, et R Markdown transforme le tout en présentations au format HTML ou PDF. En général les différents slides sont séparés au niveau de certains niveaux de titre.

Certains modèles de slides sont inclus avec R Markdown, notamment :

- `ioslides` et `Slidy` pour des présentations HTML
- `beamer` pour des présentations en PDF via `LaTeX`

Quand vous créez un nouveau document dans RStudio, ces modèles sont accessibles via l'entrée *Presentation* :

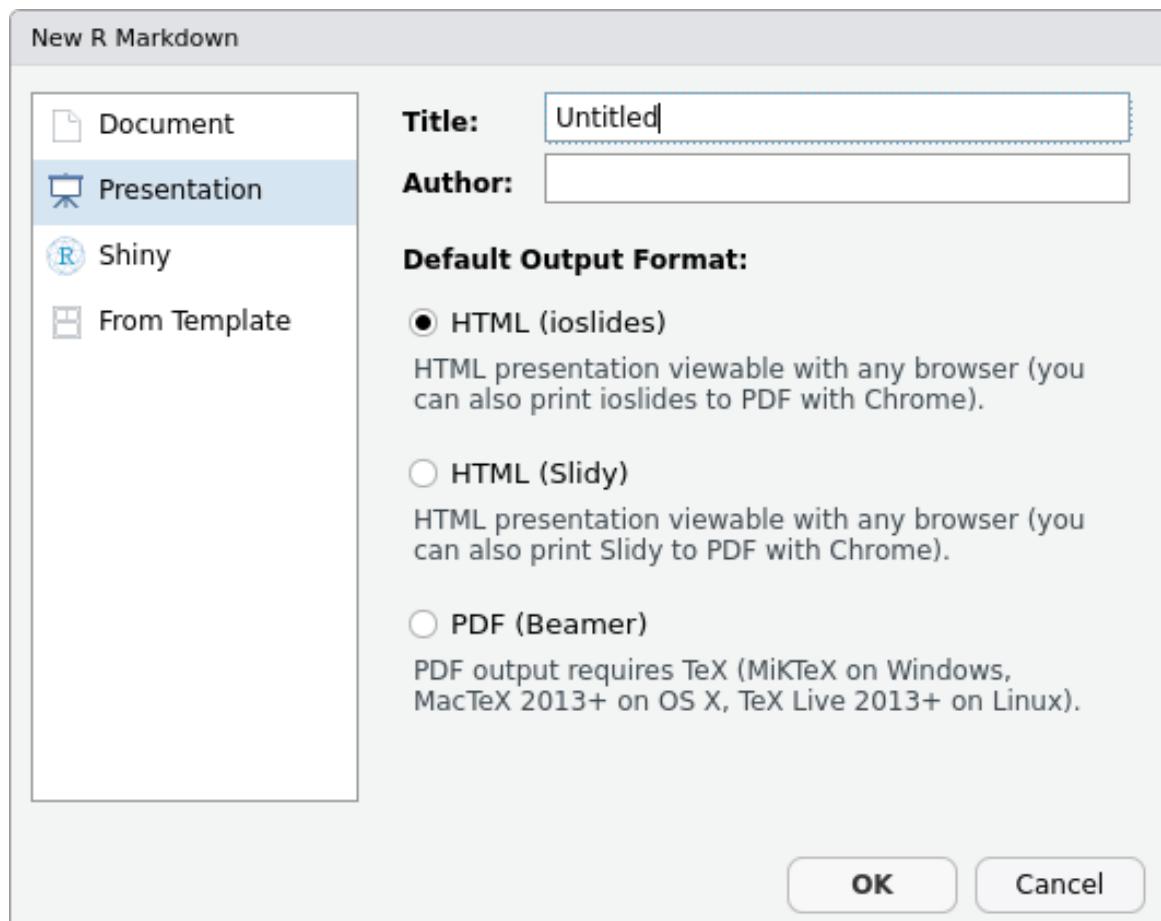


Figure 13.14: Créer une présentation R Markdown

D'autres extensions, qui doivent être installées séparément, permettent aussi des diaporamas dans des formats variés. On citera notamment :

- [xaringan](#) pour des présentations HTML basées sur [remark.js](#)
- [revealjs](#) pour des présentations HTML basées sur [reveal.js](#)
- [rmdshower](#) pour des diaporamas HTML basés sur [shower](#)

Une fois l'extension installée, elle propose en général un *template* de départ lorsqu'on crée un nouveau document dans RStudio. Ceux-ci sont accessibles depuis l'entrée *From Template*.

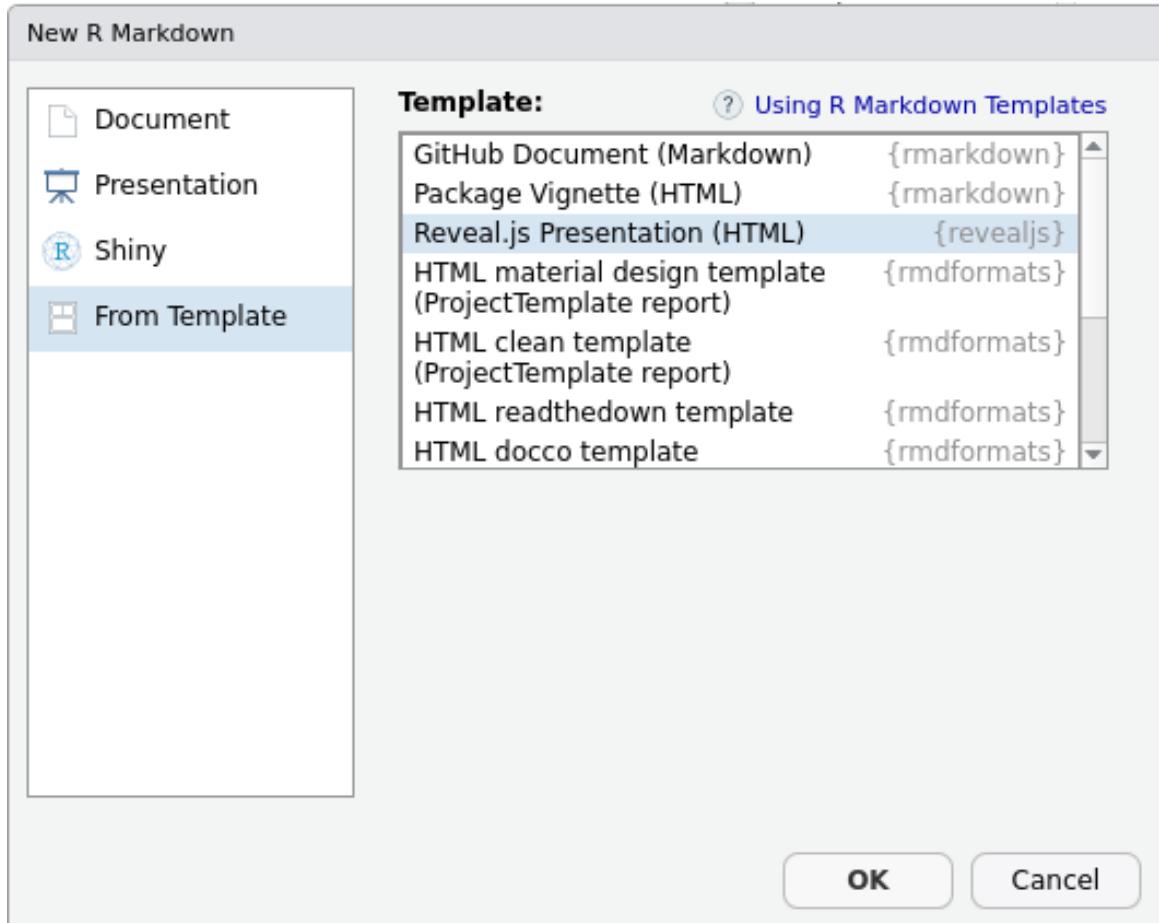


Figure 13.15: Créer une présentation à partir d'un template

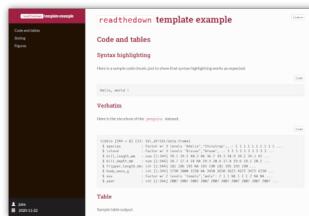
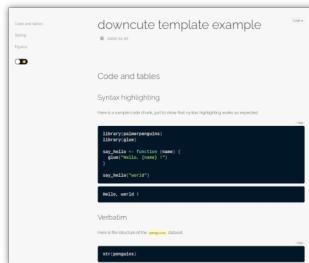
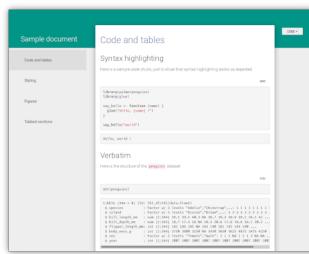
13.6.2 Templates

Il existe également différents *templates* permettant de changer le format et la présentation des documents générés. Une liste de ces formats et leur documentation associée est accessible depuis la page [formats](#) de la documentation.

On notera notamment :

- le format [Distill](#), adapté à des publications scientifiques ou techniques sur le Web
- le format [Tufte Handouts](#) qui permet de produire des documents PDF ou HTML dans un format proche de celui utilisé par Edward Tufte pour certaines de ses publications
- [rticles](#), package qui propose des templates LaTeX pour plusieurs revues scientifiques

Enfin, l'extension [rmdformats](#) propose plusieurs modèles HTML adaptés notamment pour des documents longs :

Figure 13.16: Modèle **readthedown**Figure 13.17: Modèle **downcute**Figure 13.18: Modèle **robobook**Figure 13.19: Modèle **material**

Là encore, la plupart du temps, ces modèles de documents proposent un *template* de départ lorsqu'on crée un nouveau document dans RStudio (entrée *From Template*) :

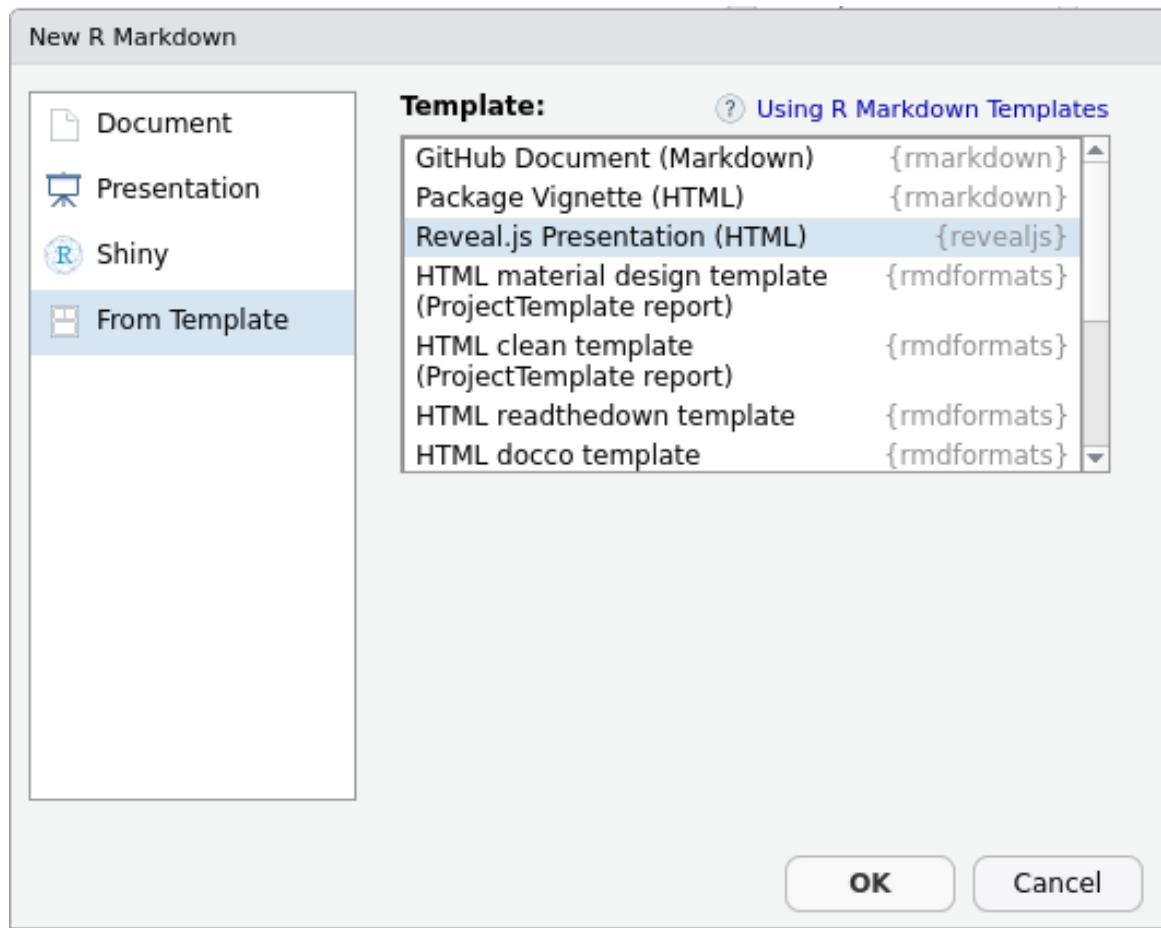


Figure 13.20: Crée un document à partir d'un template

13.7 Ressources

Les ressources suivantes sont toutes en anglais...

L'ouvrage *R for data science*, accessible en ligne, contient [un chapitre dédié à R Markdown](#).

Le [site officiel de l'extension](#) contient une documentation très complète, tant pour les débutants que pour un usage avancé.

Enfin, l'aide de RStudio (menu *Help* puis *Cheatsheets*) permet d'accéder à deux documents de synthèse : une “antisèche” synthétique (*R Markdown Cheat Sheet*) et un “guide de référence” plus complet (*R Markdown Reference Guide*).

Partie III

Aller plus loin

Chapitre 14

Écrire ses propres fonctions

14.1 Introduction et exemples

14.1.1 Structure d'une fonction

Nous avons vu lors de l'introduction à R que le langage repose sur deux grands concepts : les *objets* et les *fonctions*. Pour reprendre une citation de John Chambers, en R, tout ce qui existe est un objet, et tout ce qui se passe est une fonction.

Le principe d'une fonction est de prendre en entrée un ou plusieurs arguments (ou paramètres), d'effectuer un certain nombre d'actions et de renvoyer un résultat :



Nous avons déjà rencontré et utilisé un grand nombre de fonctions, certaines assez simples (`mean`, `max...`) et d'autres beaucoup plus complexes (`summary`, `mutate...`). R, comme tout langage de programmation, offre la possibilité de créer et d'utiliser ses propres fonctions.

Voici un exemple de fonction très simple, quoi que d'une utilité douteuse, puisqu'elle se contente d'ajouter 2 à un nombre :

```
ajoute2 <- function(x) {  
  res <- x + 2  
  return(res)  
}
```

En exécutant ce code, on crée une nouvelle fonction nommée `ajoute2`, que l'on peut directement utiliser dans un script ou dans la console :

```
ajoute2(3)  
#> [1] 5
```

On va décomposer pas à pas la structure de cette première fonction.

D'abord, une fonction est créée en utilisant l'instruction `function`. Celle-ci est suivie d'une paire de parenthèses et d'une paire d'accolades.

```
function() {  
}
```

Dans les parenthèses, on indique les *arguments* de la fonction, ceux qui devront lui être passés quand nous l'appellerons. Ici notre fonction ne prend qu'un seul argument, que nous avons décidé arbitrairement de nommer `x`.

```
function(x) {  
}
```

Les accolades comprennent une série d'instructions R qui constituent le *corps* de notre fonction. C'est ce code qui sera exécuté quand notre fonction est appelée. On peut utiliser dans le corps de la fonction les arguments qui lui sont passés. Ici, la première ligne utilise la valeur de l'argument `x`, lui ajoute 2 et stocke le résultat dans un nouvel objet `res`.

```
function(x) {  
  res <- x + 2  
}
```

Pour qu'elle soit utile, notre fonction doit renvoyer le résultat qu'elle a calculé précédemment. Ceci se fait via l'instruction `return` à qui on passe la valeur à retourner.

```
function(x) {  
  res <- x + 2  
  return(res)  
}
```

Enfin, pour que notre fonction puisse être appelée et utilisée, nous devons lui donner un nom, ou plus précisément la stocker dans un objet. Ici on la stocke dans un objet nommé `ajoute2`.

```
ajoute2 <- function(x) {  
  res <- x + 2  
  return(res)  
}
```



Les fonctions étant des objets comme les autres, elles suivent les mêmes contraintes pour leur nom : on a donc droit aux lettres, chiffres, point et tiret bas.

Attention à ne pas donner à votre fonction le nom d'une fonction déjà existante : par exemple, si vous créez une fonction nommée `table`, la fonction du même nom de R base ne sera plus disponible (sauf à la préfixer avec `base::table`). Si vous "écrasez" par erreur une fonction existante, il vous suffit de relancer votre session R et de trouver un nouveau nom.

Avec le code précédent, on a donc créé un nouvel objet `ajoute2` de type `function`. Cette nouvelle fonction prend un seul argument `x`, calcule la valeur `x + 2` et retourne ce résultat. On l'utilise en tapant son nom suivi de la valeur de son argument entre parenthèses, par exemple :

```
ajoute2(41)
#> [1] 43
```

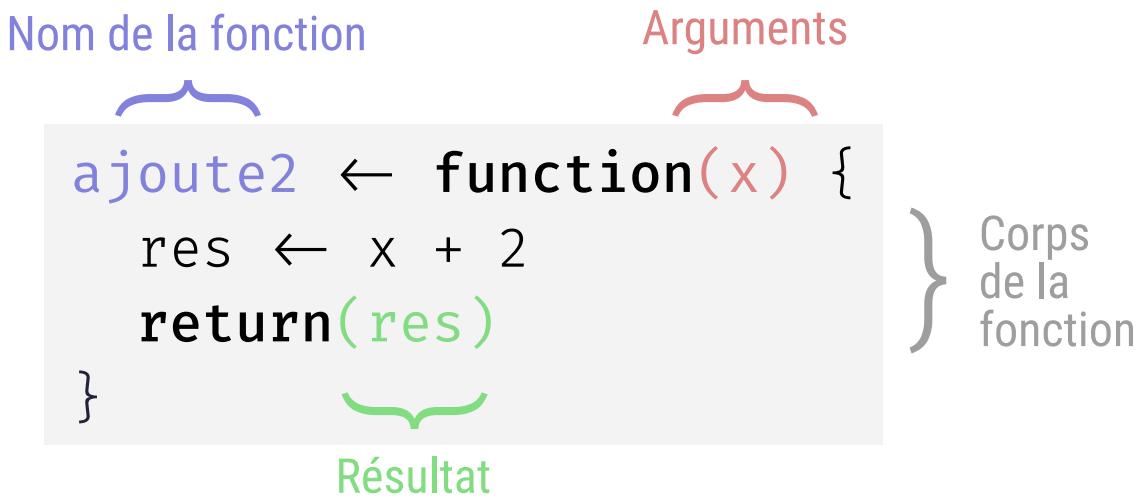
Ou encore :

```
y <- 5
z <- ajoute2(y)
z
#> [1] 7
```

À noter que comme $x + 2$ fonctionne si x est un vecteur, on peut aussi appeler notre fonction en lui passant un vecteur en argument.

```
vec <- 1:5
ajoute2(vec)
#> [1] 3 4 5 6 7
```

Si on récapitule, une fonction se définit donc de la manière suivante :



Une fonction peut évidemment prendre plusieurs arguments. Dans ce cas on liste les arguments dans les parenthèses en les séparant par des virgules :

```
somme <- function(x, y) {
  return(x + y)
}
```

```
somme(3, 5)
#> [1] 8
```

Une fonction peut aussi n'accepter aucun argument, dans ce cas on laisse les parenthèses vides.

```
miaule <- function() {
  return("Miaou")
}

miaule()
#> [1] "Miaou"
```

À noter que si on appelle une fonction avec un nombre d'arguments incorrect, cela génère une erreur.

```
somme(1)
#> Error in somme(1): l'argument "y" est manquant, avec aucune valeur par défaut
```

```
miaule("ouaf")
#> Error in miaule("ouaf"): argument inutilisé ("ouaf")
```

14.1.2 Exemple de fonction

Prenons un exemple un peu plus élaboré : la fonction `table()` retourne le tri à plat en effectifs d'une variable qualitative. On souhaite créer une fonction qui calcule plutôt le tri à plat en pourcentages. Voici une manière de le faire :

```
prop_tab <- function(v) {
  tri <- table(v)
  effectif_total <- length(v)
  tri <- tri / effectif_total * 100
  return(tri)
}
```

Notre fonction prend en entrée un argument nommé `v`, en l'occurrence un vecteur représentant une variable qualitative. On commence par faire le tri à plat de ce vecteur avec `table`, puis on calcule la répartition en pourcentages en divisant ce tri à plat par l'effectif total et en multipliant par 100.

Testons avec un vecteur d'exemple :

```
vec <- c("rouge", "vert", "vert", "bleu", "rouge")
prop_tab(vec)
#> v
#> bleu rouge vert
#> 20    40    40
```

Testons sur une variable du jeu de données `hdv2003`¹ :

```
library(questionr)
data(hdv2003)
prop_tab(hdv2003$qualif)
#> v
```

¹Le jeu de données `hdv2003` fait partie de l'extension `questionr`, il est décrit section A.3.2.2.

```
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>          10.15                  14.60                 4.30
#> Profession intermediaire           Cadre
#>          8.00                  13.00                 Employe
#>                               29.70
#>      Autre                      <NA>
#>          2.90
```

Ça fonctionne, mais avec une petite limite : par défaut `table()` ignore les NA. On peut modifier ce comportement en lui ajoutant un argument `useNA = "ifany"`.

```
prop_tab <- function(v) {
  tri <- table(v, useNA = "ifany")
  effectif_total <- length(v)
  tri <- tri / effectif_total * 100
  return(tri)
}

prop_tab(hdv2003$qualif)
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>          10.15                  14.60                 4.30
#> Profession intermediaire           Cadre
#>          8.00                  13.00                 Employe
#>                               29.70
#>      Autre                      <NA>
#>          2.90                  17.35
```

 Quand on modifie une fonction existante, il faut exécuter à nouveau le code correspondant à sa définition pour la “mettre à jour”. Ici, si on ne le fait pas l’objet `prop_tab` contiendra toujours l’ancienne définition.

Pour “mettre à jour” une fonction après avoir modifié son code, on peut soit sélectionner le code qui la définit et l’exécuter de la manière habituelle, soit, dans RStudio, se positionner dans le corps de la fonction et utiliser le raccourci clavier **Ctrl + Alt + F**.

Autre amélioration possible : on pourrait vouloir modifier le nombre de décimales affichées pour les pourcentages, par exemple en les limitant à 1. Pour cela on ajoute une instruction `round()`.

```
prop_tab <- function(v) {
  tri <- table(v, useNA = "ifany")
  effectif_total <- length(v)
  tri <- tri / effectif_total * 100
  tri <- round(tri, 1)
  return(tri)
}

prop_tab(hdv2003$qualif)
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>          10.2                  14.6                  4.3
#> Profession intermediaire           Cadre
#>          8.0                  13.0                  Employe
#>                               29.7
#>      Autre                      <NA>
#>          2.9                  17.3
```

Ça fonctionne ! Cela dit, limiter à un chiffre après la virgule ne convient pas forcément dans tous les cas. L’idéal serait d’offrir la possibilité à la personne qui appelle la fonction de choisir elle-même la précision de l’affichage.

Comment ? Tout simplement en ajoutant un deuxième argument à notre fonction, que nous nommerons `decimales`, et en utilisant cet argument à la place du 1 dans l'appel à `round()`.

```
prop_tab <- function(v, decimales) {
  tri <- table(v, useNA = "ifany")
  effectif_total <- length(v)
  tri <- tri / effectif_total * 100
  tri <- round(tri, decimales)
  return(tri)
}
```

Désormais, notre fonction s'utilise en lui indiquant deux arguments :

```
prop_tab(hdv2003$qualif, 1)
#> 
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>              10.2                14.6               4.3
#> Profession intermediaire
#>                     8.0                Cadre            Employe
#>                         Autre           13.0             29.7
#>                           2.9                <NA>
#>                               17.3
```

De la même manière, on pourrait vouloir laisser le choix à l'utilisateur d'afficher ou non les NA dans le tri à plat. C'est possible en ajoutant un troisième argument à notre fonction et en utilisant sa valeur dans le paramètre `useNA` de `table()`.

```
prop_tab <- function(v, decimales, useNA) {
  tri <- table(v, useNA = useNA)
  effectif_total <- length(v)
  tri <- tri / effectif_total * 100
  tri <- round(tri, decimales)
  return(tri)
}

prop_tab(hdv2003$qualif, 1, "no")
#> 
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>              10.2                14.6               4.3
#> Profession intermediaire
#>                     8.0                Cadre            Employe
#>                         Autre           13.0             29.7
#>                           2.9
```

14.1.3 Effets de bord et affichage de messages

Parfois une fonction n'a pas pour objectif de renvoyer un résultat mais d'accomplir une action, comme générer un graphique, afficher un message, enregistrer un fichier... Dans ce cas la fonction peut ne pas inclure d'instruction `return()`.

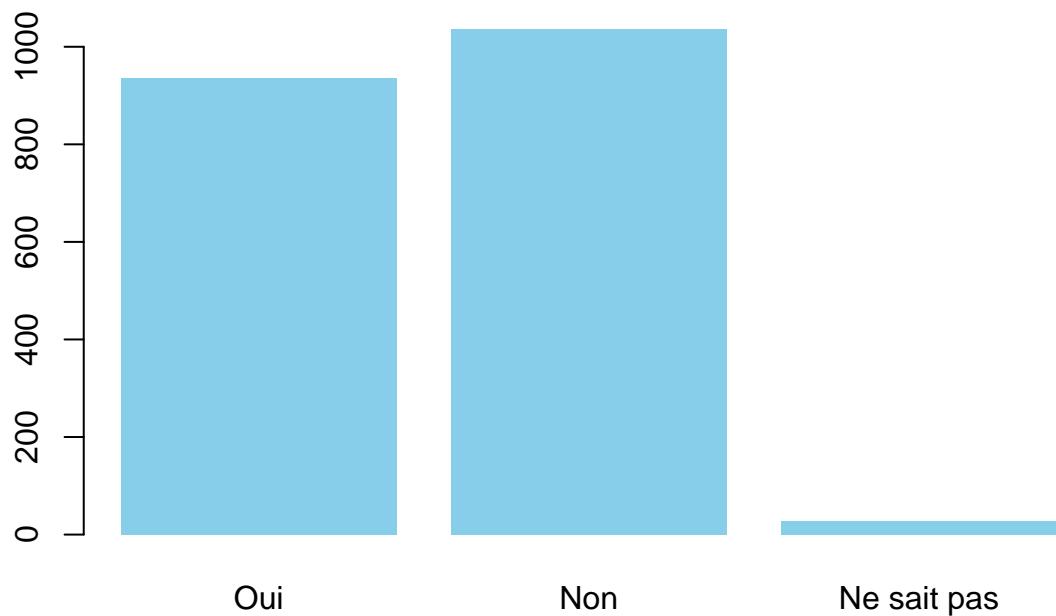


Les actions “visibles” dans notre session R accomplies par une fonction en-dehors du résultat renvoyé sont appelés des *effets de bord*.

Par exemple la fonction suivante prend en argument un vecteur et génère un diagramme en barres du tri à plat de cette variable (en modifiant un peu la présentation au passage).

```
my_barplot <- function(var) {
  tri <- table(var)
  barplot(tri, col = "skyblue", border = NA)
}

my_barplot(hdv2003$cuso)
```



Un autre effet de bord très courant consiste à afficher des informations dans la console. Pour cela on peut utiliser `print`, qui affiche de manière aussi lisible que possible l'objet qu'on lui passe en argument :

```
indicateurs <- function(v) {
  print(mean(v))
  print(sd(v))
}

indicateurs(hdv2003$age)
#> [1] 48.157
#> [1] 16.94181
```

Quand on souhaite seulement afficher une chaîne de caractère, on peut utiliser `cat()` qui fournit une sortie plus lisible que `print` :

```

hello <- function(nom) {
  cat("Bonjour,", nom, "!")
}

hello("Pierre-Edmond")
#> Bonjour, Pierre-Edmond !

```

Enfin, on peut aussi utiliser `message()` qui, comme son nom l'indique, affiche un message dans la console, avec une mise en forme spécifique. En général on l'utilise plutôt pour afficher des informations relatives au déroulement de la fonction.

Dans l'exemple suivant, on utilise la fonction `runif()` pour générer aléatoirement `n` nombres entre 0 et 1 et on affiche avec `cat()` la valeur du plus petit nombre généré. Comme l'exécution du `runif()` peut prendre du temps si `n` est grand, on affiche un message avec `message()` pour prévenir l'utilisateur.

```

min_alea <- function(n) {
  message("Génération de ", n, " nombres aléatoires...")
  v <- runif(n)
  cat("Le plus petit nombre généré vaut", min(v))
}

min_alea(50000)
#> Génération de 50000 nombres aléatoires...
#> Le plus petit nombre généré vaut 0.00001055235

```

14.1.4 Utilité des fonctions

On peut se demander dans quels cas il est utile de créer une fonction.

Une règle courante considère que dès qu'on a répété le même code plus de deux fois, il est préférable d'en faire une fonction. Celles-ci ont en effet comme avantage d'éviter la duplication du code.

Imaginons que nous avons récupéré un jeu de données avec toute une série de variables ayant les modalités "1" et "2" qui correspondent aux réponses "Oui" et "Non" à des questions. On crée un *data frame* fictif comportant quatre variables de ce type :

```

df <- data.frame(
  q1 = c("1", "1", "2", "1"),
  q2 = c("1", "2", "2", "2"),
  q3 = c("2", "2", "1", "1"),
  q4 = c("1", "2", "1", "1")
)

df
#>   q1 q2 q3 q4
#> 1  1  1  2  1
#> 2  1  2  2  2
#> 3  2  2  1  1
#> 4  1  2  1  1

```

On a vu section 9.3 qu'on peut recoder l'une de ces variables à l'aide de la fonction `fct_recode()` de l'extension `forcats` :

```
df$q1 <- fct_recode(df$q1,
  "Oui" = "1",
  "Non" = "2"
)
```

On peut donc être tenté de dupliquer ce code autant de fois qu'on a de questions à recoder :

```
df$q1 <- fct_recode(df$q1,
  "Oui" = "1",
  "Non" = "2"
)
df$q2 <- fct_recode(df$q2,
  "Oui" = "1",
  "Non" = "2"
)
df$q3 <- fct_recode(df$q3,
  "Oui" = "1",
  "Non" = "2"
)
df$q4 <- fct_recode(df$q4,
  "Oui" = "1",
  "Non" = "2"
)
```

Mais il est plus judicieux dans ce cas de créer une fonction pour ce recodage :

```
recode_oui_non <- function(var) {
  var_recodee <- fct_recode(var,
    "Oui" = "1",
    "Non" = "2"
  )
  return(var_recodee)
}
```

En effet, il est alors très simple d'appliquer ce recodage à plusieurs variables :

```
df$q1 <- recode_oui_non(df$q1)
df$q2 <- recode_oui_non(df$q2)
df$q3 <- recode_oui_non(df$q3)
df$q4 <- recode_oui_non(df$q4)
```

Autre avantage, si on réalise qu'on a commis une erreur et qu'en fait le code "1" correspondait à "Non" et le code "2" à "Oui", on n'a pas besoin de modifier tous les endroits où on a copié/collé notre recodage : on a juste à corriger la définition de la fonction.

Les avantages de procéder ainsi sont donc multiples :

- créer une fonction évite la répétition du code et le rend moins long et plus lisible, surtout si on donne à notre fonction un nom explicite permettant de comprendre facilement ce qu'elle fait.
- créer une fonction évite les erreurs de copier/coller du code.
- une fonction permet de mettre à jour plus facilement son code : si on se rend compte d'une erreur ou si on souhaite améliorer son fonctionnement, on n'a qu'un seul endroit à modifier.
- enfin, créer des fonctions permet potentiellement de rendre son code réutilisable d'un script à l'autre ou même d'un projet à l'autre. Voir, à terme, de les regrouper dans un *package* pour soi-même ou pour diffusion à d'autres utilisateurs et utilisatrices de R.

14.2 Arguments et résultat d'une fonction

14.2.1 Définition des arguments

Les arguments (ou paramètres) d'une fonction sont ce qu'on lui donne "en entrée", et qui vont soit lui fournir des données, soit modifier son comportement. La liste des arguments acceptés par une fonction est indiquée entre les parenthèses de l'appel de `function()` :

```
ma_fonction <- function(arg1, arg2, arg3) {
  print(arg1)
  print(arg2)
  print(arg3)
}
```

 Une fonction peut aussi ne pas accepter d'arguments, dans ce cas on la définit juste avec `function()`.

Lors de l'appel de la fonction, on peut lui passer les arguments *par position* :

```
ma_fonction(x, 12, TRUE)
```

Dans ce cas, `arg1` vaudra `x`, `arg2` vaudra `12` et `arg3` vaudra `TRUE`.

On peut aussi passer les arguments *par nom* :

```
ma_fonction(arg1 = x, arg2 = 12, arg3 = TRUE)
```

Quand on passe les arguments par nom, on peut les indiquer dans l'ordre que l'on souhaite :

```
ma_fonction(arg1 = x, arg3 = TRUE, arg2 = 12)
```

Et on peut évidemment mélanger passage par position et passage par nom :

```
ma_fonction(x, 12, arg3 = TRUE)
```

Le plus souvent, les premiers arguments acceptés par une fonction sont les données sur lesquelles elle va travailler, tandis que les arguments suivants sont des paramètres qui vont modifier son comportement. Par exemple, `median` accepte comme premier argument `x`, un vecteur, puis un argument `na.rm` qui va changer sa manière de calculer la médiane des valeurs de `x`.

 En général on appelle la fonction en passant les paramètres correspondant aux données par position, et les autres en les nommant. C'est ainsi qu'on ne fait ni `median(x = tailles, na.rm = TRUE)` ni `median(tailles, TRUE)`, mais plutôt `median(tailles, na.rm = TRUE)`.

En ce qui concerne le nom des arguments, en général ceux correspondant aux données transmises à une fonction peuvent avoir des noms relativement génériques (`x`, `y`, `v` pour un vecteur, `data` ou `df` pour un `data.frame`...). Les autres doivent par contre avoir des noms à la fois courts et explicites : par exemple plutôt `decimales` que `nd` ou `nombre_de_decimales`.

14.2.2 Valeurs par défaut

Au moment de la définition de la fonction, on peut indiquer une valeur par défaut qui sera prise par l'argument si la personne qui utilise la fonction n'en fournit pas.

Si on reprend la fonction `prop_tab` déjà définie plus haut :

```
prop_tab <- function(v, decimales, useNA) {
  tri <- table(v, useNA = useNA)
  tri <- tri / length(v) * 100
  tri <- round(tri, decimales)
  return(tri)
}
```

On peut indiquer une valeur par défaut aux arguments `decimales` et `useNA` de la manière suivante :

```
prop_tab <- function(v, decimales = 1, useNA = "ifany") {
  tri <- table(v, useNA = useNA)
  tri <- tri / length(v) * 100
  tri <- round(tri, decimales)
  return(tri)
}
```

Si on appelle `prop_tab` en lui passant uniquement le vecteur `v`, on voit que `decimales` vaut bien 1 et `useNA` vaut bien “`ifany`”:

```
prop_tab(hdv2003$qualif)
#> v
#>      Ouvrier specialise      Ouvrier qualifie      Technicien
#>          10.2                  14.6                 4.3
#> Profession intermediaire
#>          8.0                  Cadre                Employe
#>                           Autre           13.0                29.7
#>                           2.9             <NA>                17.3
```

14.2.3 Arguments obligatoires et arguments facultatifs

Si un argument n'a pas de valeur par défaut, il est *obligatoire* : si l'utilisateur essaye d'appeler la fonction sans définir cet argument, cela génère une erreur.

```
prop_tab <- function(v, decimales, useNA) {
  tri <- table(v, useNA = useNA)
  tri <- tri / length(v) * 100
  tri <- round(tri, decimales)
  return(tri)
}

prop_tab(hdv2003$sexe)
#> Error in match.arg(useNA): l'argument "useNA" est manquant, avec aucune valeur par défaut
```



Pour être tout à fait précis, l'erreur est générée uniquement lorsque l'argument sans valeur par défaut est utilisé dans la fonction.

Si à l'inverse un argument a une valeur par défaut, il devient *facultatif* : on peut appeler la fonction sans le définir.

```
prop_tab <- function(v, decimales = 1, useNA = "ifany") {
  tri <- table(v, useNA = useNA)
  tri <- tri / length(v) * 100
  tri <- round(tri, decimales)
  return(tri)
}

prop_tab(hdv2003$sex)
#> v
#> Homme Femme
#> 45 55
```

Parfois un argument est facultatif mais on n'a pas forcément de valeur par défaut à lui attribuer. Dans ce cas on lui attribue en général par défaut la valeur NULL, et on utilise l'instruction `if()` dans la fonction pour tester s'il a été défini ou pas. Ce cas de figure est détaillé section [17.2.4](#).

14.2.4 L'argument ...

Une fonction peut prendre un argument spécial nommé ... :

```
ma_fonction <- function(x, correct = TRUE, ...) {
```

```
}
```

Cet argument spécial “capture” tous les arguments présents et qui n'ont pas été définis avec la fonction. Par exemple, si on appelle la fonction précédente avec :

```
ma_fonction(1:5, correct = FALSE, title = "Titre", size = 12)
```

Alors ... contiendra les arguments `title` et `size` et leurs valeurs.



Si on veut accéder à la valeur de `size` dans ..., on utilise `list(...)$size`.

En général ... est utilisé pour passer ces arguments à d'autres fonctions. Reprenons notre fonction `my_barplot` définie précédemment :

```
my_barplot <- function(var) {
  tri <- table(var)
  barplot(tri, col = "skyblue", border = NA)
}
```

On pourrait permettre de personnaliser les couleurs des barres et de leurs bordures en ajoutant des arguments supplémentaires :

```
my_barplot <- function(var, col = "skyblue", border = NA) {
  tri <- table(var)
  barplot(tri, col = col, border = border)
}
```

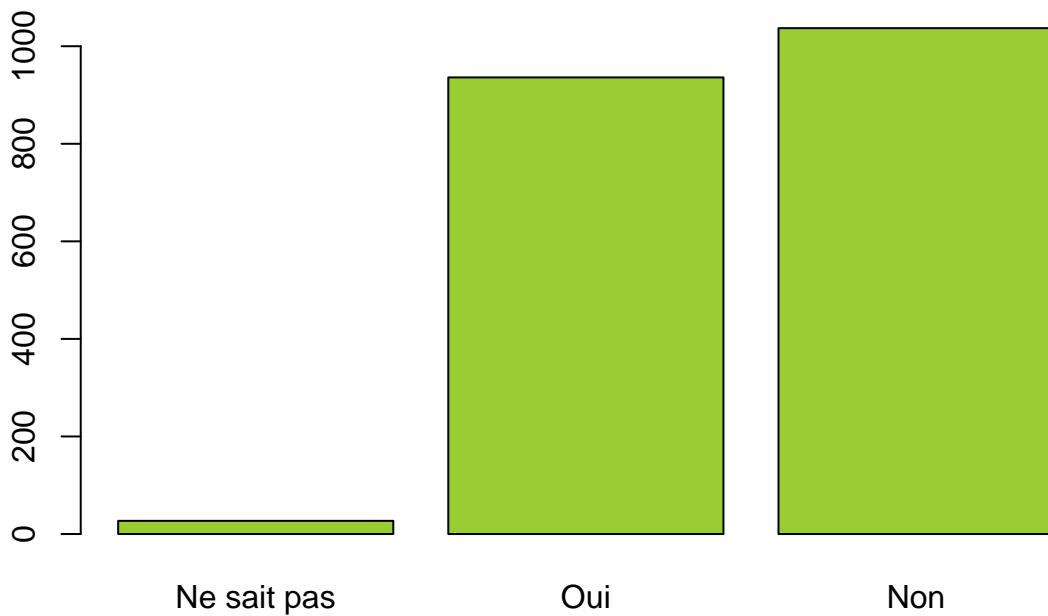
Mais si on veut aussi permettre de personnaliser d'autres arguments de `barplot` comme `main`, `xlab`, `xlim`... il faudrait rajouter autant d'arguments supplémentaires à notre fonction, ce qui deviendrait vite ingérable. Une solution est de "capturer" tous les arguments supplémentaires avec `...` et de les passer directement à `barplot`, de cette manière :

```
my_barplot <- function(var, ...) {
  tri <- table(var)
  tri <- sort(tri)
  barplot(tri, ...)
}
```

Ce qui permet d'appeler notre fonction avec tous les arguments possibles de `barplot`, par exemple :

```
my_barplot(
  hdv2003$clso,
  col = "yellowgreen",
  main = "Croyez-vous en l'existence des classes sociales ?"
)
```

Croyez-vous en l'existence des classes sociales ?



14.2.5 Résultat d'une fonction

On l'a vu, l'objectif d'une fonction est en général de renvoyer un résultat. Lors de la définition d'une fonction, le résultat peut être retourné en utilisant la fonction `return()` :

```
ajoute2 <- function(x) {
  res <- x + 2
  return(res)
}
```

En réalité, l'utilisation de `return()` n'est pas obligatoire : une fonction retourne automatiquement le résultat de la dernière instruction qu'elle exécute. On aurait donc pu écrire :

```
ajoute2 <- function(x) {
  res <- x + 2
  res
}
```

Ou même, encore mieux et plus lisible :

```
ajoute2 <- function(x) {
  x + 2
}
```



Dans la suite de ce document on utilisera, lorsque c'est possible, la syntaxe la plus “compacte” qui omet le `return()`.

Un point important à noter : lorsque R rencontre une instruction `return()` dans une fonction, il interrompt immédiatement son exécution et “sort” de la fonction en renvoyant le résultat.

Ainsi, dans la fonction suivante :

```
ajoute2 <- function(x) {
  return(x + 2)
  x * 5
}
```

L'instruction `x * 5` ne sera jamais exécutée car R “sort” de la fonction dès qu'il évalue le `return()` de la ligne précédente.

Conséquence de ce comportement, on ne peut pas utiliser plusieurs `return()` pour renvoyer plusieurs résultats depuis une seule fonction. Est-ce à dire qu'une fonction R ne pourrait renvoyer qu'une seule valeur ? Non, car si elle ne peut retourner qu'un seul objet, celui-ci peut être complexe et comporter plusieurs valeurs.

Par exemple, on a vu précédemment une fonction rudimentaire nommée `indicateurs()` qui affiche la moyenne et l'écart-type d'un vecteur numérique.

```
indicateurs <- function(v) {
  print(mean(v))
  print(sd(v))
}
```

Plutôt que de se contenter de les afficher dans la console, on pourrait vouloir retourner ces deux valeurs pour pouvoir les réutiliser par la suite. Pour cela, une première solution pourrait être de renvoyer un vecteur comportant ces deux valeurs.

```
indicateurs <- function(v) {
  moyenne <- mean(v)
  ecart_type <- sd(v)
  c(moyenne, ecart_type)
}
```

```
indicateurs(hdv2003$age)
#> [1] 48.15700 16.94181
```

Mais dans ce cas de figure il est recommandé de retourner plutôt une liste nommée², de cette manière :

```
indicateurs <- function(v) {
  moyenne <- mean(v)
  ecart_type <- sd(v)
  list(moyenne = moyenne, ecart_type = ecart_type)
}
```

```
indicateurs(hdv2003$age)
#> $moyenne
#> [1] 48.157
#>
#> $ecart_type
#> [1] 16.94181
```

On a du coup un affichage un peu plus lisible, et on peut accéder aux éléments du résultat via leur nom :

```
res <- indicateurs(hdv2003$age)
res$moyenne
#> [1] 48.157
```

14.3 Portée des variables

Un point délicat mais important quand on commence à créer ses propres fonctions concerne la *portée des variables*, c'est-à-dire la façon dont les objets créés dans une fonction et ceux existant en-dehors “cohabitent”. C'est une question assez complexe, mais seules quatre grandes règles sont réellement utiles au départ.

14.3.1 Une fonction peut accéder à un objet extérieur

Si on fait appel dans une fonction à un objet qui n'existe pas et n'a pas été passé comme argument, on obtient une erreur.

²Les listes seront abordées un peu plus en détail dans la partie 16.2.

```
f <- function() {
  obj
}

f()
#> Error in f(): objet 'obj' introuvable
```

Si on crée cet objet dans notre fonction avant de l'utiliser, on supprime évidemment l'erreur.

```
f <- function() {
  obj <- 2
  obj
}

f()
#> [1] 2
```

Mais on peut aussi accéder depuis une fonction à un objet qui existe dans notre environnement au moment où la fonction a été appelée.

```
f <- function() {
  obj
}

obj <- 3
f()
#> [1] 3
```

Dans cet exemple, au moment de l'exécution de `f()`, comme `obj` n'existe pas au sein de la fonction (il n'a pas été passé comme argument ni défini dans le corps de la fonction), R va chercher dans l'environnement global, celui depuis lequel la fonction a été appelée. Comme il trouve un objet `obj`, il utilise sa valeur au moment de l'appel de la fonction.

14.3.2 Les arguments et les objets créés dans la fonction sont prioritaires

Que se passe-t-il si un objet avec le même nom existe à la fois dans la fonction et dans notre environnement global ? Dans ce cas *R privilégie l'objet créé dans la fonction*.

```
f <- function() {
  obj <- 10
  obj
}

obj <- 3
f()
#> [1] 10
```

Cette règle s'applique également pour les arguments passés à la fonction.

```
f <- function(obj) {
  obj
}

obj <- 3
f(20)
#> [1] 20
```

14.3.3 Un objet créé dans une fonction n'existe que dans cette fonction

Autre règle importante : un objet créé à l'intérieur d'une fonction n'est pas accessible à l'extérieur de celle-ci.

```
f <- function() {
  nouvel_objet <- 15
  nouvel_objet
}

f()
#> [1] 15
nouvel_objet
#> Error in eval(expr, envir, enclos): objet 'nouvel_objet' introuvable
```

Ici, `nouvel_objet` existe tant qu'on est dans la fonction, mais il est détruit dès qu'on en sort et donc inaccessible dans notre environnement global.



Les objets créés dans notre session et qui existent dans notre environnement (tel que visible dans l'onglet *Environment* de RStudio) sont appelés des **objets globaux** : ils existent et sont accessibles pour les fonctions appelées depuis cet environnement. Les objets créés lors de l'exécution d'une fonction sont à l'inverse des **objets locaux** : ils n'existent qu'à l'intérieur de la fonction et pour la durée de son exécution. Si deux objets du même nom coexistent, l'objet local est prioritaire par rapport à l'objet global.

14.3.4 On ne peut pas modifier un objet global dans une fonction

Une conséquence importante de la troisième règle est qu'il n'est pas possible de modifier un objet de notre environnement global depuis une fonction³ :

```
f <- function() {
  obj <- 10
  message("Valeur dans la fonction : ", obj)
}

obj <- 3
f()
#> Valeur dans la fonction : 10
obj
#> [1] 3
```

Pour comprendre le résultat obtenu, on peut essayer de décomposer pas à pas :

1. Au moment du `obj <- 3`, R crée un objet global nommé `obj` avec la valeur 3.

³En réalité c'est possible avec l'opérateur `<<-`, mais c'est fortement déconseillé dans la très grande majorité des cas.

2. Quand on exécute `f()` et qu'on rencontre l'instruction `obj <- 10`, R crée un nouvel objet nommé `obj`, local celui-ci, avec la valeur 10. À ce moment-là on a donc deux objets distincts portant le même nom, l'un global avec la valeur 3, l'autre local avec la valeur 10. Comme l'objet local est prioritaire, c'est lui qui est utilisé lors de l'affichage du message.
3. Lorsqu'on sort de `f()`, l'objet local contenant la valeur 10 est détruit. Il ne reste plus que l'objet global avec la valeur 3. C'est donc lui qui est affiché lors du dernier appel à `obj`.

Pour les mêmes raisons, dans l'exemple suivant, le recodage appliqué à la variable `taille` du tableau `df` passé en argument à la fonction `recode_taille()` n'est pas conservé en-dehors de la fonction. Ce recodage n'existe que dans un tableau `d` local à la fonction, et détruit dès qu'on en est sorti.

```
df <- data.frame(taille = c(155, 182), poids = c(65, 71))

recode_taille <- function(d) {
  d$taille <- d$taille / 100
}

recode_taille(df)

# Le recodage n'est pas conservé
df
#>   taille poids
#> 1    155    65
#> 2    182    71
```

Si on souhaite modifier un objet global, on doit le passer comme argument en entrée de notre fonction, et le renvoyer comme résultat en sortie. Pour que le recodage précédent soit bien répercuté dans notre tableau `df`, on doit faire :

```
recode_taille <- function(d) {
  d$taille <- d$taille / 100
  d
}

df <- recode_taille(df)

# Le recodage est bien conservé
df
#>   taille poids
#> 1    1.55    65
#> 2    1.82    71
```

14.4 Les fonctions comme objets

Quand on crée une fonction, on la “nomme” en la stockant dans un objet. Cet objet peut être utilisé comme n'importe quel autre objet dans R. On peut ainsi copier une fonction en l'attribuant à un nouvel objet :

```
f <- function(x) {
  x + 2
}

g <- f
g(10)
#> [1] 12
```

On a déjà vu à de nombreuses reprises que quand on fournit juste un nom d'objet à R, celui-ci affiche son contenu dans la console. C'est aussi le cas pour les fonctions : dans ce cas c'est le code source de la fonction qui est affiché.

```
f
#> function(x) {
#>   x + 2
#> }
```

14.4.1 Passer des fonctions comme argument

Certaines fonctions sont prévues pour s'appliquer elles-mêmes à des fonctions. Par exemple, `formals` et `body` permettent d'afficher respectivement les arguments et le corps d'une fonction passée en argument.

```
formals(f)
#> $x
```

```
body(f)
#> {
#>   x + 2
#> }
```

Il est donc possible de passer une fonction comme argument d'une autre fonction, comme dans `body(f)`. On a déjà vu un exemple de ce type de fonctionnement avec la fonction `tapply` dans la section 4.2.2. Celle-ci prend trois arguments : un vecteur de valeurs, un facteur, et une fonction. Elle applique ensuite la fonction aux valeurs pour chaque niveau du facteur.

Par exemple, si on a un data frame avec une liste de fruits et leur poids :

```
df <- data.frame(
  fruit = c("Pomme", "Pomme", "Citron", "Citron"),
  poids = c(147, 189, 76, 91)
)

df
#>   fruit  poids
#> 1 Pomme   147
#> 2 Pomme   189
#> 3 Citron    76
#> 4 Citron    91
```

On peut utiliser `tapply` pour calculer le poids moyen par type de fruit.

```
tapply(df$poids, df$fruit, mean)
#> Citron  Pomme
#> 83.5 168.0
```

Si on souhaite plutôt calculer le poids maximal, il suffit de passer à `tapply` la fonction `max` plutôt que la fonction `mean`.

```
tapply(df$poids, df$fruit, max)
#> Citron Pomme
#> 91 189
```

Cette manière de transmettre une fonction à une autre fonction peut être un peu déroutante de prime abord, mais c'est une mécanique qu'on va retrouver très souvent dans les chapitres suivants.



Si `f` est une fonction, il est important de bien faire la différence entre `f` et `f()` :

- `f` est la fonction en elle-même
- `f()` est le résultat de la fonction quand on l'exécute sans lui passer d'argument

Quand on passe une fonction comme argument à une autre fonction, on utilise donc toujours la notation sans les parenthèses.

14.4.2 Fonctions anonymes

Dans le cas où on souhaite calculer quelque chose pour lequel une fonction n'existe pas déjà, on peut créer une nouvelle fonction :

```
poids_moyen_kg <- function(poids) {
  mean(poids / 1000)
}
```

Et la passer en argument à `tapply()` :

```
tapply(df$poids, df$fruit, poids_moyen_kg)
#> Citron Pomme
#> 0.0835 0.1680
```

Si on ne souhaite pas réutiliser cette fonction par la suite, on peut aussi définir cette fonction directement comme argument de `tapply` :

```
tapply(df$poids, df$fruit, function(poids) {
  mean(poids/1000)
})
#> Citron Pomme
#> 0.0835 0.1680
```

Dans ce cas on a créé ce qu'on appelle une *fonction anonyme*, qui n'a pas de nom (elle n'a pas été stockée dans un objet), et qui n'existe que le temps de l'appel à `tapply`.

14.5 Ressources

L'ouvrage *R for Data Science* (en anglais), accessible en ligne, contient un chapitre complet d'[introduction sur les fonctions](#).

L'ouvrage *Advanced R* (également en anglais) aborde de manière très approfondie [les fonctions](#) ainsi que la [programmation fonctionnelle](#).

Le manuel officiel *Introduction to R* (toujours en anglais) contient une partie sur [l'écriture de ses propres fonctions](#).

14.6 Exercices

14.6.1 Introduction et exemples

Exercice 1.1

Écrire une fonction nommée `perimetre` qui prend en entrée un argument nommé `r` et retourne le périmètre d'un cercle de rayon `r`, c'est-à-dire $2 * \pi * r$ (`pi` est un objet R qui contient la valeur de π).

Vérifier avec l'appel suivant :

```
perimetre(4)
#> [1] 25.13274
```

Exercice 1.2

Écrire une fonction `etendue` qui prend en entrée un vecteur numérique et retourne la différence entre la valeur maximale et la valeur minimale de ce vecteur.

Vérifier avec l'appel suivant :

```
etendue(c(18, 35, 21, 40))
#> [1] 22
```

Exercice 1.3

Écrire une fonction nommée `alea` qui accepte un argument `n`, génère un vecteur de `n` valeurs aléatoires entre 0 et 1 avec la fonction `runif(n)` et retourne ce vecteur comme résultat.

Modifier la fonction pour qu'elle accepte deux arguments supplémentaires `min` et `max` et qu'elle retourne un vecteur de `n` valeurs aléatoires comprises entre `min` et `max` avec la fonction `runif(n, min, max)`.

Modifier à nouveau la fonction pour qu'elle retourne un vecteur de `n` nombres *entiers* aléatoires compris entre `min` et `max` en appliquant la fonction `trunc()` au vecteur généré par `runif()`.

Vérifier le résultat avec :

```
v <- alea(10000, 1, 6)
table(v)
#> v
#>   1    2    3    4    5    6
#> 1642 1611 1706 1687 1696 1658
```

Exercice 1.4

Écrire une fonction nommée `meteo` qui prend un argument nommé `ville` avec le corps suivant :

```
out <- readLines(paste0("https://v2.wttr.in/", ville, "?A"))
cat(out, sep = "\n")
```

Tester la fonction avec par exemple `meteo("Lyon")` (il est possible que l'affichage dans la console ne soit pas lisible si vous travaillez sous Windows).

Exercice 1.5

Soit le code suivant, qui recode une variable du jeu de données `hdv2003` en utilisant `str_to_lower()` puis `fct_recode()` :

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

hdv2003$hard.rock <- str_to_lower(hdv2003$hard.rock)
hdv2003$hard.rock <- fct_recode(hdv2003$hard.rock, "o" = "oui", "n" = "non")
```

Transformer ce code en une fonction nommée `recode_oui_non`, et appliquer cette fonction à `hard.rock`, `lecture.bd` et `cuisine`.

14.6.2 Arguments et résultat

Exercice 2.1

Observer le code de la fonction suivante pour comprendre à quoi correspondent chacun de ses trois arguments, puis réordonner et renommer ces arguments de manière plus pertinente :

```
moyenne_arrondie <- function(d, vecteur_contenant_les_donnees, supprimer_les_na) {
  res <- mean(vecteur_contenant_les_donnees, na.rm = supprimer_les_na)
  res <- round(res, d)
  return(res)
}
```

Donner aux arguments de la fonction une valeur par défaut.

Simplifier la fonction en utilisant la syntaxe plus compacte qui ne fait pas appel à `return()`.

Exercice 2.2

Simplifier la fonction suivante pour que son corps ne fasse plus qu'une seule ligne :

```
centrer_reduire <- function(x) {
  res <- x - mean(x)
  res <- res / sd(x)
  return(res)
}
```

Exercice 2.3

Le code suivant permet de déterminer la lettre initiale et la longueur d'un mot.

```
initiale <- str_sub(mot, 1, 1)
longueur <- nchar(mot)
```

Utiliser ce code pour créer une fonction `caracteristiques_mot()` qui prend un argument `mot` et retourne à la fois son initiale et sa longueur.

```
caracteristiques_mot("Bidonnage")
#> $initiale
#> [1] "B"
#>
#> $longueur
#> [1] 9
```

Facultatif : modifier la fonction pour qu'elle retourne un vecteur plutôt qu'une liste, et l'appliquer sur un mot de votre choix. Que constatez-vous ?

14.6.3 Portée des variables

Exercice 3.1

En lisant les codes suivants, essayer de prévoir quelle va être la valeur affichée par la dernière ligne. Vérifier en exécutant le code :

```
f <- function() {  
  x <- 3  
  x  
}  
  
f()
```

```
f <- function() {  
  x  
}  
  
x <- 5  
f()
```

```
f <- function(x) {  
  x  
}  
  
x <- 5  
f(30)
```

```
f <- function(x = 100) {  
  x  
}  
  
x <- 5  
f()
```

```
f <- function(x = 100) {  
  x <- 150  
  x  
}  
  
x <- 5  
f(30)
```

```
f <- function() {  
  x <- 5  
}  
  
x <- 1000  
f()  
x
```

Exercice 3.2

Dans le code suivant, on a essayé de créer une fonction qui modifie un tableau de données passé en argument pour ne conserver que les lignes correspondant aux pommes. Est-ce que ça fonctionne ?

```
df <- data.frame(
  fruit = c("Pomme", "Pomme", "Citron", "Citron"),
  poids = c(147, 189, 76, 91)
)

filtre_pommes <- function(d) {
  d <- dplyr::filter(d, fruit == "Pomme")
}

filtre_pommes(df)
df
```

Modifier le code pour obtenir le résultat souhaité.

14.6.4 Les fonctions comme objets

Exercice 4.1

Écrire une fonction nommée `bonjour` qui ne prend aucun argument et affiche juste le texte “Bonjour !” dans la console.

Exécuter dans la console les deux commandes suivantes tour à tour :

- `bonjour()`
 - `bonjour`

Comprenez-vous la différence entre les deux ?

Copier la fonction dans un nouvel objet nommé `salut`. Exécuter la nouvelle fonction ainsi créée.

Exercice 4.2

Construire une fonction `etendue()` qui prend en entrée un vecteur numérique et retourne la différence entre la valeur maximale et la valeur minimale de ce vecteur (vous pouvez récupérer le code de l'exercice 1.2).

À l'aide de `tapply()`, appliquez la fonction `etendue()` à la variable `age` pour chaque valeur de `qualif` dans le jeu de données `hdv2003`.

| | | |
|-----------------------------|------------------|------------|
| #> Ouvrier specialise | Ouvrier qualifie | Technicien |
| #> 74 | 68 | 62 |
| #> Profession intermediaire | Cadre | Employe |
| #> 62 | 63 | 72 |
| #> Autre | | |
| #> 78 | | |

Réécrire le code précédent en utilisant une fonction anonyme (*ie* en définissant la fonction directement dans le `tapply`).

Exercice 4.3

Exécutez le code suivant. Comprenez-vous les résultats obtenus ?

```
f <- function(y) {  
  y * 4  
}  
  
body(f)  
f(5)  
  
body(f) <- quote(y + 2)  
body(f)  
f(5)
```

Intuitivement, comprenez-vous à quoi sert la fonction `quote` ?

Chapitre 15

dplyr avancé

L'extension `dplyr` a déjà été présentée dans la partie 10. On va voir ici comment aller un peu plus loin dans l'utilisation du *package*, notamment en utilisant nos propres fonctions et en appliquant des transformations à des ensembles de colonnes.

On commence par charger les extensions du *tidyverse* ainsi que les jeux de données `hdv2003` et `rp2018` de l'extension `questionr`.

```
library(tidyverse)
library(questionr)
data(hdv2003)
data(rp2018)
```

15.1 Appliquer ses propres fonctions

15.1.1 Exemple avec `mutate`

Soit le jeu de données fictif suivant, dont chaque ligne représente un individu pour lequel on dispose de sa PCS, celle de ses parents, son âge et celui de ses enfants.

```
df <- tribble(
  ~id, ~pcs, ~pcs_pere, ~pcs_mere, ~age, ~`age enf1`, ~`age enf2`, ~`age enf3`,
  1, "5", "5", "6", 25, 2, NA, NA,
  2, "3", "3", "2", 45, 12, 8, 2,
  3, "4", "2", "5", 29, 7, NA, NA,
  4, "2", "1", "4", 32, 6, 3, NA,
  5, "1", "4", "3", 65, 39, 36, 28,
  6, "6", "6", "6", 51, 18, 12, NA,
  7, "5", "4", "6", 37, 8, 4, 1,
  8, "3", "3", "1", 42, 16, 10, 5
)

df
#> # A tibble: 8 x 8
#>   id    pcs  pcs_pere  pcs_mere  age `age enf1` `age enf2` `age enf3`
#>   <dbl> <chr> <chr>     <chr>     <dbl>       <dbl>       <dbl>       <dbl>
#> 1     1  5      5        6        25         2        NA        NA
#> 2     2  3      3        2        45        12         8        2
#> 3     3  4      2        5        29         7        NA        NA
#> 4     4  2      1        4        32         6         3        NA
```

```
#> 5    5 1    4    3    65    39    36    28
#> 6    6 6    6    6    51    18    12    NA
#> 7    7 5    4    6    37    8     4     1
#> 8    8 3    3    1    42    16    10    5
```

Dans ce tableau les PCS sont indiquées sous forme de codes : il serait plus lisible de les avoir sous forme d'intitulés de catégorie socio-professionnelle. On a vu section 9.3.2 qu'on peut effectuer ce recodage avec la fonction `fct_recode()` de l'extension `forcats`.

```
df %>%
  mutate(
    pcs = fct_recode(pcs,
      "Agriculteur" = "1",
      "Indépendant" = "2",
      "Cadre" = "3",
      "Intermédiaire" = "4",
      "Employé" = "5",
      "Ouvrier" = "6"
    )
  )
```

Plutôt que d'intégrer le code du recodage directement dans le `mutate()`, on peut l'extraire en créant une fonction.

```
recode_pcs <- function(v) {
  fct_recode(v,
    "Agriculteur" = "1",
    "Indépendant" = "2",
    "Cadre" = "3",
    "Intermédiaire" = "4",
    "Employé" = "5",
    "Ouvrier" = "6"
  )
}
```

On peut dès lors simplifier notre `mutate` en appelant notre nouvelle fonction.

```
df %>%
  mutate(pcs = recode_pcs(pcs))
```

Premier avantage : on gagne en lisibilité. On a déplacé le code d'une opération spécifique dans une fonction avec un nom “parlant”, ce qui permet de savoir facilement à quoi elle sert. Et on a simplifié notre `mutate` qui est désormais plus lisible parce qu'il fait apparaître la logique de nos opérations (on veut recoder les PCS) sans en inclure les détails.

Le deuxième avantage évident, comme pour toute fonction, est qu'on peut la réutiliser pour appliquer ce recodage à plusieurs variables. Ainsi, si on veut recoder de la même manière `pcs` et `pcs_mere`, il suffit de faire :

```
df %>%
  mutate(
    pcs = recode_pcs(pcs),
    pcs_mere = recode_pcs(pcs_mere)
  )
```

Le code est plus court, plus lisible, on évite les erreurs de copier/coller, et si on souhaite modifier le recodage on n'a à intervenir qu'à un seul endroit en modifiant notre fonction.

15.1.2 Exemple avec `summarise`

Autre exemple, cette fois sur le jeu de données `rp2018`. Imaginons qu'on souhaite calculer, pour chaque région, le pourcentage de communes dont le nom se termine par une série de caractères donnée : par exemple, le pourcentage de communes dont le nom se termine par "ac".

Comme il ne s'agit pas forcément d'une question triviale, on va décomposer le problème et rappeler (comme vu section 11.6) que la fonction `str_detect()` de l'extension `stringr` permet de détecter quels éléments d'un vecteur de chaînes de caractères correspondent à une expression régulière. Ainsi, si on veut détecter si un nom de commune (variable `rp2018$commune`) se termine par "ac", on utilisera :

```
str_detect(rp2018$commune, "ac$")
```



Le symbole `\$` dans l'expression régulière `"ac$"` représente la fin de la chaîne de caractères. Il permet de s'assurer qu'on ne détecte que les noms de communes se terminant par "ac" (comme "Figeac"), et pas ceux contenant "ac" à un autre endroit (comme "Arcachon").

Si on veut compter le nombre de communes pour lesquelles on a détecté une terminaison en "ac", on peut utiliser un idiom courant en R et appliquer la fonction `sum()` au résultat précédent : les `TRUE` du résultat du `str_detect` sont alors convertis en 1, les `FALSE` en 0, et le `sum()` renverra donc le nombre de `TRUE`.

```
sum(str_detect(rp2018$commune, "ac$"))
#> [1] 131
```

Si on souhaite convertir ce résultat en pourcentage, il faut qu'on divise par le nombre total de communes, et qu'on multiplie par 100.

```
sum(str_detect(rp2018$commune, "ac$")) / length(rp2018$commune) * 100
#> [1] 2.418313
```

On crée une fonction nommée `prop_suffixe` qui a pour objectif d'effectuer ce calcul. Elle prend en entrée deux arguments : un vecteur de chaînes de caractères et un suffixe à détecter, et retourne le pourcentage d'éléments du vecteur se terminant par le suffixe. On rajoute nous-même le `"$"` à la fin du suffixe en question pour faciliter l'usage de la fonction.

Le résultat final est le suivant :

```
prop_suffixe <- function(v, suffixe) {
  # On ajoute $ à la fin du suffixe pour capturer uniquement en fin de chaîne
  suffixe <- paste0(suffixe, "$")
  # Détection du suffixe
  nb_detect <- sum(str_detect(v, suffixe))
  # On retourne le pourcentage
  nb_detect / length(v) * 100
}
```

On peut utiliser notre fonction de la manière suivante :

```
prop_suffixe(rp2018$commune, "ac")
#> [1] 2.418313
```

On a donc dans notre jeu de données 2.42% de communes dont le nom se termine par “ac”¹.

Si maintenant on souhaite calculer ce pourcentage pour toutes les régions françaises, il suffit d’appeler notre fonction dans un `summarise` :

```
rp2018 %>%
  group_by(region) %>%
  summarise(prop_ac = prop_suffixe(commune, "ac")) %>%
  arrange(desc(prop_ac))
#> # A tibble: 17 x 2
#>   region           prop_ac
#>   <chr>            <dbl>
#> 1 Nouvelle-Aquitaine 10.8 
#> 2 Occitanie          4.39 
#> 3 Bretagne           4.26 
#> 4 Auvergne-Rhône-Alpes 2.25 
#> 5 Pays de la Loire  2.20 
#> 6 Bourgogne-Franche-Comté 0.995
#> 7 Provence-Alpes-Côte d'Azur 0.581
#> 8 Normandie          0.347
#> 9 Hauts-de-France    0.185
#> 10 Centre-Val de Loire 0
#> 11 Corse              0
#> 12 Grand Est          0
#> 13 Guadeloupe         0
#> 14 Guyane              0
#> 15 Île-de-France       0
#> 16 La Réunion          0
#> 17 Martinique          0
```

L’avantage d’avoir créé une fonction pour effectuer cette opération et qu’on peut du coup très facilement faire le même calcul en faisant varier le suffixe recherché.

```
rp2018 %>%
  group_by(region) %>%
  summarise(prop_ac = prop_suffixe(commune, "ieu")) %>%
  arrange(desc(prop_ac))
#> # A tibble: 17 x 2
#>   region           prop_ac
#>   <chr>            <dbl>
#> 1 Auvergne-Rhône-Alpes 2.79 
#> 2 Occitanie          0.763 
#> 3 Bourgogne-Franche-Comté 0.498
#> 4 Pays de la Loire  0.489 
#> 5 Normandie           0.347
#> 6 Nouvelle-Aquitaine 0.187 
#> 7 Bretagne             0
#> 8 Centre-Val de Loire 0
```

¹Attention, le jeu de données ne comporte que les communes de plus de 2000 habitants.

```
#> 9 Corse 0
#> 10 Grand Est 0
#> 11 Guadeloupe 0
#> 12 Guyane 0
#> 13 Hauts-de-France 0
#> 14 Île-de-France 0
#> 15 La Réunion 0
#> 16 Martinique 0
#> 17 Provence-Alpes-Côte d'Azur 0
```

En créant une fonction plutôt qu'en mettant notre code directement dans le `summarise` on a un script plus lisible, plus facile à maintenir, et des fonctionnalités facilement réutilisables.

15.1.3 Exemple avec `rename_with`

On a vu section 10.2.3 que `dplyr` propose la fonction `rename()` pour renommer des colonnes d'un tableau de données. On peut l'utiliser par exemple pour remplacer un espace par un `_` dans le nom d'une variable de `df`.

```
df %>% rename("age_enf1" = "age enf1")
```

Supposons maintenant qu'on souhaite appliquer la même transformation à l'ensemble des variables de `df`. Une solution pour cela est d'utiliser la fonction `rename_with()`, toujours fournie par `dplyr`, qui prend en argument non pas une correspondance `"nouveau nom" = "ancien nom"` mais une fonction qui sera appliquée à l'ensemble des noms de colonnes.

Par exemple, si on souhaite convertir tous les noms de colonnes en majuscules, on peut passer comme argument la fonction `str_to_upper()` de `stringr`.

```
df %>% rename_with(str_to_upper)
#> # A tibble: 8 x 8
#>   ID PCS  PCS_PERE PCS_MERE AGE `AGE ENF1` `AGE ENF2` `AGE ENF3`
#>   <dbl> <chr> <chr>    <chr>   <dbl>      <dbl>      <dbl>      <dbl>
#> 1     1 5      5       6      25        2        NA        NA
#> 2     2 3      3       2      45       12        8        2
#> 3     3 4      2       5      29        7        NA        NA
#> 4     4 2      1       4      32        6        3        NA
#> 5     5 1      4       3      65       39        36       28
#> 6     6 6      6       6      51       18        12        NA
#> 7     7 5      4       6      37        8        4        1
#> 8     8 3      3       1      42       16        10        5
```

Pour remplacer les espaces par des `_`, on va d'abord créer une fonction *ad hoc* qui utilise `str_replace_all`.

```
remplace_espaces <- function(v) {
  str_replace_all(v, " ", "_")
}
```

Dès lors, on peut appliquer cette fonction à l'ensemble de nos noms de variables :

```
df %>% rename_with(replace_spaces)
#> # A tibble: 8 x 8
#>   id    pcs  pcs_pere  pcs_mere    age  age_enf1  age_enf2  age_enf3
#>   <dbl> <chr> <chr>     <chr>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1    1  5      5        6       25      2      NA      NA
#> 2    2  3      3        2       45     12       8       2
#> 3    3  4      2        5       29      7      NA      NA
#> 4    4  2      1        4       32      6       3      NA
#> 5    5  1      4        3       65     39      36      28
#> 6    6  6      6        6       51     18      12      NA
#> 7    7  5      4        6       37      8       4       1
#> 8    8  3      3        1       42     16      10       5
```

Certain.es lectrices et lecteurs attentifs auront peut-être noté que le même résultat peut être obtenu en utilisant `replace_spaces()` avec la fonction `names()`.

```
names(df) <- replace_spaces(names(df))
```

L'avantage de `rename_with()` c'est qu'elle peut s'intégrer dans un pipeline de dplyr, et, comme nous allons le voir un peu plus loin, permet si nécessaire de n'appliquer cette transformation qu'à certaines colonnes seulement.

15.2 `across()` : appliquer des fonctions à plusieurs colonnes

15.2.1 Appliquer une fonction à plusieurs colonnes

On a défini précédemment une fonction qui recode les modalités d'une variable PCS et on a vu comment appliquer ce recodage à deux variables de `df`.

```
recode_pcs <- function(v) {
  fct_recode(v,
    "Agriculteur" = "1",
    "Indépendant" = "2",
    "Cadre" = "3",
    "Intermédiaire" = "4",
    "Employé" = "5",
    "Ouvrier" = "6"
  )
}

df %>%
  mutate(
    pcs = recode_pcs(pcs),
    pcs_mere = recode_pcs(pcs_mere)
  )
```

Supposons qu'on souhaite appliquer ce recodage à toutes les variables PCS de notre tableau. On pourrait évidemment créer autant de lignes que nécessaires dans notre `mutate`, mais on peut aussi utiliser la fonction `across()` de `dplyr`, qui facilite justement ce type d'opérations.

`across()` prend deux arguments principaux :

- la définition d'un ensemble de colonnes de notre tableau de données
- une ou plusieurs fonctions à appliquer aux colonnes sélectionnées

Il existe de nombreuses manières de définir les colonnes qu'on souhaite transformer : celles-ci sont en fait les mêmes que celles offertes par des verbes de `dplyr` comme `select()`.

Une première possibilité est d'utiliser `c()` en lui passant les noms des variables (on notera qu'on n'est pas obligés de mettre ces noms entre guillemets).

```
df %>%
  mutate(
    across(
      c(pcs, pcs_mere),
      recode_pcs
    )
  )
#> # A tibble: 8 x 8
#>   id pcs      pcs_pere pcs_mere   age `age enf1` `age enf2` `age enf3`
#>   <dbl> <fct>    <chr>     <dbl>     <dbl>     <dbl>     <dbl>
#> 1 1 Employé   Ouvrier     25        2       NA       NA
#> 2 2 Cadre     Indépenda~ 45       12        8       2
#> 3 3 Intermédiaire Employé~ 29        7       NA       NA
#> 4 4 Indépendant Intermédi~ 32        6        3       NA
#> 5 5 Agriculteur Cadre      65       39       36      28
#> 6 6 Ouvrier    Ouvrier     51       18       12       NA
#> 7 7 Employé   Ouvrier     37        8        4       1
#> 8 8 Cadre     Agriculter~ 42       16       10       5
```

Une autre possibilité est d'utiliser `:`, qui permet de définir une plage de colonnes en lui indiquant la colonne de début et la colonne de fin. Ainsi dans l'exemple suivant notre recodage est appliqué à toutes les colonnes situées entre `pcs` et `pcs_mere` (incluses).

```
df %>%
  mutate(
    across(
      pcs:pcs_pere,
      recode_pcs
    )
  )
#> # A tibble: 8 x 8
#>   id pcs      pcs_pere pcs_mere   age `age enf1` `age enf2` `age enf3`
#>   <dbl> <fct>    <fct>     <chr>     <dbl>     <dbl>     <dbl>
#> 1 1 Employé   Employé~   Cadre      25        2       NA       NA
#> 2 2 Cadre     Cadre      Indépenda~ 45       12        8       2
#> 3 3 Intermédiaire Indépenda~ 5 29        7       NA       NA
#> 4 4 Indépendant Agriculte~ 4 Intermédi~ 32        6        3       NA
#> 5 5 Agriculteur Intermédi~ 3 Cadre      65       39       36      28
#> 6 6 Ouvrier    Ouvrier    Ouvrier    51       18       12       NA
#> 7 7 Employé   Intermédi~ 6 Cadre      37        8        4       1
#> 8 8 Cadre     Cadre      1          42       16       10       5
```

On peut aussi sélectionner les variables via leurs noms. On peut ainsi choisir les variables qui commencent par une certaine chaîne de caractères via la fonction `starts_with()`, celles qui se terminent ou qui contiennent certains caractères avec `ends_with()` et `contains()`.

```
df %>%
  mutate(
    across(
```

```

        starts_with("pcs"),
        recode_pcs
    )
)
#> # A tibble: 8 x 8
#>   id pcs      pcs_pere  pcs_mere   age `age enf1` `age enf2` `age enf3`
#>   <dbl> <fct>    <fct>     <fct>     <dbl>      <dbl>      <dbl>      <dbl>
#> 1 1 Employé   Employé    Ouvrier    25         2       NA        NA
#> 2 2 Cadre     Cadre     Indépend~  Employé    45         12       8        2
#> 3 3 Intermédiaire  Indépend~  Agricult~ Interméd~  29         7       NA        NA
#> 4 4 Indépendant  Agricult~  Indépend~ Agricult~  32         6       3        NA
#> 5 5 Agriculteur  Interméd~  Cadre     Ouvrier    65         39       36       28
#> 6 6 Ouvrier    Ouvrier    Ouvrier    Ouvrier    51         18       12       NA
#> 7 7 Employé    Interméd~  Ouvrier    Ouvrier    37         8        4        1
#> 8 8 Cadre     Cadre     Agricult~  Agricult~  42         16       10       5

```

`across()` fonctionne dans un `mutate`, mais aussi dans un `summarise`. Dans l'exemple suivant, on calcule la moyenne de toutes les variables qui contiennent “enf”.

```

df %>%
  summarise(
    across(
      contains("enf"),
      mean
    )
  )
#> # A tibble: 1 x 3
#>   `age enf1` `age enf2` `age enf3`
#>   <dbl>       <dbl>       <dbl>
#> 1 13.5        NA         NA

```

De manière similaire, la fonction `num_range()` permet de sélectionner des colonnes ayant un préfixe commun suivi d'un indicateur numérique, comme `x1`, `x2...` Par exemple la syntaxe suivante sélectionnerait toutes les colonnes de `Q01` à `Q12` :

```
across(num_range("Q", 1:12, width = 2))
```

On peut également sélectionner des colonnes via une condition avec la fonction `where()`. Celle-ci prend elle-même en argument une fonction qui doit renvoyer `TRUE` ou `FALSE`, et ne conserve que les colonnes qui correspondent à des `TRUE`.

Dans l'exemple suivant, on applique la fonction `mean` seulement aux colonnes de `df` pour lesquelles la fonction `is.numeric` renvoie `TRUE`.

```

df %>%
  summarise(
    across(
      where(is.numeric),
      mean
    )
  )
#> # A tibble: 1 x 5
#>   id   age `age enf1` `age enf2` `age enf3`
#>   <dbl> <dbl>      <dbl>      <dbl>      <dbl>
#> 1 4.5  40.8      13.5       NA        NA

```

Pour des conditions plus complexes, on doit parfois définir soi-même la fonction passée à `where()`. Dans l'exemple suivant on calcule la moyenne uniquement pour les variables de `df` qui sont numériques et n'ont pas de valeurs manquantes.

```
no_na <- function(v) {
  is.numeric(v) && sum(is.na(v)) == 0
}

df %>%
  summarise(
    across(
      where(no_na),
      mean
    )
  )
#> # A tibble: 1 x 3
#>   id    age `age enf1`
#>   <dbl> <dbl>     <dbl>
#> 1  4.5  40.8     13.5
```

Il est même possible, pour les cas les plus complexes, de combiner plusieurs sélections avec les opérateurs `&`, `|` et `!`. L'exemple suivant applique la fonction `mean()` à toutes les colonnes numériques de `df`, sauf à la colonne `id`.

```
df %>%
  summarise(
    across(
      where(is.numeric) & !id,
      mean
    )
  )
#> # A tibble: 1 x 4
#>   age `age enf1` `age enf2` `age enf3`
#>   <dbl>     <dbl>     <dbl>     <dbl>
#> 1  40.8     13.5      NA        NA
```

Enfin, la fonction spéciale `everything()` permet de sélectionner la totalité des colonnes d'un tableau. Dans l'exemple suivant, on applique `n_distinct()` pour afficher le nombre de valeurs distinctes de toutes les variables de `df`.

```
df %>%
  summarise(
    across(
      everything(),
      n_distinct
    )
  )
#> # A tibble: 1 x 8
#>   id    pcs pcs_pere pcs_mere   age `age enf1` `age enf2` `age enf3`
#>   <int> <int>     <int>     <int> <int>     <int>     <int>     <int>
#> 1     8       6       6       6     8         8         7         5
```

Ces différentes manières de sélectionner un ensemble de colonnes sont appelées *tidy selection*. Il y a encore d'autres possibilités de sélection, pour avoir un aperçu complet on pourra se référer à la page de documentation de la fonction `select()`.



Une erreur de syntaxe fréquente est de mettre la sélection des colonnes dans l'appel à `across()`, mais pas la fonction qu'on souhaite appliquer.

Ainsi le code suivant générera une erreur :

```
mutate(across(pcs:pcs_mere), recode_pcs)
```

Il faut bien penser à passer la fonction comme argument du `across()`, donc à l'intérieur de ses parenthèses.

```
mutate(across(pcs:pcs_mere, recode_pcs))
```

15.2.2 Passer des arguments supplémentaires à la fonction appliquée

Par défaut, si on passe des arguments supplémentaires à `across()`, ils seront automatiquement transmis comme arguments à la fonction appliquée.

Dans l'exemple vu précédemment, on appliquait `mean()` à toutes les variables d'âge de `df`. Or comme certaines colonnes ont des valeurs manquantes, leur résultat vaut `NA`.

```
df %>%
  summarise(
    across(
      starts_with("age"),
      mean
    )
  )
```

Si on préfère que `mean()` soit appelée avec l'argument `na.rm = TRUE`, on pourrait définir explicitement une fonction à part qui utilise cet argument :

```
mean_sans_na <- function(x) {
  max(x, na.rm = TRUE)
}

df %>%
  summarise(
    across(
      starts_with("age"),
      mean_sans_na
    )
  )
```

Mais on peut faire plus simple, car tout argument supplémentaire passé à `across()` est transmis directement à la fonction appelée. Il est donc possible de faire :

```
df %>%
  summarise(
    across(
      starts_with("age"),
      max,
      na.rm = TRUE
    )
  )
#> # A tibble: 1 x 4
#>   age `age enf1` `age enf2` `age enf3`
#>   <dbl>     <dbl>     <dbl>     <dbl>
#> 1    65       39       36       28
```

15.2.3 Noms des colonnes créées par un `mutate`

Par défaut, lorsqu'on utilise `across()` dans un `mutate`, les nouvelles colonnes portent le même nom que les colonnes d'origine, ce qui signifie que ces dernières sont “écrasées” par les nouvelles valeurs.

Ainsi dans l'exemple suivant, les valeurs d'origine des colonnes PCS ont été écrasées par le résultat du recodage.

```
df %>%
  mutate(
    across(
      starts_with("pcs"),
      recode_pcs
    )
  )
```

Si on préfère créer de nouvelles colonnes, on doit indiquer la manière de les nommer en utilisant l'argument `.names` de `across()`. Celui prend comme valeur une chaîne de caractère dans laquelle le motif `{.col}` sera remplacé par le nom de la colonne d'origine.

Ainsi, si on souhaite plutôt que les variables recodées soient stockées dans de nouvelles colonnes nommées avec le suffixe `_rec`, on peut utiliser :

```
df %>%
  mutate(
    across(
      starts_with("pcs"),
      recode_pcs,
      .names = "{.col}_rec"
    )
  )
#> # A tibble: 8 x 11
#>   id pcs  pcs_pere pcs_mere  age `age` enf1` `age` enf2` `age` enf3` pcs_rec
#>   <dbl> <chr> <chr>     <chr>   <dbl>    <dbl>    <dbl>    <dbl> <fct>
#> 1  1 5    5       6        25      2       NA       NA Employé
#> 2  2 3    3       2        45      12      8        2 Cadre
#> 3  3 4    2       5        29      7       NA       NA Interméd~
#> 4  4 2    1       4        32      6       3        NA Indépend~
#> 5  5 1    4       3        65      39      36      28 Agricult~
#> 6  6 6    6       6        51      18      12      NA Ouvrier
#> 7  7 5    4       6        37      8       4        1 Employé
#> 8  8 3    3       1        42      16      10      5 Cadre
#> # ... with 2 more variables: pcs_pere_rec <fct>, pcs_mere_rec <fct>
```

15.2.4 Appliquer plusieurs fonctions à plusieurs colonnes

`across()` offre également la possibilité d'appliquer plusieurs fonctions à un ensemble de colonnes. Dans ce cas, plutôt que de lui passer une seule fonction comme deuxième argument, on lui passe une liste nommée de fonctions.

Le code suivant calcule le minimum et le maximum pour les variables d'âge de `df`.

```
df %>%
  summarise(
    across(
      starts_with("age"),
      ...
```

```

        list(minimum = min, maximum = max)
    )
}

#> # A tibble: 1 x 8
#>   age_minimum age_maximum `age_enf1_minimum` `age_enf1_maximum` `age_enf2_minimum` 
#>   <dbl>       <dbl>           <dbl>           <dbl>           <dbl>           <dbl>
#> 1      25       65             2              39            NA
#> # ... with 3 more variables: age_enf2_maximum <dbl>, age_enf3_minimum <dbl>,
#> #   age_enf3_maximum <dbl>

```

Par défaut les nouvelles variables sont nommées sous la forme `{nom_variable}_{nom_fonction}`, mais on peut personnaliser cette règle en ajoutant un argument `.names` à `across()`. Cet argument est une chaîne de caractères dans laquelle `{.col}` sera remplacé par le nom de la colonne courante, et `{.fn}` par le nom de la fonction.

```

df %>%
  summarise(
    across(
      starts_with("age"),
      list(minimum = min, maximum = max),
      .names = "{.fn}_{.col}"
    )
  )
#> # A tibble: 1 x 8
#>   minimum_age maximum_age `minimum_age_enf1` `maximum_age_enf1` `minimum_age_en~ 
#>   <dbl>       <dbl>           <dbl>           <dbl>           <dbl>
#> 1      25       65             2              39            NA
#> # ... with 3 more variables: maximum_age_enf2 <dbl>, minimum_age_enf3 <dbl>,
#> #   maximum_age_enf3 <dbl>

```

15.2.5 Renommer plusieurs colonnes avec une fonction

On a vu précédemment qu'on peut utiliser `rename_with()` pour renommer les colonnes d'un tableau de données à l'aide d'une fonction.

```

remplace_espaces <- function(v) {
  str_replace_all(v, " ", "_")
}

df %>% rename_with(remplace_espaces)

```

Par défaut, `rename_with()` applique la fonction de renommage à l'ensemble des colonnes du tableau. Il est cependant possible de lui indiquer de ne renommer que certaines de ces colonnes. Pour cela, on peut lui ajouter un argument supplémentaire nommé `.cols`, dont la syntaxe est exactement la même que pour `across()` ou `select()`.

Par exemple, le code suivant convertit en majuscule uniquement les noms des colonnes `id` et `poids`.

```

df %>%
  rename_with(str_to_upper, .cols = starts_with("pcs"))
#> # A tibble: 8 x 8
#>   id PCS  PCS_PERE PCS_MERE   age `age_enf1` `age_enf2` `age_enf3` 
#>   <dbl> <dbl>     <dbl>     <dbl>           <dbl>           <dbl>           <dbl>
#> 1      1      1         1         1             1             1             1
#> 2      2      2         2         2             2             2             2
#> 3      3      3         3         3             3             3             3
#> 4      4      4         4         4             4             4             4
#> 5      5      5         5         5             5             5             5
#> 6      6      6         6         6             6             6             6
#> 7      7      7         7         7             7             7             7
#> 8      8      8         8         8             8             8             8

```

```
#>   <dbl> <chr> <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1    1 5    5      6      25      2     NA     NA
#> 2    2 3    3      2      45     12      8      2
#> 3    3 4    2      5      29      7     NA     NA
#> 4    4 2    1      4      32      6      3     NA
#> 5    5 1    4      3      65     39     36     28
#> 6    6 6    6      6      51     18     12     NA
#> 7    7 5    4      6      37      8      4      1
#> 8    8 3    3      1      42     16     10      5
```

Et le code suivant remplace les espaces par des _ uniquement pour les colonnes dont le nom contient “enf”.

```
df %>%
  rename_with(remplace_espaces, .cols = contains("enf"))
#> # A tibble: 8 x 8
#>   id pcs  pcs_pere pcs_mere  age age_enf1 age_enf2 age_enf3
#>   <dbl> <chr> <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1    1 5    5      6      25      2     NA     NA
#> 2    2 3    3      2      45     12      8      2
#> 3    3 4    2      5      29      7     NA     NA
#> 4    4 2    1      4      32      6      3     NA
#> 5    5 1    4      3      65     39     36     28
#> 6    6 6    6      6      51     18     12     NA
#> 7    7 5    4      6      37      8      4      1
#> 8    8 3    3      1      42     16     10      5
```

15.3 Fonctions anonymes et syntaxes abrégées

Dans les sections précédentes, nous avons rencontré plusieurs fonctions, comme `rename_with()` ou `across()`, qui prennent une fonction en argument.

Par exemple, dans l'utilisation suivante de `rename_with()`, on avait créé une fonction `remplace_points()`.

```
remplace_espaces <- function(v) {
  str_replace_all(v, " ", "_")
}

df %>% rename_with(remplace_espaces)
```

Le fait de créer une fonction à part pour une opération d'une seule ligne ne se justifie pas forcément, surtout si on n'utilise pas cette fonction ailleurs dans notre code. Dans, ce cas, on peut définir notre fonction directement dans l'appel à `rename_with()` en utilisant une *fonction anonyme*, déjà introduites section 14.4.2.

```
df %>%
  rename_with(function(v) {
    str_replace_all(v, " ", "_")
})
```

Cette notation est assez pratique et souvent utilisée pour les fonctions à usage unique, ne serait-ce que pour s'économiser le fait de devoir lui trouver un nom pertinent.

La syntaxe étant un peu lourde, il existe deux alternatives permettant une définition plus “compacte”.

- La première alternative est propre aux packages du *tidyverse* (notamment **dplyr** et **purrr**), et ne fonctionnera pas pour les fonctions n'appartenant pas à ces packages. Il s'agit d'utiliser une syntaxe de type "formule" : le corps de la formule contient les instructions de la fonction, et les arguments sont nommés **.x** (ou **.**) s'il n'y en a qu'un, **.x** et **.y** s'il y en a deux, et **..1**, **..2**, etc. s'ils sont plus nombreux.
- La deuxième alternative est une syntaxe apparue avec la version 4.1 de R, qui permet de remplacer **function(...)** par le raccourci **\(...)**.

Ainsi les définitions suivantes sont équivalents :

```
# Fonctionne partout et tout le temps
function(v) { v + 2 }

# Fonctionne uniquement dans les fonctions du tidyverse
~ { .x + 2 }

# Fonctionne uniquement à partir de R 4.1
\_(v) { v + 2 }
```

De même que les définitions suivantes :

```
function(v1, v2) {
  res <- v1 / v2
  round(res, 1)
}

~ {
  res <- .x / .y
  round(res, 1)
}

\_(v1, v2) {
  res <- v1 / v2
  round(res, 1)
}
```

Quand la fonction anonyme est constituée d'une seule instruction, on peut supprimer les accolades dans sa définition.

```
function(x) x + 2
~ .x + 2
\_(x) x + 2
```

On pourra du coup, si on le souhaite, utiliser ces syntaxes compactes dans notre **rename_with()** pour définir notre fonction anonyme.

```
df %>%
  rename_with(~ str_replace_all(.x, " ", "_"))

df %>%
  rename_with( \_(x) str_replace_all(x, " ", "_") )
```

Cette syntaxe peut être utilisée partout où on peut passer une fonction comme argument et donc définir des fonctions anonymes. Dans cet exemple déjà vu précédemment, on passe la fonction **no_na** comme argument de **where()**.

```
no_na <- function(v) {
  is.numeric(v) && sum(is.na(v)) == 0
}

df %>%
  summarise(
    across(
      where(no_na),
      mean
    )
  )

```

On peut donc remplacer la fonction `no_na` par une fonction anonyme définie directement dans le `where()`.

```
df %>%
  summarise(
    across(
      where(function(v) { is.numeric(v) && sum(is.na(v)) == 0 }),
      mean
    )
  )

```

Et du coup utiliser une des deux syntaxes “compactes”.

```
df %>%
  summarise(
    across(
      where(~ is.numeric(.x) && sum(is.na(.x)) == 0),
      mean
    )
  )

df %>%
  summarise(
    across(
      where(\(v) is.numeric(v) && sum(is.na(v)) == 0),
      mean
    )
  )

```

15.4 `rowwise()` et `c_across()` : appliquer une transformation ligne par ligne

Soit le tableau de données suivant, qui contient des évaluations de restaurants sur quatre critères différents² :

```
restos <- tribble(
  ~nom, ~cuisine, ~decor, ~accueil, ~prix,
  "La bonne fourchette", 4, 2, 5, 4,
  "La choucroute de l'amer", 3, 3, 2, 3,
```

²Un nom de salon de coiffure s'est glissé dans cette liste de restaurants. Saurez-vous le retrouver ?

```

  "L'Hair de rien",           1,      4,      4,      3,
  "La blanquette de Vaulx", 5,      4,      4,      5,
)

restos
#> # A tibble: 4 x 5
#>   nom              cuisine decor accueil  prix
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>
#> 1 La bonne fourchette     4     2     5     4
#> 2 La choucroute de l'amer 3     3     2     3
#> 3 L'Hair de rien          1     4     4     3
#> 4 La blanquette de Vaulx 5     4     4     5

```

Imaginons qu'on souhaite faire la moyenne, pour chaque restaurant, des critères `decor` et `accueil`. On pourrait être tentés d'utiliser `mean()` de la manière suivante :

```

restos %>%
  mutate(
    decor_accueil = mean(c(decor, accueil))
  )
#> # A tibble: 4 x 6
#>   nom              cuisine decor accueil  prix decor_accueil
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>        <dbl>
#> 1 La bonne fourchette     4     2     5     4       3.5
#> 2 La choucroute de l'amer 3     3     2     3       3.5
#> 3 L'Hair de rien          1     4     4     3       3.5
#> 4 La blanquette de Vaulx 5     4     4     5       3.5

```

Si on regarde le résultat, on constate qu'il ne correspond pas à ce que l'on souhaite puisque toutes les valeurs sont les mêmes.

Que s'est-il passé ? En fait le `mutate` s'est appliqué sur la totalité du tableau. Ceci signifie que dans `mean(c(decor, accueil))`, les objets `decor` et `accueil` correspondent à la totalité des valeurs de chaque variable. On a donc concaténé ces deux vecteurs et calculé la moyenne, qui est du coup la même pour chaque ligne.

La valeur obtenue correspond aux résultat de :

```

mean(c(restos$decor, restos$accueil))
#> [1] 3.5

```

Ce que nous souhaitons ici, c'est calculer la moyenne non pas pour l'ensemble du tableau mais *pour chaque ligne*. Pour cela, on va utiliser la fonction `rowwise()` : celle-ci est équivalente à un `group_by()` qui créerait autant de groupes qu'il y a de lignes dans notre tableau.

```

restos %>% rowwise()
#> # A tibble: 4 x 5
#> # Rowwise:
#>   nom              cuisine decor accueil  prix
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>
#> 1 La bonne fourchette     4     2     5     4
#> 2 La choucroute de l'amer 3     3     2     3
#> 3 L'Hair de rien          1     4     4     3
#> 4 La blanquette de Vaulx 5     4     4     5

```

Quant notre tableau est groupé via un `rowwise()`, les opérations s'effectuent sur un tableau constitué uniquement de la ligne courante. Si on calcule la moyenne précédente, on obtient désormais le bon résultat.

```
restos %>%
  rowwise() %>%
  mutate(decor_accueil = mean(c(decor, accueil)))
#> # A tibble: 4 x 6
#> # Rowwise:
#>   nom              cuisine decor accueil  prix decor_accueil
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>      <dbl>
#> 1 La bonne fourchette     4     2       5     4      3.5
#> 2 La choucroute de l'amer 3     3       2     3      2.5
#> 3 L'Hair de rien          1     4       4     3      4
#> 4 La blanquette de Vaulx  5     4       4     5      4
```

Supposons qu'on souhaite désormais calculer la moyenne de l'ensemble des critères. On peut évidemment reprendre le code précédent en saisissant toutes les variables concernées.

```
restos %>%
  rowwise() %>%
  mutate(moyenne = mean(c(decor, accueil, cuisine, prix)))
#> # A tibble: 4 x 6
#> # Rowwise:
#>   nom              cuisine decor accueil  prix moyenne
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>      <dbl>
#> 1 La bonne fourchette     4     2       5     4      3.75
#> 2 La choucroute de l'amer 3     3       2     3      2.75
#> 3 L'Hair de rien          1     4       4     3      3
#> 4 La blanquette de Vaulx  5     4       4     5      4.5
```

Lister les variables de cette manière peut vite devenir pénible si le nombre de variables est important. C'est pourquoi `dplyr` propose la fonction `c_across()` : celle-ci permet de sélectionner des colonnes de la même manière que `select()` ou `across()`, et retourne un vecteur constitué des valeurs concaténées de ces colonnes.

L'exemple suivant calcule la moyenne de toutes les colonnes comprises entre `decor` et `prix`, en utilisant l'opérateur `:`.

```
restos %>%
  rowwise() %>%
  mutate(
    moyenne = mean(c_across(decor:prix))
  )
#> # A tibble: 4 x 6
#> # Rowwise:
#>   nom              cuisine decor accueil  prix moyenne
#>   <chr>            <dbl> <dbl>    <dbl> <dbl>      <dbl>
#> 1 La bonne fourchette     4     2       5     4      3.67
#> 2 La choucroute de l'amer 3     3       2     3      2.67
#> 3 L'Hair de rien          1     4       4     3      3.67
#> 4 La blanquette de Vaulx  5     4       4     5      4.33
```

Comme pour `across()` ou `select()`, on peut utiliser la fonction `where()` pour calculer la moyenne sur toutes les colonnes numériques.

```
restos %>%
  rowwise() %>%
  mutate(
    moyenne = mean(
      c_across(where(is.numeric)))
  )
)
#> # A tibble: 4 x 6
#> # Rowwise:
#>   nom                  cuisine decor accueil  prix moyenne
#>   <chr>                <dbl> <dbl>  <dbl> <dbl> <dbl>
#> 1 La bonne fourchette     4     2     5     4   3.75
#> 2 La choucroute de l'amer   3     3     2     3   2.75
#> 3 L'Hair de rien          1     4     4     3     3
#> 4 La blanquette de Vaulx    5     4     4     5   4.5
```

L'utilisation de `rowwise()` et `c_across()` est intéressante principalement quand il n'existe pas de fonction vectorisée pour la transformation qu'on souhaite appliquer. Quand elle existe, il est en général plus simple et plus rapide de l'utiliser.

Par exemple, pour trouver la valeur la plus élevée par restaurant, on pourrait être tenté d'utiliser le code suivant :

```
restos %>%
  rowwise() %>%
  summarise(note_max = max(c(decor, accueil)))
#> # A tibble: 4 x 1
#>   note_max
#>   <dbl>
#> 1     5
#> 2     3
#> 3     4
#> 4     4
```

Il est cependant plus lisible et plus efficace d'utiliser la fonction `pmax`, qui a justement pour objectif de parcourir des vecteurs en parallèle et de ne conserver que la plus grande valeur.

```
restos %>%
  summarise(note_max = pmax(decor, accueil))
#> # A tibble: 4 x 1
#>   note_max
#>   <dbl>
#> 1     5
#> 2     3
#> 3     4
#> 4     4
```

Une des limites de `pmax` cependant est qu'on ne peut pas l'utiliser avec `c_across()`, et qu'on ne peut donc pas faire de sélection des colonnes : on est obligés de saisir leurs noms.

```
restos %>%
  summarise(note_max = pmax(cuisine, decor, accueil, prix))
#> # A tibble: 4 x 1
```

```
#>   note_max
#>   <dbl>
#> 1      5
#> 2      3
#> 3      4
#> 4      5
```

Dans certains cas, notamment lorsque les colonnes sont nombreuses ou qu'on ne les a pas identifiées à l'avance, on pourra donc utiliser `rowwise()` et `c_across()` même quand des alternatives vectorisées existent.

```
restos %>%
  rowwise() %>%
  summarise(
    note_max = max(
      c_across(where(is.numeric)))
  )
)
#> # A tibble: 4 x 1
#>   note_max
#>   <dbl>
#> 1      5
#> 2      3
#> 3      4
#> 4      5
```

15.5 Ressources

[La page d'aide de la fonction select](#) (en anglais) liste toutes les possibilités offertes pour spécifier des ensembles de colonnes d'un tableau de données.

La vignette [Column-wise operations](#) de `dplyr` (en anglais) présente en détail l'utilisation et les fonctionnalités de `across()`.

La vignette [Row-wise operations](#) de `dplyr` (toujours en anglais) présente de manière approfondie l'utilisation de `rowwise()` et `c_across()` pour opérer individuellement sur les lignes d'un tableau de données.

15.6 Exercices

Pour certains des exercices qui suivent on utilisera le jeu de données `starwars` de `dplyr`. On peut le charger avec les instructions suivantes :

```
library(dplyr)
data(starwars)
```

Le jeu de données contient les caractéristiques de 87 personnages présents dans les films : espèce, âge, planète d'origine, etc.

15.6.1 Appliquer ses propres fonctions

Exercice 1.1

Créer une fonction `imc` qui prend en argument un vecteur `taille` (en cm) et un vecteur `poids` (en kg) et retourne les valeurs correspondantes de l'indice de masse corporelle, qui se calcule en divisant le poids en kilos par la taille en mètres au carré.

Utiliser cette fonction pour ajouter une nouvelle variable `imc` au tableau `starwars`.

À l'aide de `group_by()` et `summarise()`, utiliser à nouveau cette fonction pour calculer l'IMC moyen selon les valeurs de la variable `species`.

Exercice 1.2

Toujours dans le jeu de données `starwars`, à l'aide d'un `group_by()` et d'un `summarise()`, calculer pour chaque valeur de la variable `sex` la valeur de l'étendue de la variable `height` du jeu de données `starwars`, c'est-à-dire la différence entre sa valeur maximale et sa valeur minimale.

En partant du code précédent, créer une fonction `etendue` qui prend en argument un vecteur et retourne la différence entre sa valeur maximale et sa valeur minimale. En utilisant cette fonction, calculer pour chaque valeur de `sex` la valeur de l'étendue des variables `height` et `mass`.

Exercice 1.3

On a vu que la fonction suivante permet de calculer le pourcentage des éléments d'un vecteur de chaînes de caractères se terminant par un suffixe passé en argument.

```
prop_suffixe <- function(v, suffixe) {
  # On ajoute $ à la fin du suffixe pour capturer uniquement en fin de chaîne
  suffixe <- paste0(suffixe, "$")
  # Détection du suffixe
  nb_detect <- sum(str_detect(v, suffixe))
  # On retourne le pourcentage
  nb_detect / length(v) * 100
}
```

Modifier cette fonction en une fonction `prop_prefixe` qui retourne le pourcentage d'éléments commençant par un préfixe passé en argument. *Indication* : pour détecter si une chaîne commence par "ker", on utilise l'expression régulière "^ker".

Utiliser `prop_prefixe` dans un `summarise` appliqué à `rp2018` pour calculer le pourcentage de communes commençant par "Saint" selon le département. Ordonner les résultats par pourcentage décroissant.

Créer une fonction `tab_prefixe` qui prend un seul argument `prefixe` et renvoie le tableau obtenu à la question précédente pour le préfixe passé en argument. Tester avec `tab_prefixe("Plou")` et `tab_prefixe("Sch")`

Exercice 1.4

Le vecteur suivant donne, pour chacun des neuf principaux films de la saga *Star Wars*, la date à laquelle ils se déroulent dans l'univers de la saga.

```
c(
  "I"    = -32,
  "II"   = -22,
  "III"  = -19,
  "IV"   =  0,
  "V"    =  3,
  "VI"   =  4,
  "VII"  = 34,
  "VIII" = 34,
  "IX"   = 35
)
```

Dans le jeu de données `starwars`, la variable `birth_year` indique l'année de naissance du personnage en "années avant l'an zéro" (une valeur de 19 signifie donc une année de naissance de -19).

Créer une fonction `age_film` qui prend en entrée un vecteur d'années de naissance au même format que `birth_year` ainsi que l'identifiant d'un film, et calcule les âges à la date du film.

Vérifier avec :

```
age_film(starwars$birth_year, "IV")
#> [1] 19.0 112.0 33.0 41.9 19.0 52.0 47.0 NA 24.0 57.0 41.9 64.0
#> [13] 200.0 29.0 44.0 600.0 21.0 NA 896.0 82.0 31.5 15.0 53.0 31.0
#> [25] 37.0 41.0 48.0 NA 8.0 NA 92.0 NA 91.0 52.0 NA NA
#> [37] NA NA NA 62.0 72.0 54.0 NA 48.0 NA NA NA 72.0
#> [49] 92.0 NA NA NA NA 22.0 NA NA NA 82.0 NA
#> [61] 58.0 40.0 NA 102.0 67.0 66.0 NA NA NA NA NA
#> [73] NA NA
#> [85] NA NA 46.0
```

Utiliser la fonction pour ajouter deux nouvelles variables au tableau `starwars` : `age_iv` qui correspond à l'âge (potentiel) au moment du film IV, et `age_ix` qui correspond à l'âge au moment du film IX.

15.6.2 `across()`

Exercice 2.1

Reprendre la fonction `etendue` de l'exercice 1.2 :

```
etendue <- function(v) {
  max(v, na.rm = TRUE) - min(v, na.rm = TRUE)
}
```

Dans le jeu de données `starwars`, calculer l'étendue des variables `height` et `mass` pour chaque valeur de `sex` à l'aide de `group_by()`, `summarise()` et `across()`.

Toujours à l'aide d'`across()`, appliquer `etendue` à toutes les variables numériques, toujours pour chaque valeur de `sex`.

En utilisant `&` et `!`, appliquer `etendue` à toutes les variables numériques sauf à celles qui finissent par “year”.

Exercice 2.2

Dans le jeu de données `starwars`, appliquer en un seul `summarise` les fonctions `min` et `max` aux variables `height` et `mass`.

Si vous ne l'avez pas déjà fait à la question précédente, modifier le code pour que le calcul des valeurs minimales et maximales ne prennent pas en compte les valeurs manquantes.

Exercice 2.3

Dans le jeu de données `hdv2003`, utiliser `across()` pour transformer les modalités “Oui” et “Non” en `TRUE` et `FALSE` pour toutes les variables de `hard.rock` à `sport`.

Ajouter un argument `.names` à `across()` pour que les variables recodées soient stockées dans de nouvelles colonnes nommées avec le suffixe `_true`.

15.6.3 Fonctions anonymes et notations abrégées

Exercice 3.1

Dans un exercice précédent, on a vu que le code ci-dessous permet de calculer l'étendue des variables `height` et `mass` du jeu de données `starwars`.

```
etendue <- function(v) {  
  max(v, na.rm = TRUE) - min(v, na.rm = TRUE)  
}  
  
starwars %>%  
  group_by(sex) %>%  
  summarise(  
    across(  
      c(height, mass),  
      etendue  
    )  
  )
```

Modifier ce code en supprimant la définition de `etendue` et en utilisant à la place une fonction anonyme directement dans le `across()`.

Modifier à nouveau ce code pour utiliser la syntaxe abrégée de type "formule" du *tidyverse*.

Exercice 3.2

Soit le code suivant, qui renomme les colonnes du tableau `starwars` de type liste en leur ajoutant le préfixe "liste".

```
ajoute_prefixe_liste <- function(nom) {  
  paste0("liste_", nom)  
}  
  
starwars %>%  
  rename_with(ajoute_prefixe_liste, .cols = where(is.list))
```

Réécrire ce code avec une fonction anonyme en utilisant les trois notations :

- classique (avec `function()`)
 - formule (du *tidyverse*)
 - compacte (à partir de R 4.1)

Exercice 3.3

Le code suivant indique, pour chaque région du jeu de données rp2018, le nom de la commune ayant la valeur maximale pour les variables `dipl aucun` et `dipl sup`.

```
nom_commune_max <- function(valeurs, communes) {
  communes[valeurs == max(valeurs)]
}

rp2018 %>%
  group_by(region) %>%
  summarise(
    across(
      c(dipl_aucun, dipl_sup),
      nom_commune_max,
      commune
    )
  )
#> # A tibble: 17 x 3
#>   region           dipl_aucun      dipl_sup
#>   <fct>             <dbl>        <dbl>
```

```
#> <chr> <chr> <chr>
#> 1 Auvergne-Rhône-Alpes Dyonnax Corenc
#> 2 Bourgogne-Franche-Comté Saint-Loup-sur-Semouse Fontaine-lès-Dijon
#> 3 Bretagne Louvigné-du-Désert Saint-Grégoire
#> 4 Centre-Val de Loire La Loupe Olivet
#> 5 Corse Ghisonaccia Ville-di-Pietrabugno
#> 6 Grand Est Behren-lès-Forbach Mittelhausbergen
#> 7 Guadeloupe Saint-Louis Le Gosier
#> 8 Guyane Papaïchton Remire-Montjoly
#> 9 Hauts-de-France Bohain-en-Vermandois La Madeleine
#> 10 Île-de-France Clichy-sous-Bois Paris 5e Arrondissement
#> 11 La Réunion Cilaos La Possession
#> 12 Martinique Basse-Pointe Schœlcher
#> 13 Normandie Sourdeval Mont-Saint-Aignan
#> 14 Nouvelle-Aquitaine Aiguillon Bordeaux
#> 15 Occitanie Bessèges Montferrier-sur-Lez
#> 16 Pays de la Loire Saint-Calais Nantes
#> 17 Provence-Alpes-Côte d'Azur Marseille 15e Arrondissement Le Tholonet
```

Réécrire ce code en utilisant une fonction anonyme, avec la syntaxe de votre choix (classique, formule ou compacte).

À l'aide d'une fonction anonyme supplémentaire, modifier le code pour qu'il retourne également, pour les mêmes variables, le nom des communes avec les valeurs minimales.

15.6.4 `rowwise()` et `c_across()`

Exercice 4.1

On repart du code final de l'exercice 2.3, qui recodait une série de variables de `hdv2003` en valeurs TRUE/FALSE dans de nouvelles variables avec le suffixe `_true`.

```
detecte_oui <- function(v) {
  v == "Oui"
}
hdv2003 <- hdv2003 %>%
  mutate(
    across(
      hard.rock:sport,
      detecte_oui,
      .names = "{.col}_true"
    )
  )
```

Calculer le plus simplement possible une nouvelle variable `total` qui contient, pour chaque ligne, le nombre de valeurs TRUE des deux variables `cinema_true` et `sport_true` (si une ligne contient TRUE pour ces deux variables, `total` doit valoir 2, etc.)

Recalculer la variable `total` pour qu'elle contienne le nombre de TRUE par ligne pour les variables `bricol_true`, `cinema_true` et `sport_true`.

Recalculer la variable `total` pour qu'elle contienne le nombre de TRUE par ligne pour toutes les variables se terminant par `_true`.

Reprendre le code précédent pour qu'il puisse s'appliquer directement sur les variables `hard.rock...sport`, sans passer par le recodage en TRUE/FALSE.

Exercice 4.2

Dans le jeu de données `starwars`, la colonne `films` contient la liste des films dans lesquels apparaissent les différents personnages. Cette colonne a une forme un peu particulière puisqu'il s'agit d'une “colonne-liste” : les éléments de cette colonne sont eux-mêmes des listes.

```
head(starwars$films, 3)
#> [[1]]
#> [1] "The Empire Strikes Back" "Revenge of the Sith"
#> [3] "Return of the Jedi"      "A New Hope"
#> [5] "The Force Awakens"
#>
#> [[2]]
#> [1] "The Empire Strikes Back" "Attack of the Clones"
#> [3] "The Phantom Menace"      "Revenge of the Sith"
#> [5] "Return of the Jedi"      "A New Hope"
#>
#> [[3]]
#> [1] "The Empire Strikes Back" "Attack of the Clones"
#> [3] "The Phantom Menace"      "Revenge of the Sith"
#> [5] "Return of the Jedi"      "A New Hope"
#> [7] "The Force Awakens"
```

On essaye de calculer le nombre de films pour chaque personnage avec le code suivant. Est-ce que ça fonctionne ? Pourquoi ?

```
starwars %>%
  mutate(n_films = length(films))
```

Trouver une manière d'obtenir le résultat attendu.

Chapitre 16

Structures de données

R propose de nombreuses structures de données différentes, et les extensions peuvent en implémenter de nouvelles. Cette section introduit trois structures parmi les plus utilisées : les **vecteurs atomiques**, les **listes** et les **tableaux de données**. Certaines ont déjà été abordées et utilisées précédemment, mais connaître leurs spécificités et savoir les manipuler est utile voire indispensable, notamment lorsqu'on veut créer ses propres fonctions.

16.1 Vecteurs atomiques

Les vecteurs atomiques sont des structures qui regroupent ensemble plusieurs éléments constitués d'une seule valeur, avec deux contraintes : ces valeurs doivent toutes être du même type. Les vecteurs atomiques ont déjà été introduits section 9.1.

16.1.1 Crédation d'un vecteur

On peut construire un vecteur manuellement avec la fonction `c()`.

```
x <- c(1, 3, 8)
```

Si on souhaite générer un vecteur de valeurs entières successives, on peut utiliser l'opérateur `:` ou la fonction `seq_len()`.

```
2:8  
#> [1] 2 3 4 5 6 7 8  
seq_len(5)  
#> [1] 1 2 3 4 5
```

La fonction `seq()` permet de générer des séquences régulières plus complexes.

```
seq(0.5, 2.5, by = 0.5)  
#> [1] 0.5 1.0 1.5 2.0 2.5  
seq(0, 4, length.out = 6)  
#> [1] 0.0 0.8 1.6 2.4 3.2 4.0
```

Une autre variante de `seq()`, nommée `seq_along()`, permet de générer un vecteur d'entiers correspondant à la longueur d'un objet passé en argument :

```
x <- c("Pomme", "Poire")
seq_along(x)
#> [1] 1 2
y <- runif(10)
seq_along(y)
#> [1] 1 2 3 4 5 6 7 8 9 10
```

Enfin, la fonction `rep()` permet de répéter un élément ou un vecteur.

```
rep("Pomme", 6)
#> [1] "Pomme" "Pomme" "Pomme" "Pomme" "Pomme" "Pomme"
rep(1:4, 2)
#> [1] 1 2 3 4 1 2 3 4
```

Si on souhaite connaître le nombre d'éléments d'un vecteur, on peut utiliser la fonction `length()`.

```
v <- rep(1:4, 2)
length(v)
#> [1] 8
```

Il peut parfois être utile de créer des vecteurs “vides”. Dans ce cas on peut les initialiser avec les fonctions `vector()`, `character()` ou `numeric()`. Par défaut ces fonctions renvoient un vecteur sans élément, mais on peut aussi leur indiquer en argument le nombre d'éléments souhaités (qui seront alors initialisés avec une valeur par défaut).

```
numeric()
#> numeric(0)
character(2)
#> [1] "" ""
```

16.1.2 Vecteurs nommés

Les éléments d'un vecteur peuvent être nommés. Ces noms peuvent être définis au moment de la création du vecteur.

```
x <- c(e1 = 1, e2 = 3, e3 = 8)
x
#> e1 e2 e3
#> 1 3 8
```

On peut utiliser `names()` pour récupérer les noms des éléments d'un vecteur.

```
names(x)
#> [1] "e1" "e2" "e3"
```

On peut aussi utiliser `names()` pour créer ou modifier les noms d'un vecteur existant.

```
names(x) <- c("brouette", "moto", "igloo")
x
#> brouette     moto      igloo
#>     1         3         8
```

16.1.3 Types de vecteurs

On peut déterminer le type d'un vecteur avec l'instruction `typeof`.

```
x <- c(1, 3, 8)
typeof(x)
#> [1] "double"
y <- c("foo", "bar", "baz")
typeof(y)
#> [1] "character"
z <- c(TRUE, FALSE, FALSE)
typeof(z)
#> [1] "logical"
```

Parmi les principaux types de données on notera¹ :

- les chaînes de caractères (`character`)
- les nombres flottants (`double`)
- les nombres entiers (`integer`)
- les valeurs logiques (`logical`)

À noter que par défaut les nombres sont considérés comme des nombres flottants (des nombres décimaux avec une virgule) : pour les définir explicitement comme nombres entiers on peut leur ajouter le suffixe L.

```
x <- c(1L, 3L, 8L)
typeof(x)
#> [1] "integer"
```

On peut tester le type d'un vecteur avec les fonctions `is.character`, `is.double`, `is.logical`... Autre fonction utile, `is.numeric` teste si un vecteur est de type `double` ou `integer`.

```
x <- c(1, 3, 8)
is.numeric(x)
#> [1] TRUE
x > 2
#> [1] FALSE  TRUE   TRUE
is.logical(x > 2)
#> [1] TRUE
y <- c("foo", "bar", "baz")
is.character(y)
#> [1] TRUE
```

Petite spécificité, les facteurs (voir section 9.3.1) ne sont pas considérés par R comme des `character`, même s'ils comportent des chaînes de caractères. Pour tester si un vecteur est de type facteur, on utilise `is.factor()`.

¹Il en existe d'autres, comme `complex` ou `raw`, mais qui sont moins fréquemment utilisés.

```
fac <- factor(c("rouge", "vert", "rouge"))
is.character(fac)
#> [1] FALSE
is.factor(fac)
#> [1] TRUE
```

Tous les éléments d'un vecteur doivent être du même type. Si ça n'est pas le cas, les éléments seront convertis au type le plus "général" présent dans le vecteur, sachant que les `character` sont plus généraux que les `numeric`, qui sont eux-mêmes plus généraux que les `logical`.

Dans l'exemple suivant, le nombre 1 est transformé en chaîne de caractère "1".

```
c(1, "foo")
#> [1] "1"    "foo"
```

Si on mélange nombres et valeurs logiques, les `TRUE` sont convertis en 1 et les `FALSE` en 0.

```
c(TRUE, 2, FALSE)
#> [1] 1 2 0
```



Si la valeur `NA`, comme on l'a vu, permet d'indiquer une valeur manquante (`Not Available`), il existe en réalité plusieurs types de `NA`, même si cette distinction est la plupart du temps transparente pour l'utilisateur. On a ainsi notamment des valeurs `NA_integer_`, `NA_character_`, `NA_real_`.

La conversion automatique d'un type en un autre est à l'origine d'un idiom courant en R. Quand on applique une fonction qui attend un vecteur de nombres à un vecteur de valeurs logiques, celles-ci sont automatiquement converties, les `TRUE` devenant 1 et les `FALSE` devenant 0. Du coup, si on applique `sum()` à un vecteur de valeurs logiques, le résultat est égal au nombre de valeurs `TRUE`.

```
sum(c(TRUE, FALSE, TRUE))
#> [1] 2
```

On peut donc appliquer `sum()` à un test, et on obtiendra le nombre de valeurs pour lesquelles le test est vrai.

```
x <- c(1, 5, 8, 12, 14)
sum(x > 10)
#> [1] 2
```

Ceci fournit un raccourci très pratique. Dans l'exemple suivant, on tire 1000 nombres au hasard entre 0 et 1 et on calcule le nombre de valeurs obtenues qui sont inférieures à 0.5.

```
x <- runif(1000)
sum(x < 0.5)
#> [1] 513
```

Autre raccourci moins utilisé, appliquer `mean()` au résultat d'un test donne la proportion de valeurs pour lesquelles le test est vrai.

```
x <- c(1, 5, 8, 12, 14)
mean(x > 10)
#> [1] 0.4
x <- runif(1000)
mean(x < 0.5)
#> [1] 0.522
```

On peut convertir un vecteur d'un type à un autre avec les fonctions `as.character()`, `as.numeric()` et `as.logical()`. Si une valeur ne peut pas être convertie, elle est remplacée par un NA, et R affiche un avertissement.

```
as.character(1:3)
#> [1] "1" "2" "3"
as.logical(c(0, 2, 4))
#> [1] FALSE TRUE TRUE
as.numeric(c("foo", "23"))
#> Warning: NAs introduced lors de la conversion automatique
#> [1] NA 23
```

16.1.4 Sélection d'éléments

On a vu section 9.1 que l'opérateur `[]` peut être utilisé pour sélectionner des éléments d'un vecteur. Cet opérateur peut comporter :

- des nombres (qui sélectionnent par position)
- des chaînes de caractères (qui sélectionnent par nom)
- un test ou des valeurs logiques (qui sélectionnent les éléments correspondant à TRUE)

```
x <- c(e1 = 1, e2 = 2, e3 = 8, e4 = 12)
x[c(1, 4)]
#> e1 e4
#> 1 12
x[c("e2", "e4")]
#> e2 e4
#> 2 12
x[x < 10]
#> e1 e2 e3
#> 1 2 8
```

Si on fournit à `[]` un ou plusieurs nombres négatifs, les valeurs correspondantes seront supprimées plutôt que sélectionnées.

```
x[-1]
#> e2 e3 e4
#> 2 8 12
x[c(-2, -4)]
#> e1 e3
#> 1 8
```

Si on souhaite afficher les premières ou dernières valeurs d'un vecteur, les fonctions `head()` et `tail()` peuvent être utiles.

```
head(x, 2)
#> e1 e2
#> 1 2
tail(x, 1)
#> e4
#> 12
```

16.1.5 Modification

Utilisé conjointement avec l'opérateur d'assignation `<-`, l'opérateur `[]` permet de remplacer des éléments.

```
x <- c(e1 = 1, e2 = 2, e3 = 8, e4 = 12)
x[1] <- -1000
x
#>   e1    e2    e3    e4
#> -1000     2     8    12
x["e2"] <- 0
x
#>   e1    e2    e3    e4
#> -1000     0     8    12
x[x > 10] <- NA
x
#>   e1    e2    e3    e4
#> -1000     0     8    NA
```

Utilisé sans arguments, `[]` se contente de renvoyer le vecteur entier. Mais couplé à une assignation, il remplace chacun des éléments du vecteur plutôt que le vecteur lui-même.

```
x[] <- 3
x
#> e1 e2 e3 e4
#> 3 3 3 3
```

16.2 Listes

Les listes sont une généralisation des vecteurs : elles regroupent également plusieurs éléments ensemble, mais ceux-ci peuvent être de n'importe quel type, y compris des objets complexes. Une liste peut donc contenir des vecteurs, des listes, des tableaux de données, des fonctions, des graphiques `ggplot2` stockés dans un objet, etc.

16.2.1 Crédation

On construit une liste avec la fonction `list`.

```
list(1, "foo", c("Pomme", "Citron"))
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "foo"
#>
#> [[3]]
#> [1] "Pomme"  "Citron"
```

L'affichage du contenu d'une liste dans la console diffère de celui d'un vecteur. Dans le cas d'une liste les éléments sont affichés les uns en dessous des autres, et séparés par leur indice numérique entre une paire de crochets. Dans l'affichage ci-dessus, il faut bien distinguer les `[[1]]`, `[[2]]` et `[[3]]`, qui correspondent au numéro de l'élément de la liste, et les `[1]` qui font partie de l'affichage du contenu de ces éléments.

Comme pour les vecteurs, on peut nommer les éléments à la création de la liste.

```
liste <- list(nombre = 1, char = "foo", vecteur = c("Pomme", "Citron"))
liste
#> $nombre
#> [1] 1
#>
#> $char
#> [1] "foo"
#>
#> $vecteur
#> [1] "Pomme"   "Citron"
```

Dans ce cas l'affichage de la liste dans la console montre ces noms plutôt que les indices numériques des éléments. Comme pour les vecteurs atomiques, on peut utiliser `names()` pour afficher ou modifier les noms des éléments.

```
names(liste)
#> [1] "nombre"   "char"      "vecteur"
```

Quand la liste est plus complexe, l'affichage peut vite devenir illisible.

```
liste <- list(
  l2 = list(x = 1:10, y = c("Pomme", "Citron")),
  df = data.frame(v1 = 2:5, v2 = LETTERS[2:5]),
  y = runif(10)
)
liste
#> $l2
#> $l2$x
#> [1] 1 2 3 4 5 6 7 8 9 10
#>
#> $l2$y
#> [1] "Pomme"   "Citron"
#>
#>
#> $df
#>   v1 v2
#> 1  2  B
#> 2  3  C
#> 3  4  D
#> 4  5  E
#>
#> $y
#> [1] 0.69901501 0.09206802 0.12078778 0.72223653 0.09476776 0.38314043
#> [7] 0.48006825 0.65483899 0.12235615 0.33985941
```

Dans ce cas la fonction `str` peut être utile pour afficher de manière plus compacte la structure de la liste. Dans cet exemple elle permet de voir un peu plus clairement que `x` et `y` sont des éléments d'une sous-liste `l2`.

```

str(liste)
#> List of 3
#> $ l2:List of 2
#>   ..$ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
#>   ..$ y: chr [1:2] "Pomme" "Citron"
#> $ df:'data.frame': 4 obs. of 2 variables:
#>   ..$ v1: int [1:4] 2 3 4 5
#>   ..$ v2: chr [1:4] "B" "C" "D" "E"
#> $ y : num [1:10] 0.699 0.0921 0.1208 0.7222 0.0948 ...

```

16.2.2 Ajout d'éléments

Attention, si on souhaite ajouter un nouvel élément à une liste, il ne faut pas utiliser à nouveau `list()`, car dans ce cas notre liste de départ est insérée comme une “sous-liste”.

```

liste <- list(e1 = 1:3, e2 = "Chihuhua")
liste2 <- list(liste, nouveau = 100)
str(liste2)
#> List of 2
#> $           :List of 2
#>   ..$ e1: int [1:3] 1 2 3
#>   ..$ e2: chr "Chihuhua"
#> $ nouveau: num 100

```

Il faut à la place utiliser `c()`, comme pour les vecteurs.

```

liste3 <- c(liste, nouveau = 100)
str(liste3)
#> List of 3
#> $ e1      : int [1:3] 1 2 3
#> $ e2      : chr "Chihuhua"
#> $ nouveau: num 100

```

`c()` permet aussi de “concaténer” deux listes existantes en une seule.

```

liste1 <- list(a = 1, b = 2)
liste2 <- list(x = 3, y = 4)
c(liste1, liste2)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
#>
#> $x
#> [1] 3
#>
#> $y
#> [1] 4

```

16.2.3 Sélection d'éléments

Il y a deux opérateurs différents qui permettent de sélectionner les éléments d'une liste : les crochets simples `[]` et les crochets doubles `[[]]`. La différence entre ces deux opérateurs est souvent source de confusion.

Partons de la liste suivante :

```
liste <- list(1:5, "foo", c("Pomme", "Citron"))
liste
#> [[1]]
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "foo"
#>
#> [[3]]
#> [1] "Pomme"   "Citron"
```

Si on utilise les crochets simples pour sélectionner le premier élément de cette liste, on obtient le résultat suivant :

```
liste[1]
#> [[1]]
#> [1] 1 2 3 4 5
```

On notera que le résultat est une liste à un seul élément.

Si on utilise les crochets doubles :

```
liste[[1]]
#> [1] 1 2 3 4 5
```

On obtient cette fois-ci non pas une liste composée du premier élément, mais le contenu de ce premier élément.



La différence est importante, mais pas toujours facile à retenir. On peut utiliser deux petites astuces mnémotechniques :

- si une liste est un train composé de plusieurs wagons, `[1]` retourne le premier wagon du train, tandis que `[[1]]` renvoie le contenu du premier wagon.
- une alternative est de considérer que `[[[]]]` va chercher “plus profondément” que `[]`.

Un autre point important est que si on passe plusieurs éléments à `[[[]]]`, la sélection se fait d'une manière récursive peu intuitive et source d'erreurs. Il est donc conseillé de **toujours utiliser `[[[]]]` avec un seul argument**, et d'utiliser `[]` si on souhaite sélectionner plusieurs éléments d'une liste.

```
liste[c(1, 2)]
#> [[1]]
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "foo"
```



En résumé :

- si on souhaite récupérer uniquement le contenu d'un élément d'une liste, on utilise `[[]]` avec un seul argument.
- si on souhaite récupérer une nouvelle liste en sélectionnant des éléments de notre liste actuelle, on utilise `[]` avec un ou plusieurs arguments.



Comme pour les vecteurs, on peut utiliser des nombres négatifs avec `[]` pour exclure des éléments plutôt que les sélectionner, et on peut également utiliser les fonctions `head()` et `tail()`.

Si la liste est nommée, on peut sélectionner des éléments par noms avec les deux opérateurs.

```
liste <- list(nombre = 1, char = "foo", vecteur = c("Pomme", "Citron"))
liste[["nombre", "char"]]
#> $nombre
#> [1] 1
#>
#> $char
#> [1] "foo"
liste[["vecteur"]]
#> [1] "Pomme"   "Citron"
```

On peut aussi utiliser l'opérateur `$`, qui équivaut à `[[]]` :

```
liste$vecteur
#> [1] "Pomme"   "Citron"
```

16.2.4 Modification

Comme pour les vecteurs, on peut utiliser l'opérateur `[]` et l'opérateur d'assignation `<-` pour modifier des éléments d'une liste.

```
liste <- list(nombre = 1:5, char = "foo", vecteur = c("Pomme", "Citron"))
liste["nombre"] <- "first"
liste
#> $nombre
#> [1] "first"
#>
#> $char
#> [1] "foo"
#>
#> $vecteur
#> [1] "Pomme"   "Citron"
```

```
liste[c(1, 3)] <- 0
liste
#> $nombre
#> [1] 0
#>
#> $char
#> [1] "foo"
#>
#> $vecteur
#> [1] 0
```



Attention à ne pas utiliser les crochets doubles pour modifier des éléments d'une liste car ceux-ci peuvent avoir un comportement inattendu si on veut modifier plusieurs éléments d'un coup.

Enfin, on peut supprimer un ou plusieurs éléments d'une liste, en leur attribuant la valeur `NULL`².

```
liste <- list(nombre = 1:5, char = "foo", vecteur = c("Pomme", "Citron"))
liste$char <- NULL
liste
#> $nombre
#> [1] 1 2 3 4 5
#>
#> $vecteur
#> [1] "Pomme"  "Citron"
```

16.2.5 Utilisation

En tant que généralisation des vecteurs atomiques, les listes sont utiles dès qu'on souhaite regrouper des éléments complexes ou hétérogènes.

On les utilisera par exemple pour retourner plusieurs résultats depuis une fonction.

```
indicateurs <- function(x) {
  list(
    moyenne = mean(x),
    variance = var(x)
  )
}

x <- 1:10
res <- indicateurs(x)
res$moyenne
#> [1] 5.5
res$variance
#> [1] 9.166667
```

On utilise également les listes pour stocker des objets complexes et leur appliquer des fonctions. Ce fonctionnement sera abordé en détail dans la section 18, mais en guise de petit aperçu, l'exemple fictif suivant récupère les noms de tous les fichiers CSV du répertoire courant et les importe tous dans une liste à l'aide de `purrr::map()` et de `read_csv()`.

²Si on veut ajouter un élément `NULL` à une liste, il faut utiliser les crochets simples avec la syntaxe `liste["foo"] <- list(NULL)`.

```
files <- list.files(pattern = "*.csv")
dfs <- purrr::map(files, read_csv)
```

On pourra ensuite utiliser cette liste de tableaux pour leur appliquer des transformations ou les fusionner.

16.3 Tableaux de données (*data frame* et *tibble*)

On a déjà utilisé les tableaux de données à de nombreux reprises en manipulant des *data frames* ou des *tibbles*. Les seconds sont une variante des premiers, les différences entre les deux ayant été abordées section 6.4.

Un tableau de données est en réalité une liste nommée de vecteurs atomiques avec une contrainte spécifique : ces vecteurs doivent tous être de même longueur, ce qui garantit le format “tabulaire” des données.

16.3.1 Crédation

Un tableau de données est le plus souvent créé en important des données depuis un fichier au format CSV, tableur ou autre. On peut cependant créer un *data frame* manuellement via la fonction `data.frame()` :

```
df <- data.frame(
  fruit = c("Pomme", "Pomme", "Citron"),
  poids = c(154, 167, 92),
  couleur = c("vert", "vert", "jaune")
)
```

On peut aussi créer un *tibble* manuellement avec la fonction `tibble()`. La syntaxe est la même que celle de `data.frame()`, mais avec un comportement un peu différent : notamment, les noms comportant des espaces ou des caractères spéciaux sont conservés tels quels.

La fonction `tribble()` permet de créer un tibble manuellement avec une syntaxe “par ligne” qui peut être un peu plus lisible.

```
df_trib <- tribble(
  ~fruit, ~poids, ~couleur,
  "Pomme", 154, "vert",
  "Pomme", 167, "vert",
  "Citron", 92, "jaune"
)
```

On peut convertir un *data frame* en *tibble* avec la fonction `as_tibble()`.

```
df_tib <- as_tibble(df)
df_tib
#> # A tibble: 3 x 3
#>   fruit  poids couleur
#>   <chr> <dbl> <chr>
#> 1 Pomme    154  vert
#> 2 Pomme    167  vert
#> 3 Citron     92  jaune
```

16.3.2 Noms de colonnes et de lignes

On peut lister et modifier les noms des colonnes d'un tableau avec les fonctions `names()` ou `colnames()` (qui sont équivalentes).

```
names(df)
#> [1] "fruit"    "poids"    "couleur"
colnames(df)
#> [1] "fruit"    "poids"    "couleur"
```

On peut attribuer des noms aux lignes d'un *data frame* à l'aide de la fonction `rownames()`. Attention cependant, les noms de ligne ne sont (volontairement) pas pris en charge par les *tibbles*.

```
rownames(df) <- c("fruit1", "fruit2", "fruit3")
rownames(df)
#> [1] "fruit1" "fruit2" "fruit3"
```

```
rownames(df_tib) <- c("fruit1", "fruit2", "fruit3")
#> Warning: Setting row names on a tibble is deprecated.
```

Si on souhaite conserver des noms de ligne en passant d'un *data frame* à un *tibble*, il faut les stocker dans une nouvelle colonne, soit en la créant manuellement soit avec la fonction `rownames_to_column()` (qui a l'avantage de placer la nouvelle colonne en première position du tableau).

```
rownames_to_column(df, "name")
#>   name fruit poids couleur
#> 1 fruit1 Pomme 154     vert
#> 2 fruit2 Pomme 167     vert
#> 3 fruit3 Citron  92      jaune
```

16.3.3 Sélection de lignes et de colonnes

On a déjà vu dans les parties précédentes plusieurs manières de sélectionner des éléments dans un tableau de données.

Ainsi, on peut sélectionner une colonne via l'opérateur `$`.

```
df$fruit
#> [1] "Pomme"  "Pomme"  "Citron"
```

Comme un tableau de données est en réalité une liste de colonnes, on peut aussi utiliser l'opérateur `[]` pour sélectionner l'une de ses colonnes, par position ou par nom³.

³Attention, comme pour les listes, à ne pas utiliser `[]` avec un argument de longueur supérieur à 1, car cela mène soit à des erreurs soit à des résultats contre-intuitifs.

```
df[["fruit"]]
#> [1] "Pomme"   "Pomme"   "Citron"
df[[2]]
#> [1] 154 167 92
```

On peut utiliser `head()` et `tail()` avec un tableau de données : dans ce cas ces fonctions retourneront les premières ou dernières *lignes* du tableau.

```
head(df, 2)
#>      fruit poids couleur
#> fruit1 Pomme    154     vert
#> fruit2 Pomme    167     vert
```

```
tail(df, 1)
#>      fruit poids couleur
#> fruit3 Citron    92     jaune
```

On peut également utiliser l'opérateur `[,]` pour sélectionner à la fois des lignes et des colonnes, en lui passant deux arguments séparés par une virgule : d'abord la sélection des lignes puis celle des colonnes. Dans les deux cas on peut sélectionner par position, nom ou condition. Si on laisse un argument vide, on sélectionne l'intégralité des lignes ou des colonnes.

```
# Lignes 1 et 3 et colonne "poids"
df[c(1, 3), "poids"]
#> [1] 154 92
```

```
# Toutes les lignes et colonnes "poids" et "fruit"
df[, c("poids", "fruit")]
#>      poids fruit
#> fruit1 154 Pomme
#> fruit2 167 Pomme
#> fruit3  92 Citron
```

```
# Lignes pour lesquelles poids > 150, et toutes les colonnes
df[df$poids > 150, ]
#>      fruit poids couleur
#> fruit1 Pomme    154     vert
#> fruit2 Pomme    167     vert
```

```
library(stringr)
# Colonnes dont le nom contient un "o", et toutes les lignes
df[, str_detect(names(df), "o")]
#>      poids couleur
#> fruit1 154     vert
#> fruit2 167     vert
#> fruit3  92     jaune
```

Attention, le comportement de `[,]` est différent entre les *tibbles* et les *data frame* lorsqu'on ne sélectionne qu'une seule colonne. Dans le cas d'un *data frame*, le résultat est un vecteur, dans le cas d'un *tibble* le résultat est un tableau à une colonne.

```
df[, "fruit"]
#> [1] "Pomme"  "Pomme"  "Citron"
```

```
df_tib[, "fruit"]
#> # A tibble: 3 x 1
#>   fruit
#>   <chr>
#> 1 Pomme
#> 2 Pomme
#> 3 Citron
```

Cette différence peut parfois être source d'erreurs, notamment quand on développe une fonction qui prend un tableau de données en argument.

16.3.4 Modification

On peut utiliser `[][]` et `[,]` avec l'opérateur d'assignation `<-` pour modifier tout ou partie d'un tableau de données.

```
# Création d'une nouvelle colonne poids_kg
df[["poids_kg"]] <- df$poids / 1000
df
#>       fruit poids couleur poids_kg
#> fruit1 Pomme    154     vert    0.154
#> fruit2 Pomme    167     vert    0.167
#> fruit3 Citron    92      jaune   0.092
```

```
# Remplacement de la valeur de la colonne "fruit" pour les lignes
# pour lesquelles "fruit" vaut "Citron"
df[df$fruit == "Citron", "fruit"] <- "Agrume"
df
#>       fruit poids couleur poids_kg
#> fruit1 Pomme    154     vert    0.154
#> fruit2 Pomme    167     vert    0.167
#> fruit3 Agrume    92      jaune   0.092
```

Pour conclure, on peut noter que l'utilisation des opérateurs `[][]` et `[,]` sur un tableau de données peut sembler redondante et moins pratique que l'utilisation des verbes de `dplyr` comme `select()` ou `filter()`. Ils peuvent cependant être utiles lorsqu'on souhaite éviter les complications liées à l'utilisation du *tidyverse* à l'intérieur de fonctions, comme indiqué section 19. Ils peuvent également être plus rapides, et il est important de les connaître car on les rencontrera très fréquemment dans du code R sur le Web ou dans des packages.

16.4 Ressources

L'ouvrage *R for Data Science* (en anglais), accessible en ligne, contient [un chapitre sur les vecteurs atomiques et les listes](#), et [un chapitre dédié aux *tibbles*](#).

Pour aller encore plus loin, l'ouvrage *Advanced R* (également en anglais) aborde de manière approfondie les structures de données et les opérateurs de sélection [], [[]] et \$.

16.5 Exercices

16.5.1 Vecteurs atomiques

Exercice 1.1

À l'aide de seq(), créer un vecteur v contenant tous les nombres pairs entre 10 et 20.

Sélectionner les 3 premières valeurs de v.

Sélectionner toutes les valeurs de v strictement inférieures à 15.

Créer une fonction dernière() qui prend en paramètre un vecteur et retourne son dernier élément (la fonction doit pouvoir s'appliquer à n'importe quel vecteur, quelle que soit sa longueur).

```
derniere(v)
#> [1] 20
```

Créer une fonction sauf_derniere() qui prend en paramètre un vecteur et retourne ce vecteur sans son dernier élément.

```
sauf_derniere(v)
#> [1] 10 12 14 16 18
```

Exercice 1.2

Soit le vecteur vn suivant :

```
vn <- c(val1 = 10, val2 = 0, val3 = 14)
```

Sélectionner les valeurs nommées "val1" et "val3".

Créer une fonction select_noms() qui prend en argument un vecteur v et un ou plusieurs noms, et retourne uniquement les éléments de v correspondant à ces noms.

```
select_noms(vn, c("val2", "val3"))
#> val2 val3
#>    0    14
```

Facultatif : créer une fonction sauf_nom() qui prend en argument un vecteur v et un nom, et retourne tous les éléments de v sauf celui correspondant à ce nom.

```
sauf_nom(vn, "val2")
#> val1 val3
#>    10    14
```

Facultatif : comparer les résultats des deux instructions suivantes.

```
vn["val1"]
vn[["val1"]]
```

Exercice 1.3

Soit les vecteurs **x** et **y** suivants :

```
x <- c(1, NA, 3, 4, NA)
y <- c(10, 20, 30, 40, 50)
```

À l'aide de l'opérateur `[]`, sélectionner uniquement les valeurs **NA** de **x**.

De la même manière, sélectionner les valeurs de **y** correspondant aux valeurs **NA** de **x** (c'est-à-dire les valeurs 20 et 50).

En utilisant les deux instructions précédentes et l'opérateur d'assignation `<-`, remplacer les valeurs manquantes de **x** par les valeurs correspondantes de **y**.

Exercice 1.4

Créer une fonction **problemes_conversion** qui :

- prend en argument un vecteur **v**
- le convertit en vecteur numérique
- retourne les valeurs de **v** qui n'ont pas été converties correctement, c'est-à-dire celles qui ne valaient pas **NA** dans **v** mais valent **NA** après la conversion.

Vérifier avec :

```
x <- c("igloo", "20", NA, "3.5", "4,8")
problemes_conversion(x)
#> Warning in problems_conversion(x): NAs introduits lors de la conversion
#> automatique
#> [1] "igloo" "4,8"
```

16.5.2 Listes**Exercice 2.1**

Créer une liste **liste** ayant la structure suivante :

```
#> List of 3
#> $ : num 1
#> $ : chr "oui"
#> $ : int [1:3] 10 11 12
```

Donner les noms suivants aux éléments de la liste : **num**, **reponse** et **vec**.

Ajouter un élément nommé **chat** et ayant pour valeur “Ronron” à la fin de **liste**.

Modifier l'élément **chat** pour lui donner la valeur “Ronpchi”.

Supprimer l'élément **vec** de **liste**.

Exercice 2.2

Créer une fonction nommée `extremes` qui prend en argument un vecteur et retourne une liste nommée comportant sa valeur minimale et sa valeur maximale.

Appliquer cette fonction à un vecteur de votre choix et utiliser le résultat pour calculer l'étendue (soit la différence entre la valeur maximale et la valeur minimale).

Exercice 2.3

Soit la liste suivante :

```
liste <- list(1:3, runif(5), "youpi")
```

Sélectionner la sous liste composée des éléments 1 et 3 de `liste`.

Sélectionner la sous-liste composée du premier élément de `liste`.

Sélectionner le contenu du premier élément de `liste`.

En enchaînant deux opérations de sélection, sélectionner le deuxième élément du premier élément de `liste`.

Exercice 2.4

Créer une fonction `description_liste` qui prend en argument une liste et retourne :

- son premier élément
- son dernier élément
- le nombre d'éléments qu'elle contient

Vérifier avec :

```
liste <- list(1:3, runif(5), "youpi")
description_liste(liste)
#> $premier_element
#> [1] 1 2 3
#>
#> $dernier_element
#> [1] "youpi"
#>
#> $nb_elements
#> [1] 3
```

16.5.3 Tableaux de données

Exercice 3.1

Créer le tableau `df` suivant :

```
df <- tribble(
  ~fruit,    ~poids, ~couleur,
  "Pomme",   154,    "vert",
  "Pomme",   167,    "vert",
  "Citron",  92,     "jaune"
)
```

À l'aide de l'opérateur `$`, sélectionner la colonne `fruit` de `df`.

Faire de même avec l'opérateur `[[[]]]`.

À l'aide de l'opérateur `[[]]` et de la fonction `str_to_upper()` de `stringr`, transformer la colonne `fruit` en passant ses valeurs en majuscules.

Créer une fonction `colonne_maj` qui prend en argument un tableau de données `d` et un nom de colonne `colonne`, et retourne le tableau avec la colonne correspondante convertie en majuscules. Vérifier avec :

```
colonne_maj(df, "couleur")
#> # A tibble: 3 x 3
#>   fruit  poids couleur
#>   <chr> <dbl> <chr>
#> 1 Pomme    154 VERT
#> 2 Pomme    167 VERT
#> 3 Citron     92 JAUNE
```

Exercice 3.2

Créer le tableau `df` suivant :

```
df <- tribble(
  ~fruit, ~poids, ~couleur,
  "Pomme", 154, "vert",
  "Pomme", 167, "vert",
  "Citron", 92, "jaune"
)
```

À l'aide de l'opérateur `[,]`, sélectionner :

- les citrons
- les pommes et les colonnes `fruit` et `couleur`
- la première colonne des lignes ayant un poids inférieur à 100

Créer une fonction `filtre_valeur()` qui prend un seul argument nommé `valeur` et retourne les lignes de `df` pour lesquelles la colonne `fruit` vaut `valeur`. Vérifier avec :

```
filtrer_valeur("Pomme")
#> # A tibble: 2 x 3
#>   fruit  poids couleur
#>   <chr> <dbl> <chr>
#> 1 Pomme    154 vert
#> 2 Pomme    167 vert
```

Modifier la fonction pour qu'elle accepte également un argument `d` contenant le tableau à filtrer. Vérifier avec :

```
filtrer_valeur(df, "Pomme")
#> # A tibble: 2 x 3
#>   fruit  poids couleur
#>   <chr> <dbl> <chr>
#> 1 Pomme    154 vert
#> 2 Pomme    167 vert
```

Modifier à nouveau la fonction pour qu'elle accepte aussi un argument `colonne` qui contient le nom de la colonne à utiliser pour filtrer les lignes. Vérifier avec :

```
filtre_valeur(df, colonne = "couleur", valeur = "jaune")
#> # A tibble: 1 x 3
#>   fruit    poids couleur
#>   <chr>   <dbl> <chr>
#> 1 Citron     92  jaune
```

Vérifier que cette fonction marche aussi sur un autre jeu de données :

```
library(questionr)
data(hdv2003)
filtre_valeur(hdv2003, "sexe", "Femme")
```

Exercice 3.3

Reprendre le tableau `df` des exercices précédents :

```
df <- tribble(
  ~fruit,    ~poids, ~couleur,
  "Pomme",  154,    "vert",
  "Pomme",  167,    "vert",
  "Citron", 92,     "jaune"
)
```

À l'aide de l'opérateur `[,]`, effectuer les opérations suivantes :

- Créer une nouvelle colonne `id` avec les valeurs 1, 2, 3
- Remplacer la valeur “jaune” de la variable `couleur` par “jaune citron”
- Créer une nouvelle colonne `poids_rec` qui vaut “léger” si `poids` est inférieur à 100, et “lourd” sinon

Facultatif : effectuer les mêmes opérations en utilisant les verbes de `dplyr`.

Chapitre 17

Exécution conditionnelle et boucles

Nous avons vu précédemment comment écrire nos propres fonctions. Cette section présente des éléments du langage qui permettent de programmer des actions un peu plus complexes : exécuter du code de manière conditionnelle selon le résultat d'un test, et répéter des opérations avec des boucles.

Les notions décrites dans cette partie s'appliquent pour le développement de fonctions, mais peuvent aussi être mises en œuvre à tout moment dans un script.

On commence par charger les jeux de données d'exemple utilisés par la suite :

```
library(questionr)
data(hdv2003)
data(rp2018)
```

17.1 if et else : exécuter du code sous certaines conditions

17.1.1 if

L'instruction `if` permet de n'exécuter du code que si une condition est remplie.

`if` est suivie d'une condition (entre parenthèses) puis d'un bloc de code (entre accolades). Ce bloc de code n'est exécuté que si la condition est vraie.



Par exemple, dans le code suivant, le message `Bonjour !` ne sera affiché que si la valeur de l'objet `prenom` vaut "`Pierre-Edmond`" :

```

prenom <- "Pierre-Edmond"
if (prenom == "Pierre-Edmond") {
  message("Bonjour !")
}
#> Bonjour !

```

On peut utiliser ce code pour créer une passionnante fonction qui a pour objectif de ne dire bonjour qu'aux personnes qui s'appellent Pierre-Edmond :

```

bonjour_pierre_edmond <- function(prenom) {
  if (prenom == "Pierre-Edmond") {
    message("Bonjour !")
  }
}

bonjour_pierre_edmond("Pierre-Edmond")
#> Bonjour !

bonjour_pierre_edmond("Valérie-Charlotte")

```

Une autre utilisation possible (et un peu plus utile) dans le cadre d'une fonction est de n'exécuter certaines instructions que si la valeur d'un argument vaut une valeur donnée. Dans l'exemple suivant, on n'applique la fonction `round()` que si l'argument `arrondir` vaut `TRUE`.

```

moyenne <- function(x, arrondir = TRUE) {
  res <- mean(x)
  if (arrondir == TRUE) {
    res <- round(res)
  }
  res
}

v <- c(1.4, 2.3, 8.9)
moyenne(v)
#> [1] 4
moyenne(v, arrondir = FALSE)
#> [1] 4.2

```

 On notera que le test `x == TRUE` est en fait redondant, car son résultat est le même que la valeur de `x` :

- si `x` vaut `TRUE`, `x == TRUE` vaut `TRUE`
- si `x` vaut `FALSE`, `x == TRUE` vaut `FALSE`

On remplacera donc en général `if (x == TRUE)` par `if (x)`.

De la même manière, on pourra remplacer `if (x == FALSE)` par `if (!x)`.

Dans notre fonction `moyenne` ci-dessus, on peut donc remplacer :

```

if (arrondir == TRUE) {
  res <- round(res)
}

```

Par :

```
if (arrondir) {
  res <- round(res)
}
```

À noter également que quand le bloc de code qui suit une instruction **if** ne comporte qu'une seule instruction, on peut omettre les accolades qui l'entourent. Les syntaxe suivantes sont donc équivalentes :

```
if (arrondir) {
  res <- round(res)
}

if (arrondir) res <- round(res)
```

17.1.2 **if / else**

On utilise souvent **if** en le faisant suivre par une instruction **else**. **else** précède un autre bloc de code R qui ne s'exécute que si la condition donnée au **if** est fausse :

Condition

```
if (x > 0) {
  message("Positif !")
} else {
  message("Négatif !")
```

} Code si la condition est vraie

} Code si la condition est fausse

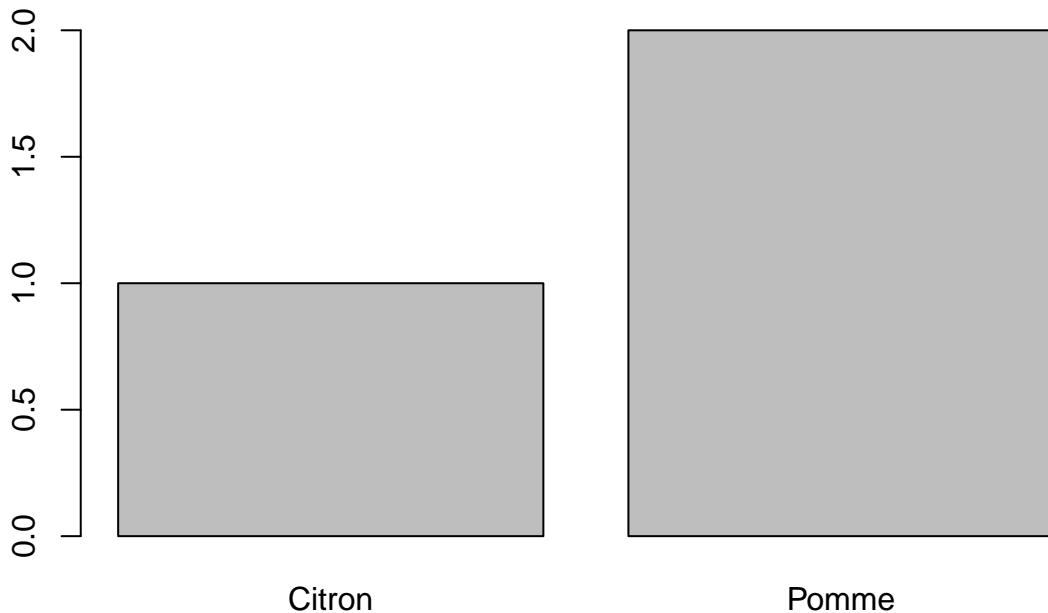
On peut ainsi utiliser **if / else** pour une nouvelle fonction fort utile qui nous évitera bien des désagréments météorologiques.

```
conseil_vestimentaire <- function(temperature) {
  if (temperature > 15) {
    message("La polaire n'est pas forcément nécessaire.")
  } else {
    message("Vous devriez prendre une petite laine.")
  }
}

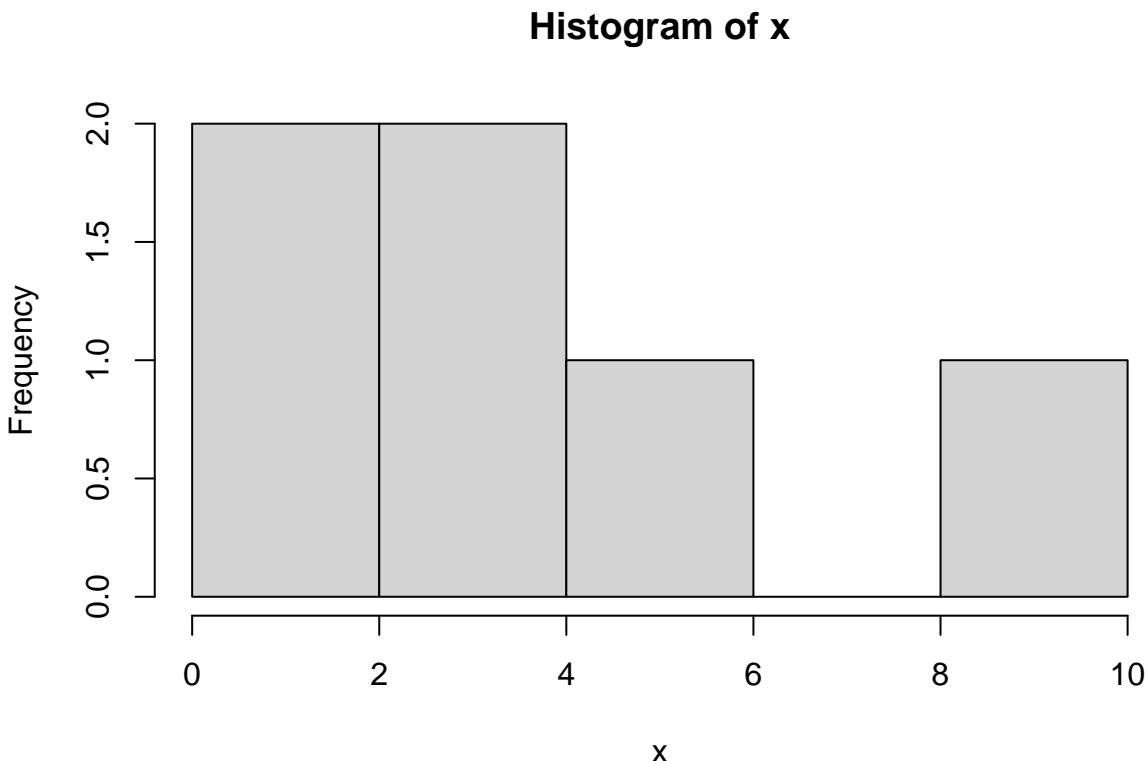
conseil_vestimentaire(-5)
#> Vous devriez prendre une petite laine.
```

Plus utile, on peut l'utiliser pour effectuer deux actions différentes en fonction de la valeur d'un argument. La fonction suivante génère deux graphiques différents selon le type du vecteur passé en argument :

```
graph_var <- function(x) {  
  if (is.character(x)) {  
    barplot(table(x))  
  } else {  
    hist(x)  
  }  
}  
  
graph_var(c("Pomme", "Pomme", "Citron"))
```



```
graph_var(c(1, 5, 10, 3, 1, 4))
```



17.1.3 “if” / “else if” / “else”

Une possibilité complémentaire est d’ajouter des blocs `else if` qui permettent d’ajouter des conditions supplémentaires. Dès qu’une condition est vraie, le bloc de code correspondant est exécuté. Le dernier bloc `else` est exécuté si aucune des conditions n’est vraie.

On peut donc améliorer encore notre fonction `graph_var()` pour tester plusieurs types explicitement et afficher un message si aucun type géré n’a été reconnu.

```
graph_var <- function(x) {
  if (is.character(x)) {
    barplot(table(x))
  } else if (is.numeric(x)) {
    hist(x)
  } else {
    message("Le type de x n'est pas géré par la fonction")
  }
}

graph_var(c(TRUE, FALSE, TRUE))
#> Le type de x n'est pas géré par la fonction
```

Attention, seul le bloc de la première condition vraie est exécuté, l’ordre des conditions est donc important. Dans l’exemple suivant, le second bloc n’est jamais exécuté et donc le second message jamais affiché.

```
test_x <- function(x) {
  if (x < 100) {
    message("x est inférieur à 100")
```

```

    } else if (x < 10) {
        message("x est inférieur à 10")
    }
}

test_x(5)
#> x est inférieur à 100

```

Il est donc important d'ordonner les conditions de la plus spécifique à la plus générale.

17.1.4 Construction de conditions complexes

On peut combiner plusieurs tests avec les opérateurs logiques classiques :

- `&&` est l'opérateur “et”, qui est vrai si les deux conditions qu'il réunit sont vraies
- `||` est l'opérateur “ou”, qui est vrai si au moins l'une des deux conditions qu'il réunit sont vraies
- `!` est l'opérateur “not”, qui teste si la condition qu'il précède est fausse

Ainsi, si on veut qu'une variable `temperature` soit comprise entre 15 et 25, on écrira :

```

verifie_temperature <- function(temperature) {
  if (temperature >= 15 && temperature <= 25) {
    message("Température ok")
  }
}
verifie_temperature(20)

```

Si on souhaite tester que `temperature` est inférieure à 15 ou supérieure à 25 :

```

verifie_temperature <- function(temperature) {
  if (temperature < 15 || temperature > 25) {
    message("Température pas glop")
  }
}
verifie_temperature(10)

```

Si on veut tester si `temperature` vaut `NULL`, on peut utiliser `is.null()`.

```

verifie_temperature <- function(temperature = NULL) {
  if (is.null(temperature)) {
    message("Merci d'indiquer une température")
  }
}
verifie_temperature()

```

Mais si à l'inverse on veut tester si `temperature` *n'est pas* `NULL`, on inverse le test précédent en utilisant `!`.

```
verifie_temperature <- function(temperature = NULL) {
  if (!is.null(temperature)) {
    message("Merci d'avoir indiqué une température")
  }
}
verifie_temperature(15)
```

On pourra noter qu'il existe deux types d'opérateurs "et" et "ou" dans R :

- Les opérateurs simples `&` et `|` sont des opérateurs *vectorisés*. Ils peuvent s'appliquer à des vecteurs et retourneront un vecteur de `TRUE` et `FALSE`.
- Les opérateurs doubles `&&` et `||` ne peuvent retourner qu'une seule valeur, et si on leur fournit des vecteurs ils n'utiliseront que la première valeur de chacun d'entre eux.

```
x <- 1:5
x > 0 & x <= 2
#> [1] TRUE TRUE FALSE FALSE FALSE
x > 0 && x <= 2
#> [1] TRUE
```

Quand on passe un test à un `if`, celui-ci est censé retourner une unique valeur `TRUE` ou `FALSE`. Une erreur fréquente, notamment quand on est dans une fonction, est de passer à `if` une condition appliquée à un vecteur. Dans ce cas R a la bonne idée d'afficher un avertissement, et il n'utilise alors que la première valeur du vecteur pour déterminer si le bloc de code doit être exécuté ou non.

```
superieur_a_5 <- function(x) {
  if (x >= 5) {
    message(">=5")
  }
}

superieur_a_5(1:10)
#> Warning in if (x >= 5) { : la condition a une longueur > 1 et seul le premier
#> élément est utilisé
```



À retenir : quand on utilise l'instruction `if`, la condition qui lui est passée entre parenthèses ne doit renvoyer qu'une seule valeur `TRUE` ou `FALSE`. Si on utilise une condition complexe, on utilisera donc plutôt les opérateurs doubles `&&` et `||`.

17.1.5 Différence entre `if / else` et `ifelse`

Une source fréquente de confusion concerne la différence entre les instructions `if / else` et la fonction `ifelse()` de R base (ou son équivalent `if_else()` de `dplyr`, voir section 9.4.1). Les deux sont pourtant très différentes :

- `if / else` s'utilisent quand on teste une seule condition et qu'on veut exécuter des blocs de code différents selon son résultat
- `ifelse` applique un test à tous les éléments d'un vecteur et retournent un vecteur dont les éléments dépendent du résultat de chaque test

Premier cas de figure : un objet `x` contient une seule valeur et on veut afficher un message différent selon si celle-ci est inférieure ou supérieure à 10. Dans ce cas on utilise `if / else`.

```
x <- 5

if (x >= 10) {
  message(">=10")
} else {
  message("<10")
}
#> <10
```

Deuxième cas de figure : `x` est un vecteur et on souhaite recoder chacune de ses valeurs selon le même critère que ci-dessus. Dans ce cas on utilise `ifelse`.

```
x <- 5:15
x_rec <- ifelse(x >= 10, ">=10", "<10")
x_rec
#> [1] "<10"  "<10"  "<10"  "<10"  "<10"  ">=10"  ">=10"  ">=10"  ">=10"
#> [1] ">=10"
```

17.2 Contrôle de l'exécution et gestion des erreurs

L'instruction `if` est souvent utilisée dans des fonctions pour valider les valeurs passées en arguments, ou plus généralement pour contrôler que l'exécution du code se déroule comme prévu.

17.2.1 Utilisation de `return` pour sortir de la fonction

On peut utiliser un `return` pour interrompre l'exécution de la fonction et retourner un résultat. On a en effet vu section 14.2.5 que dès que R rencontre un `return` dans une fonction, il interrompt immédiatement l'exécution de celle-ci.

La fonction suivante retourne la longueur du mot le plus long dans un vecteur de chaînes de caractères.

```
longueur_max <- function(x) {
  max(nchar(x))
}

longueur_max(c("Pomme", "Pamplemousse"))
#> [1] 12
```

Cette fonction n'a pas trop de sens si on lui passe en entrée un vecteur qui n'est pas un vecteur de chaînes de caractères. On peut donc rajouter un test qui, si `x` n'est pas de type `character`, retourne directement la valeur `NA`.

```
longueur_max <- function(x) {
  if (!is.character(x)) {
    return(NA)
  }
  max(nchar(x))
}
```

```
longueur_max(1:5)
#> [1] NA
```

17.2.2 warning

La fonction `warning` fonctionne comme `message` mais permet d'afficher un avertissement. Celui-ci est présenté un peu différemment dans la console de manière à attirer l'attention, et il indique quelle fonction a déclenché l'avertissement, ce qui peut être utile pour retrouver l'origine du problème.

Dans la fonction précédente, on peut ajouter un avertissement dans le cas où le vecteur passé en argument n'est pas de type `character`.

```
longueur_max <- function(x) {
  if (!is.character(x)) {
    warning("x n'est pas de type character, le résultat vaut NA.")
    return(NA)
  }
  max(nchar(x))
}

longueur_max(1:5)
#> Warning in longueur_max(1:5): x n'est pas de type character, le résultat vaut
#> NA.
#> [1] NA
```

17.2.3 stop et stopifnot

`stop` fonctionne comme `warning` mais déclenche une erreur qui interrompt totalement l'exécution du code. Quand R le rencontre dans une fonction, il sort immédiatement de la fonction, ne retourne aucun résultat, et il interrompt également toutes les autres instructions en attente d'exécution.

On peut ainsi considérer, toujours dans la fonction `longueur_max`, que le fait de ne pas fournir en argument un vecteur de type `character` est suffisamment “grave” pour interrompre l'exécution en cours et forcer la personne qui utilise la fonction à régler le problème.

```
longueur_max <- function(x) {
  if (!is.character(x)) {
    stop("x doit être de type character.")
  }
  max(nchar(x))
}

longueur_max(1:5)
#> Error in longueur_max(1:5): x doit être de type character.
```



Savoir si un problème doit être traité comme un avertissement ou comme une erreur relève du choix de la personne qui développe la fonction : chaque cas est particulier.

`stopifnot` est une syntaxe alternative plus compacte qui combine test et message d'erreur. On lui passe en premier argument une condition, et en deuxième argument un message à afficher si la condition est fausse.

On peut donc réécrire notre fonction `longueur_max` ci-dessus de la manière suivante :

```

longueur_max <- function(x) {
  stopifnot(is.character(x))
  max(nchar(x))
}

longueur_max(1:5)
#> Error in longueur_max(1:5): is.character(x) n'est pas TRUE

```

Si on souhaite un message d'erreur personnalisé il faut le passer comme nom de la condition.

```

longueur_max <- function(x) {
  stopifnot(
    "x doit être de type character" = is.character(x)
  )
  max(nchar(x))
}

longueur_max(1:5)
#> Error in longueur_max(1:5): x doit être de type character

```

17.2.4 Tester la présence d'un argument facultatif

On a vu section 14.2.3 que pour rendre un argument de fonction “facultatif”, on doit lui attribuer une valeur par défaut. Parfois cependant, on n'a pas de valeur par défaut évidente à lui attribuer directement : dans ce cas on lui attribue la valeur NULL et on utilise un `if()` dans la fonction pour déterminer s'il a été défini ou non par l'utilisateur.

Par exemple, soit une fonction qui génère un graphique avec un argument `titre` qui permet de définir son titre.

```

histo <- function(x, titre) {
  hist(x, main = titre)
}

```

Si l'utilisateur ne donne pas de titre, on souhaite ajouter un titre qui indique la valeur de la moyenne de la variable représentée. Dans ce cas on attribue à `titre` la valeur par défaut NULL, et on vérifie dans le corps de la fonction que l'utilisateur n'a pas fourni de valeur avec `if (is.null(titre))`. On peut alors calculer la valeur “par défaut” souhaitée :

```

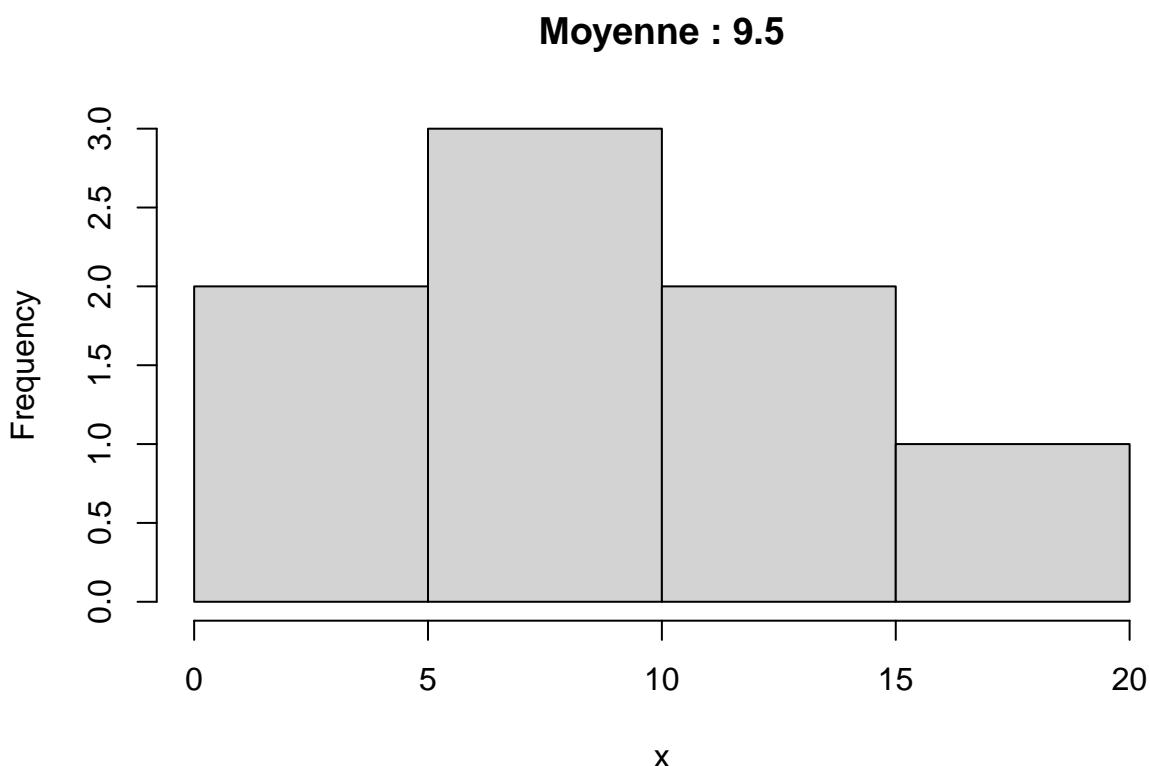
histo <- function(x, titre = NULL) {
  if (is.null(titre)) {
    titre <- paste("Moyenne :", mean(x))
  }
  hist(x, main = titre)
}

```

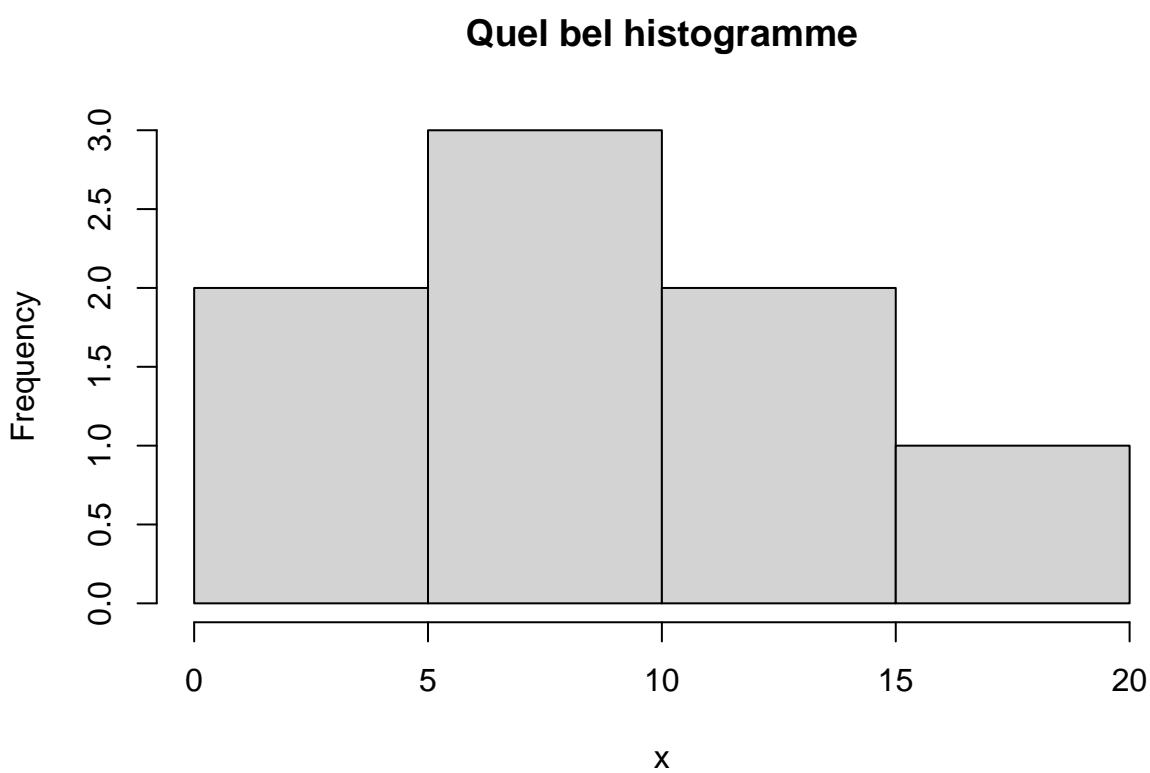
```

x <- c(1, 15, 8, 10, 12, 18, 8, 4)
histo(x)

```



```
histo(x, titre = "Quel bel histogramme")
```

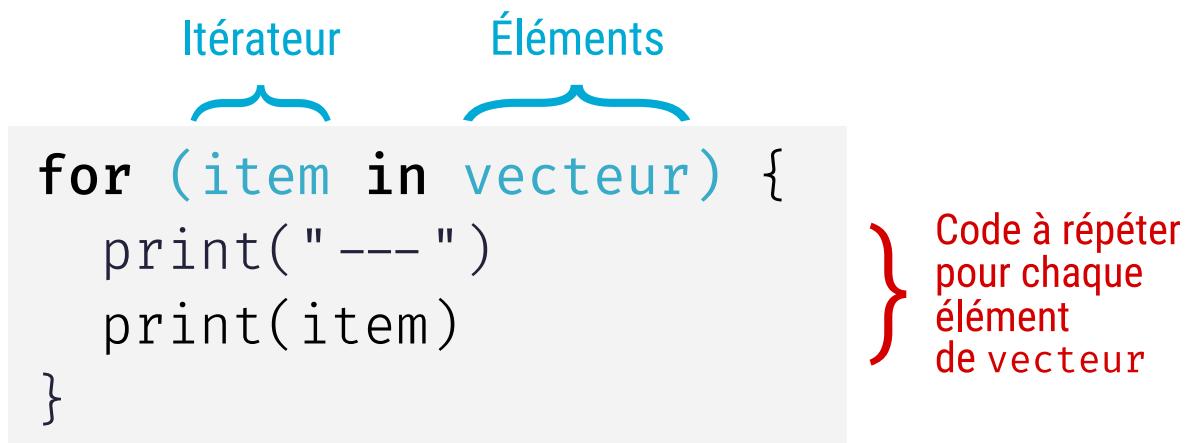


17.3 `for` et `while` : répéter des instructions dans une boucle

Les boucles permettent de répéter du code plusieurs fois, soit en fonction d'une condition soit selon les éléments d'un vecteur¹.

17.3.1 `for`

Le premier type de boucle est défini par l'instruction `for`. Sa structure est la suivante :



Le principe est le suivant : on fournit à `for` entre parenthèses une expression du type `item in vecteur`, puis un bloc de code entre accolades. `for` va exécuter le bloc de codes pour chacune des valeurs de `vecteur`, et affectera tour à tour à `item` la valeur courante de `vecteur`.

Prenons tout de suite un exemple pour mieux comprendre.

```

for (item in 1:5) {
  print(item)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
  
```

Ici notre vecteur “source” est constitué des entiers de 1 à 5. `for` va donc exécuter l'instruction `print(item)` 5 fois, en remplaçant la première fois `item` par 1, la seconde fois par 2, etc.

On peut itérer sur différents types d'objets, et le nom `item` peut être remplacé par ce que l'on souhaite :

```

for (prenom in c("Pierre-Edmond", "Valérie-Charlotte")) {
  message("Bonjour ", prenom, " !")
}
#> Bonjour Pierre-Edmond !
#> Bonjour Valérie-Charlotte !
  
```

Exemple un peu plus complexe, la fonction suivante prend en entrée un tableau de données et un vecteur de noms de variables, et affiche le résultat de `summary` pour chacune de ces variables.

¹En complément, on verra également dans la section 18 d'autres fonctions tirées de l'extension `purrr` qui permettent d'appliquer une fonction en itérant sur les éléments de plusieurs objets.

```

summaries <- function(d, vars) {
  for (var in vars) {
    message(" --- ", var, " ---")
    print(summary(d[, var]))
  }
}

summaries(hdv2003, c("sexe", "age", "heures.tv"))
#> --- sexe ---
#> Homme Femme
#>   899 1101
#> --- age ---
#>   Min. 1st Qu. Median Mean 3rd Qu. Max.
#>   18.00 35.00 48.00 48.16 60.00 97.00
#> --- heures.tv ---
#>   Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#>   0.000 1.000 2.000 2.247 3.000 12.000      5

```

Parfois on souhaite itérer sur les éléments par leur position plutôt que par leur valeur. Dans l'exemple suivant, la fonction `affiche_dimensions` affiche le nombre de lignes et de colonnes des tableaux de données contenus dans une liste :

```

affiche_dimensions <- function(dfs) {
  for (df in dfs) {
    message("Dimensions : ", nrow(df), "x", ncol(df))
  }
}

l <- list(
  hdv = hdv2003,
  rp = rp2018
)

affiche_dimensions(l)
#> Dimensions : 2000x20
#> Dimensions : 5417x62

```

Si on souhaite afficher le nom du tableau en plus de ses dimensions afin de rendre les résultats plus lisibles, on doit itérer sur la position des éléments pour pouvoir récupérer à la fois leur nom (dans `names(dfs)`) et leur valeur (dans `dfs`). Dans ces cas-là on peut utiliser la fonction `seq_along()` qui génère une liste d'entiers correspondant au nombre d'éléments de l'objet qu'on lui passe en argument.

```

x <- c("rouge", "vert", "bleu")
seq_along(x)
#> [1] 1 2 3

```

On peut du coup réécrire `affiche_dimensions` de la façon suivante.

```

affiche_dimensions <- function(dfs) {
  for (i in seq_along(dfs)) {
    name <- names(dfs)[[i]]
    df <- dfs[[i]]
    message("Dimensions de ", name, " : ", nrow(df), "x", ncol(df))
  }
}

```

```

    }
}

affiche_dimensions(1)
#> Dimensions de hdu : 2000x20
#> Dimensions de rp : 5417x62

```



Il est assez naturel d'utiliser `for (i in 1:length(x))` plutôt que `for (i in seq_along(x))`. L'utilisation de `seq_along(x)` est cependant préférable, notamment lorsqu'on est dans une fonction, car elle n'essaie pas d'exécuter le bloc de code si jamais `x` est de longueur nulle.

En effet, si `length(x)` vaut 0 alors `1:length(x)` vaut `1:0`, c'est-à-dire le vecteur `c(1, 0)`, ce qui signifie que la boucle sera exécutée et risque de générer une erreur. De son côté, `seq_along(x)` garantit dans ces cas-là qu'aucune itération du `for` n'est exécutée.

À noter que quand on sort d'une boucle `for`, l'objet utilisé pour itérer sur les valeurs du vecteur existe toujours, et contient la dernière valeur qu'il a prise.

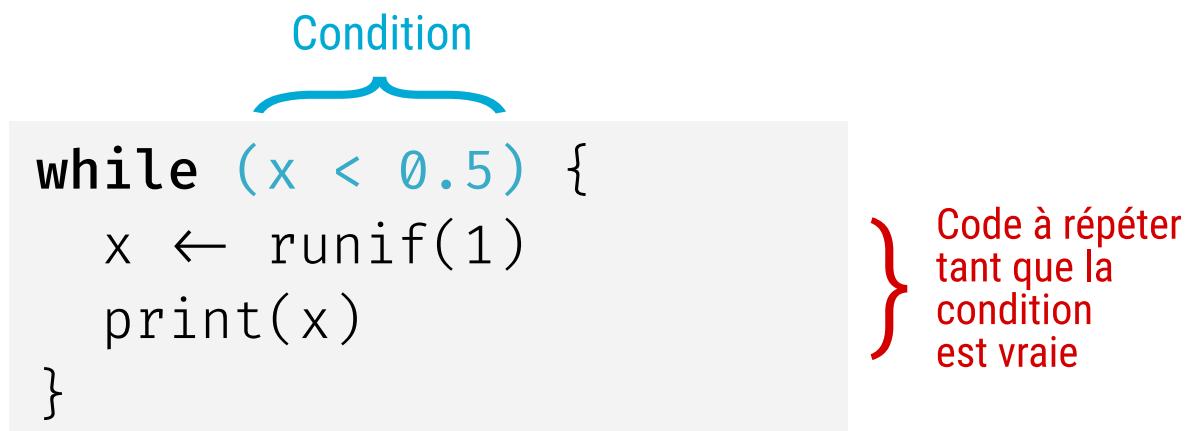
```

for (i in 1:3) {
  print("a")
}
#> [1] "a"
#> [1] "a"
#> [1] "a"
print(i)
#> [1] 3

```

17.3.2 while

`while` prend en argument une condition et est suivi d'un bloc de code entre accolades. Elle exécute le bloc tant que la condition est vraie :



Par exemple, la fonction suivante simule un tirage à pile ou face à l'aide de la fonction `sample()`. La simulation de tirage s'exécute et affiche le résultat tant qu'on obtient “Pile” (et interrompt la boucle au premier “Face”) :

```

resultat <- ""
while (resultat != "Face") {
  resultat <- sample(c("Pile", "Face"), size = 1)
  print(resultat)
}

```

```

[1] "Pile"
[1] "Pile"
[1] "Face"

```

Le déroulement de la boucle est le suivant :

1. la première instruction initialise la valeur de la variable `resultat` avec une chaîne vide
2. à la première entrée dans le `while`, `resultat` vaut "", elle est donc différente de "Face" et le bloc de code est donc exécuté
3. à la fin de cette exécution, `resultat` vaut soit "Pile" soit "Face". On entre alors une deuxième fois dans le `while`. Si `resultat` vaut "Pile", la condition du `while` n'est pas vraie, on n'exécute donc pas le bloc de code et on sort de la boucle. Si `resultat` vaut "Face", la condition est vraie, on exécute le bloc de code et on rentre ensuite une troisième fois dans le `while`, etc.

17.3.3 `next` et `break`

Les instructions `next` et `break` permettent de modifier les itérations d'une boucle `for` ou `while`.

`next` permet de sortir de l'itération courante et de passer directement à l'itération suivante sans exécuter le reste du code.

Reprendons la fonction `summaries`, vue plus haut, qui affiche le résumé de plusieurs variables d'un tableau de données.

```

summaries <- function(d, vars) {
  for (var in vars) {
    message(" --- ", var, " ---")
    print(summary(d[, var]))
  }
}

summaries(hdv2003, c("sexe", "age"))
#> --- sexe ---
#> Homme Femme
#>   899 1101
#> --- age ---
#>      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#>     18.00  35.00  48.00  48.16  60.00  97.00

```

Si on passe à la fonction un nom de colonne qui n'existe pas, on obtient une erreur et les autres variables ne sont pas affichées.

```

summaries(hdv2003, c("sexe", "igloo", "age"))
#> --- sexe ---
#> Homme Femme
#>   899 1101
#> --- igloo ---
#> Error in `data.frame`(d, , var): colonnes non définies sélectionnées

```

On pourrait dans ce cas vouloir afficher les résultats pour les “bonnes” colonnes, et ignorer les autres. C’est possible si on ajoute une instruction `next` quand la valeur courante de `var` ne fait pas partie des noms de colonnes (on pourrait aussi ajouter un `warning()` juste avant le `next` pour informer l’utilisateur).

```
summaries <- function(d, vars) {
  for (var in vars) {
    if (!(var %in% names(d))) {
      next
    }
    message(" --- ", var, " ---")
    print(summary(d[, var]))
  }
}

summaries(hdv2003, c("sexe", "igloo", "age"))
#> --- sexe ---
#> Homme Femme
#>   899 1101
#> --- age ---
#>   Min. 1st Qu. Median     Mean 3rd Qu.    Max.
#>   18.00   35.00   48.00   48.16   60.00   97.00
```

L’instruction `break` est un peu plus radicale : non seulement elle sort de l’itération courante sans exécuter la suite du code, mais elle interrompt carrément la boucle toute entière et n’exécute pas les itérations restantes.

Dans l’exemple précédent, si on remplace `next` par `break`, on voit bien qu’on sort de la boucle et que seule la première itération est totalement exécutée.

```
summaries <- function(d, vars) {
  for (var in vars) {
    if (!(var %in% names(d))) {
      break
    }
    message(" --- ", var, " ---")
    print(summary(d[, var]))
  }
}

summaries(hdv2003, c("sexe", "igloo", "age"))
#> --- sexe ---
#> Homme Femme
#>   899 1101
```

17.3.4 Quand (ne pas) utiliser des boucles

Le mécanisme des boucles, assez intuitif, peut être utilisé pour beaucoup d’opérations. Il y a cependant sous R des alternatives souvent plus rapides, qu’il est préférable de privilégier.

Avant tout, de nombreuses fonctions R sont “vectorisées” et s’appliquent directement à tous les éléments d’un vecteur. Dans le cas où une fonction vectorisée existe déjà, elle propose en général une syntaxe plus compacte et une exécution (beaucoup) plus rapide.

Pour prendre un exemple caricatural, si on souhaite ajouter 10 à chaque élément d’un vecteur on évitera absolument de faire :

```
for (i in seq_along(x)) {
  x[i] <- x[i] + 10
}
```

Et on se contentera d'un beaucoup plus simple `x + 10`.

Autre exemple, si on souhaite remplacer dans tous les éléments d'un vecteur de chaînes de caractères le caractère "X" par le caractère "o", on pourrait être tenter de faire une boucle du type :

```
motz <- c("brXuette", "mXtX", "iglXX")
for (i in seq_along(mots)) {
  mots[i] <- str_replace_all(mots[i], "X", "o")
}
```

Or c'est tout à fait inutile car `str_replace_all()` étant vectorisée, on peut l'appliquer directement à un vecteur sans utiliser de boucle.

```
motz <- str_replace_all(mots, "X", "o")
```

Dernier exemple, la boucle suivante remplace les valeurs manquantes d'un vecteur par les valeurs correspondantes d'un deuxième vecteur.

```
x <- c(1, NA, 4, 110, NA)
y <- c(20, 30, 40, 50, 60)

for (i in seq_along(x)) {
  if (is.na(x[i])) {
    x[i] <- y[i]
  }
}
```

Cette boucle sera avantageusement remplacée par une utilisation plus compacte et plus rapide de l'opérateur `[]`.

```
x[is.na(x)] <- y[is.na(x)]
```

En dehors des questions de performance, une boucle peut aussi être moins lisible que certaines alternatives. Soit la fonction suivante, qui prend en entrée un vecteur de mots et retourne le nombre total de voyelles qu'il contient.

```
library(stringr)
n_voyelles <- function(mots) {
  nb <- str_count(mots, "[aeiou]")
  sum(nb)
}

n_voyelles(c("le", "chat", "roupille"))
#> [1] 6
```

Supposons qu'on souhaite appliquer cette fonction à une série de vecteurs de mots contenus dans une liste. On pourrait utiliser une boucle `for` parcourant cette liste, appliquant la fonction, et ajoutant le résultat à un vecteur numérique vide préalablement créé avec `numeric()`.

```
phrases <- list(
  c("le", "chat", "roupille"),
  c("l'autre", "chat", "roupille", "aussi")
)

res <- numeric()
for (i in seq_along(phrases)) {
  res[i] <- n_voyelles(phrases[[i]])
}
res
#> [1] 6 11
```

On verra cependant dans la section 18 que des fonctions permettent de faire ce genre de choses de manière beaucoup plus simple et plus lisible. Ici par exemple on obtiendrait le même résultat avec un simple :

```
phrases %>% map_int(n_voyelles)
#> [1] 6 11
```

Au final, entre les fonctions vectorisées existantes et les possibilités fournies par `purrr`, il est assez rare de devoir utiliser une boucle directement dans R. Pour autant, il ne faut pas non plus tomber dans l'excès inverse et considérer que tout usage de `for` ou `while` doit être évité : ces fonctions sont parfaitement justifiées dans certains cas de figure, et si vous trouvez une solution qui fonctionne de manière efficace avec une boucle `for`, il n'est pas forcément utile de chercher à la remplacer.

17.4 Ressources

L'ouvrage *R for Data Science* (en anglais), accessible en ligne, contient [un chapitre sur les boucles for](#), et [un chapitre sur les blocs if / else](#).

L'ouvrage *Advanced R* (également en anglais) aborde de manière approfondie [les tests et les boucles](#).

Sur le blog de [ThinkR](#), [un article détaillé sur l'utilisation des boucles](#) et les alternatives possibles.

Sur le blog de [Florian Privé](#), un [billet approfondi sur les raisons pour lesquelles les boucles peuvent être lentes](#) et sur les cas où il est préférable de ne pas les utiliser.

17.5 Exercices

17.5.1 if et else

Exercice 1.1

Écrire une fonction `gel` qui prend un argument nommé `temperature` et effectue les actions suivantes :

- si `temperature` est négative, affiche le message “ça gèle” avec la fonction `message()`
- sinon, affiche le message “ça gèle pas” avec la fonction `message()`

Exercice 1.2

Écrire une fonction `meteo` qui prend un argument nommé `temperature` et effectue les actions suivantes :

- si `temperature` est inférieure à 0, affiche le message “ça caille” avec la fonction `message()`
- si `temperature` est comprise entre 0 et 15, affiche le message “fais pas chaud”
- si `temperature` est comprise entre 15 et 30, affiche le message “on est pas mal”
- si `temperature` est supérieure à 30, affiche le message “tous à Miribel”

Exercice 1.3

Écrire une fonction `avertissement` qui prend deux arguments `pluie` et `parapluie` et qui effectue les opérations suivantes :

- si `pluie` vaut TRUE et `parapluie` vaut FALSE, affiche “mouillé” avec la fonction `message()`
- si `pluie` vaut TRUE et `parapluie` vaut TRUE, affiche “bien vu”
- si `pluie` vaut FALSE, affiche “RAS”

17.5.2 Contrôle de l'exécution**Exercice 2.1**

Créer une fonction `moyenne_arrondie`, qui prend en argument un vecteur `x` et un argument facultatif `decimales`. La fonction doit effectuer les opérations suivantes :

- calculer la moyenne du vecteur
- si `decimales` est défini, arrondir la moyenne au nombre de décimales correspondant avec la fonction `round()`
- retourner le résultat

Exercice 2.2

Créer une fonction `etendue` qui retourne la différence entre la plus grande et la plus petite valeur d'un vecteur.

Modifier la fonction pour qu'elle retourne `NA` si le vecteur passé en argument n'est pas numérique.

Modifier à nouveau la fonction pour qu'elle affiche un avertissement avant de renvoyer la valeur `NA`.

Exercice 2.3

Créer une fonction `proportion` qui prend en argument un vecteur et retourne les valeurs de ce vecteur divisée par leur somme.

Essayer d'exécuter `proportion(c(-2, 1, 1))`. Pourquoi obtient-on ce résultat ?

Modifier la fonction pour qu'elle retourne un message d'erreur si la somme des éléments du vecteur vaut 0.

17.5.3 Boucles**Exercice 3.1**

Charger le jeu de données `hdv2003` de l'extension `questionr` avec :

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

À l'aide d'une boucle `for`, parcourir les noms des variables de `hdv2003`. Si la variable en question est numérique, faire l'histogramme de la variable avec la fonction `hist()`.

Ajouter le nom de la variable comme titre du graphique en utilisant l'argument `main` de `hist()`.

Exercice 3.2

La fonction `readline()` permet de lire une chaîne de caractère saisie au clavier de la manière suivante :

```
réponse <- readline("Quelle est votre réponse ?")
```

Écrire le code qui effectue les opérations suivantes :

- Afficher le message “Quel est le plus grand sociologue de tous les temps ?” et demander la réponse à l’utilisateur
- Si la réponse saisie est “Tonton Michel”, afficher “Bingo !”
- Sinon, afficher “Nope”

À l’aide d’une boucle `while()`, modifier le code précédent pour que la question soit répétée jusqu’à ce que l’utilisateur saisisse “Tonton Michel”.

Exercice 3.3

À l’aide d’une boucle `for`, écrire une fonction `somme_positifs` qui prend en argument un vecteur et retourne la somme de tous les nombres positifs qu’il contient.

Réécrire cette fonction pour qu’elle retourne le même résultat mais sans utiliser de boucle.

Exercice 3.4

En utilisant une boucle `for`, créer une fonction `somme_premiers_positifs` qui prend en argument un vecteur et retourne la somme de tous les nombres positifs qu’il contient en partant du début du vecteur et en s’arrêtant au premier élément négatif (on pourra recopier et modifier la première fonction `somme_positifs` de l’exercice précédent).

Facultatif : réécrire la fonction pour qu’elle retourne le même résultat sans utiliser de boucle `for`.

Exercice 3.5

Soit la fonction `pile_ou_face` suivante, qui simule un jet de pièce :

```
pile_ou_face <- function() {
  alea <- runif(1)
  if (alea < 0.5) {
    result <- "pile"
  } else {
    result <- "face"
  }
  result
}
```

Modifier cette fonction en utilisant une boucle `for` pour qu’elle accepte un argument `n` et retourne un vecteur comprenant le résultat de `n` tirages.

```
pile_ou_face(4)
#> [1] "face" "pile" "face" "face"
```

Réécrire la fonction pour qu’elle retourne le même résultat sans utiliser de boucle `for`.

Chapitre 18

Itérer avec purrr

purrr est une extension du *tidyverse* qui fournit des outils pour travailler avec les vecteurs et les fonctions, et notamment pour itérer sur les éléments de vecteurs ou de listes en leur appliquant une fonction.

Dans cette section on aura besoin des extensions du *tidyverse* (dont purrr fait partie), que nous chargeons donc immédiatement, de même que les jeux de données rp2018 et hdv2003 de questionr.

```
library(tidyverse)
library(questionr)
data(hdv2003)
data(rp2018)
```

18.1 Exemple d'application

Pour mieux appréhender de quoi il s'agit, on part du vecteur suivant, qui contient des extraits (fictifs ?) de discours politiques.

```
discours <- c(
  "nous privilégierons une intergouvernementalisation sans agir anticonstitutionnellement",
  "le souffle de la nation est le vent qui agite les drapeaux de nos libertés",
  "nous devons faire preuve de plus de pédagogie pour cette réforme",
  "mon compte twitter a été piraté"
)
```

On souhaite calculer la longueur de chaque extrait, en nombre de mots.

On commence par découper grossièrement chaque extrait en mots en utilisant la fonction `str_split()` de `stringr`¹.

```
mot <- str_split(discours, " ")
str(mot)
#> List of 4
#> $ : chr [1:7] "nous" "privilégierons" "une" "intergouvernementalisation" ...
#> $ : chr [1:15] "le" "souffle" "de" "la" ...
#> $ : chr [1:11] "nous" "devons" "faire" "preuve" ...
#> $ : chr [1:6] "mon" "compte" "twitter" "a" ...
```

¹Pour une application sérieuse on utilisera des fonctions spécifiques de “tokenization” d’extensions dédiées à l’analyse textuelle, comme `quanteda`.

L'objet `mot`s est une liste de vecteurs de chaînes de caractères qui contiennent les mots des différents extraits. Calculer le nombre de mots de chaque extrait revient à calculer la longueur de chaque élément de `mot`s. Pour cela on pourrait vouloir utiliser la fonction `length()` directement :

```
length(mots)
#> [1] 4
```

Ceci ne fonctionne pas, car `length()` nous retourne le nombre d'éléments de `mot`s, pas celui de chacun de ses éléments : ce qu'on veut, ça n'est pas `length(mots)` mais `length(mots[[1]])`, `length(mots[[2]])`, etc.

On a vu section 17.3.4 qu'on peut pour cela utiliser une boucle `for`, par exemple de la manière suivante.

```
resultat <- list()
for (item in mots) {
  resultat <- c(resultat, length(item))
}
resultat
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 15
#>
#> [[3]]
#> [1] 11
#>
#> [[4]]
#> [1] 6
```

Ça fonctionne, mais la syntaxe est un peu “lourde”.

La fonction `map()` de `purrr` propose exactement cette fonctionnalité. Elle prend deux arguments principaux :

1. un vecteur ou une liste
2. une fonction

et elle retourne une liste contenant le résultat de la fonction appliquée à chaque élément du vecteur ou de la liste.

En utilisant `map()` on peut remplacer notre boucle `for` par un simple :

```
map(mots, length)
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 15
#>
#> [[3]]
#> [1] 11
#>
#> [[4]]
#> [1] 6
```

`map` va itérer sur les éléments de `mots`, leur appliquer tour à tour la fonction `length` passée en argument, et regrouper les résultats dans une liste.

À noter qu'on peut évidemment utiliser le *pipe*.

```
mot > %>% map(length)
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 15
#>
#> [[3]]
#> [1] 11
#>
#> [[4]]
#> [1] 6
```

Ici notre résultat est une liste. Or il pourrait être simplifié sous forme de vecteur atomique, puisque tous ses éléments sont des nombres.

Si on souhaitait obtenir un vecteur numérique avec une boucle `for`, il faut soit convertir le résultat de la boucle précédente en vecteur atomique (avec `unlist()` ou `purrr::flatten_int()`), soit modifier cette boucle pour qu'elle génère plutôt un vecteur numérique :

```
resultat <- numeric(length(discours))
for (i in seq_along(mots)) {
  resultat[i] <- length(mots[[i]])
}
resultat
#> [1] 7 15 11 6
```

Mais `purrr` propose des variantes de la fonction `map` qui permettent justement de s'assurer du type de résultat obtenu. Ainsi, `map_dbl()` renverra toujours un vecteur de nombres flottants, et `map_int()` un vecteur de nombres entiers. En remplacement de la boucle `for` ci-dessus, on peut donc utiliser :

```
mot > %>% map_int(length)
#> [1] 7 15 11 6
```

18.2 `map` et ses variantes

18.2.1 Modes d'appel de `map`

L'objectif de `map` est donc d'appliquer une fonction à l'ensemble des éléments d'un vecteur ou d'une liste.

On a vu qu'on pouvait l'utiliser pour appliquer la fonction `length` à chacun des vecteurs contenus par la liste `mots`. En utilisant `map_int` on s'assure de récupérer un simple vecteur numérique, plus facile à utiliser par la suite si on souhaite par exemple calculer une moyenne.

```
mot > %>% map_int(length)
#> [1] 7 15 11 6
```

Si on souhaitait plutôt extraire le dernier mot de chaque vecteur, on pourrait créer une fonction spécifique et l'appliquer avec `map()`.

```
dernier_mot <- function(v) {
  tail(v, 1)
}

mots %>% map(dernier_mot)
#> [[1]]
#> [1] "anticonstitutionnellement"
#>
#> [[2]]
#> [1] "libertés"
#>
#> [[3]]
#> [1] "réforme"
#>
#> [[4]]
#> [1] "piraté"
```

Comme notre résultat est une liste de chaînes de caractères simples, on peut forcer le résultat à être plutôt un vecteur de type *character* en utilisant `map_chr()` :

```
mots %>% map_chr(dernier_mot)
#> [1] "anticonstitutionnellement" "libertés"
#> [3] "réforme"                  "piraté"
```

Comme notre fonction est très courte, on peut aussi préférer utiliser une *fonction anonyme*, introduites section 14.4.2.

```
mots %>% map_chr(function(v) { tail(v, 1) })
#> [1] "anticonstitutionnellement" "libertés"
#> [3] "réforme"                  "piraté"
```

On peut aussi utiliser la notation abrégée sous forme de formule, propre aux fonctions du *tidyverse*, présentée section 15.3.

```
mots %>% map_chr(~ tail(.x, 1) )
#> [1] "anticonstitutionnellement" "libertés"
#> [3] "réforme"                  "piraté"
```

On peut également utiliser la notation compacte pour les fonctions anonymes disponible sous R à partir de la version 4.1.

```
mots %>% map_chr(\(v) tail(v, 1))
#> [1] "anticonstitutionnellement" "libertés"
#> [3] "réforme"                  "piraté"
```

Enfin, si on fournit des arguments supplémentaires à `map`, ils sont passés comme argument à la fonction qu'il applique, on peut donc également utiliser la notation suivante :

```
mots %>% map_chr(tail, 1)
#> [1] "anticonstitutionnellement" "libertés"
#> [3] "réforme"                  "piraté"
```

Dans ce qui suit on utilisera de préférence la notation “formule”, mais toutes les versions ci-dessus sont équivalentes et donnent le même résultat.



Petite astuce à noter, si on transmet à `map()` autre chose qu'une fonction, elle utilisera cette information pour extraire des éléments. Ainsi, `v %>% map(1)` extraiera le premier élément de chaque élément du vecteur `v`, `v %>% map("foo")` extraiera les éléments nommés “foo”, etc.

18.2.2 Variantes de `map`

On a vu que `map` propose plusieurs variantes qui permettent de contrôler le type de résultat qu'elle retourne :

- `map()` retourne une liste
- `map_int()` retourne un vecteur atomique d'entiers
- `map_dbl()` retourne un vecteur atomique de nombres flottants
- `map_chr()` retourne un vecteur atomique de chaînes de caractères
- `map_lgl()` retourne un vecteur atomique de `TRUE / FALSE`

Attention, ces variantes sont très strictes : si la fonction appelée retourne un résultat qui n'est pas compatible avec le résultat attendu, elle génère une erreur. C'est le cas si dans le code précédent on essaie de récupérer chaque dernier mot sous forme d'un vecteur de nombres :

```
mots %>% map_dbl(tail, 1)
#> Error: Can't coerce element 1 from a character to a double
```

Pour pouvoir utiliser ces variantes et obtenir un vecteur atomique, chaque résultat retourné par la fonction appliquée doit être de longueur 1. Ainsi, si on souhaitait extraire plutôt la liste des mots contenant un “f”, certains résultats ne contiennent aucun élément, et d'autres en contiennent deux :

```
mots %>% map(~ str_subset(.x, "f") )
#> [[1]]
#> character(0)
#>
#> [[2]]
#> [1] "souffle"
#>
#> [[3]]
#> [1] "faire"    "réforme"
#>
#> [[4]]
#> character(0)
```

Dans ce cas on ne peut pas utiliser `map_chr()` : si on essaie on obtient un message d'erreur nous indiquant que certains résultats de `str_subset()` ne sont pas au bon format.

```
mots %>% map_chr(~ str_subset(.x, "f") )
#> Error: Result 1 must be a single string, not a character vector of length 0
```

Dans ce cas, on doit donc utiliser `map()` et conserver le résultat sous forme de liste, qui elle peut contenir des éléments de longueurs différentes.

```
mot >%
  map(~ str_subset(.x, "f"))
#> [[1]]
#> character(0)
#>
#> [[2]]
#> [1] "souffle"
#>
#> [[3]]
#> [1] "faire"    "réforme"
#>
#> [[4]]
#> character(0)
```

On notera qu'on peut tout à fait enchaîner les `map()` si on veut effectuer des opérations supplémentaires.

```
mot >%
  map(~ str_subset(.x, "f")) %>%
  map_int(length)
#> [1] 0 1 2 0
```

18.2.3 `map_dfr()` et `map_dfc()`

La page suivante contient les données du jeu de données `rp2018` sous la forme de fichiers CSV, avec un fichier par département :

<https://github.com/juba/tidyverse/tree/main/resources/data/rp2018>

À partir de cette page, on peut télécharger les fichiers CSV en utilisant des adresses de la forme :

https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_01.csv

En remplaçant “01” par le code du département souhaité.

On peut créer une fonction `genere_url()` qui, à partir d'une liste de codes de départements, retourne les adresses des fichiers correspondant.

```
genere_url <- function(codes) {
  paste0(
    "https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_",
    codes,
    ".csv"
  )
}

genere_url(c("42", "69"))
#> [1] "https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_42.csv"
#> [2] "https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_69.csv"
```

Grâce à la fonction `read_csv()`, on peut charger directement dans notre session R un fichier en indiquant son URL.

```
data69 <- read_csv(genere_url("69"))
```

Comment faire si l'on souhaite charger les fichiers de plusieurs départements ? La fonction `read_csv()` n'accepte qu'une seule URL à la fois, elle n'est pas vectorisée.

Dans ce cas on peut utiliser `map()` pour l'appliquer tour à tour à plusieurs URL.

```
departements <- c("38", "42", "69")
urls <- genere_url(departements)

dfs <- urls %>% map(read_csv)
```

Le résultat `dfs` est une liste de trois tableaux de données. Chacun de ces éléments est un *tibble* : d'abord celui du fichier CSV des données de l'Isère, puis celui de la Loire, et enfin celui du Rhône.

On peut itérer sur cette liste `dfs` pour appliquer une fonction à chacun de ces tableaux.

```
# Affichage des dimensions de chaque tableau
dfs %>% map(dim)
#> [[1]]
#> [1] 146 37
#>
#> [[2]]
#> [1] 56 37
#>
#> [[3]]
#> [1] 132 37
```

```
# Calcul de la moyenne de la variable dipl_aucun
dfs %>% map_dbl(~ mean(.x$dipl_aucun))
#> [1] 19.24298 23.57400 17.74882
```

```
# Calcul du coefficient associé à la variable dipl_sup dans
# la régression linéaire de cadres en fonction de dipl_sup
dfs %>% map_dbl(~ {
  reg <- lm(cadres ~ dipl_sup, data = .x)
  reg$coefficients["dipl_sup"]
})
```

#> [1] 1.132317 1.264469 1.027263

Si on souhaite réunir ces trois *tibbles* en un seul, on peut utiliser la fonction `bind_rows()` de `dplyr`.

```
departements <- c("38", "42", "69")
urls <- genere_url(departements)

df <- urls %>%
  map(read_csv) %>%
  bind_rows()
```

Mais on peut aussi utiliser une autre variante de `map()`, nommée `map_dfr()`, qui considère les résultats obtenus par l'application de la fonction comme les lignes d'un tableau de données qu'elle va automatiquement rassembler en un seul tableau, de la même manière qu'avec un `bind_rows()`.

```
df <- urls %>% map_dfr(read_csv)

df
#> # A tibble: 334 x 37
#>   code_insee commune code_region region departement log_rp log_proprio log_loc
#>   <dbl> <chr>     <dbl> <chr>    <chr>      <dbl>      <dbl>      <dbl>
#> 1 38001 Les Abr~     84 Auver~ Isère    2630.    1773.     812.
#> 2 38006 Allevard     84 Auver~ Isère    1845.    1157.     644.
#> 3 38012 Aoste        84 Auver~ Isère    1158.     762.      378.
#> 4 38013 Apprieu       84 Auver~ Isère    1316.    1086.     211.
#> 5 38022 Les Ave~     84 Auver~ Isère    3388.    2452.     885.
#> 6 38034 Beaurep~     84 Auver~ Isère    2116.    1189.     885.
#> 7 38039 Bernin       84 Auver~ Isère    1244.    1028.     199.
#> 8 38045 Biviers      84 Auver~ Isère    1015.     870.      126.
#> 9 38052 Le Bour~     84 Auver~ Isère    1438.     890.      494.
#> 10 38053 Bourgoi~    84 Auver~ Isère   13150.    5273.    7569.
#> # ... with 324 more rows, and 29 more variables: log_hlm <dbl>, log_sec <dbl>,
#> #   log_maison <dbl>, log_appart <dbl>, age_0_14 <dbl>, age_15_29 <dbl>,
#> #   age_75p <dbl>, femmes <dbl>, chom <dbl>, agric <dbl>, indep <dbl>,
#> #   cadres <dbl>, interm <dbl>, empl <dbl>, ouvr <dbl>, etud <dbl>,
#> #   dipl_aucun <dbl>, dipl_bepc <dbl>, dipl_capbep <dbl>, dipl_bac <dbl>,
#> #   dipl_sup2 <dbl>, dipl_sup34 <dbl>, dipl_sup <dbl>, resid_sec <dbl>,
#> #   proprio <dbl>, locataire <dbl>, hlm <dbl>, maison <dbl>, appart <dbl>
```

Les deux lignes de code ci-dessus partent donc d'une liste d'identifiants de départements, génèrent les URL des fichiers CSV correspondant, les importent dans R et assemblent le résultat en un seul tableau. Plutôt efficace !

Une fonction utile en complément de `map_dfr()` est la fonction `list.files()`, qui peut lister les fichiers ayant une certaine extension dans un dossier spécifique. Par exemple, l'instruction suivante liste tous les fichiers se terminant par `.csv` du sous-dossier `data`.

```
fichiers <- list.files("data", "*.csv", full.names = TRUE)
```

On peut dès lors utiliser `map_dfr()` et `read_csv()` pour lire tous ces fichiers en une seule fois et les concaténer en un seul tableau de données.

```
d <- fichiers %>% map_dfr(read_csv)
```



Il existe également une variante `map_dfc()` qui considère les résultats comme des colonnes d'un tableau de données et les rassemble en un seul tableau comme le ferait la fonction `bind_cols()` de `dplyr`.

18.3 Itérer sur les colonnes d'un tableau de données

On a vu section 16.3 que les tableaux de données (*data frame* ou *tibble*) sont en fait des listes dont les éléments sont les colonnes du tableau. Si on applique `map()` à un tableau, celle-ci itérera donc sur ses colonnes.

Par exemple, on peut appliquer `n_distinct` au jeu de données `starwars` et obtenir le nombre de valeurs distinctes de chacune de ses colonnes.

```
starwars %>% map_int(n_distinct)
#>   name      height      mass hair_color skin_color eye_color birth_year
#>   87       46       39       13       31       15       37
#>   sex      gender homeworld species     films vehicles starships
#>   5        3       49       38       24       11       17
```

Le résultat est équivalent à celui qu'on obtient en faisant un `summarise()` sur l'ensemble des colonnes, comme vu section 15.2, sauf que `map_int()` retourne un vecteur numérique tandis que `summarise()` renvoie un *tibble* à une ligne.

```
starwars %>%
  summarise(
    across(everything(), n_distinct)
  )
#> # A tibble: 1 x 14
#>   name height mass hair_color skin_color eye_color birth_year   sex gender
#>   <int> <int> <int> <int> <int> <int> <int> <int>
#> 1   87     46   39     13     31     15     37     5     3
#> # ... with 5 more variables: homeworld <int>, species <int>, films <int>,
#> #   vehicles <int>, starships <int>
```

De la même manière, si on veut connaître le nombre de valeurs manquantes pour chaque variable :

```
starwars %>% map_int(~ sum(is.na(.x)) )
#>   name      height      mass hair_color skin_color eye_color birth_year
#>   0       6       28       5       0       0       44
#>   sex      gender homeworld species     films vehicles starships
#>   4       4       10       4       0       0       0
```

Contrairement à `across()`, on ne peut pas spécifier directement une sélection de colonnes à `map()`. On peut par contre utiliser des fonctions comme `keep()` ou `discard()` qui “filtrent” les éléments d’une liste via une fonction qui renvoie TRUE ou FALSE.

On peut par exemple utiliser `discard()` après `map()` pour ne conserver que les colonnes ayant au moins une valeur NA.

```
starwars %>%
  map_int(~ sum(is.na(.x)) ) %>%
  discard(~ .x == 0 )
#>   height      mass hair_color birth_year      sex   gender homeworld
#>   6       28       5       44       4       4       10
#>   species
#>   4
```

Ou bien utiliser `keep()` pour n’appliquer `mean()` qu’aux variables numériques.

```
starwars %>%
  keep(is.numeric) %>%
  map_dbl(mean, na.rm = TRUE)
#>   height      mass birth_year
#> 174.35802  97.31186  87.56512
```

18.4 modify

`modify()` est une variante de `map()` qui a pour particularité de renvoyer un résultat du même type que la liste ou le vecteur donné en entrée.

Ainsi, si on l'applique à un vecteur de chaînes de caractères, le résultat sera aussi un vecteur de chaînes de caractères même si la fonction appliquée retourne un résultat numérique.

```
v <- c("brouette", "moto", "igloo")
v %>% modify(length)
#> [1] "1" "1" "1"
```

Si on l'applique à une liste, le résultat sera aussi une liste.

```
v <- list("brouette", "moto", "igloo")
v %>% modify(length)
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 1
#>
#> [[3]]
#> [1] 1
```

L'objectif de `modify()` est de permettre de “modifier” une liste ou un vecteur en lui appliquant une fonction tout en étant sûr qu'on ne va pas modifier son type.

L'intérêt principal de `modify()` est qu'elle propose deux variantes, `modify_if()` et `modify_at()`, qui sélectionnent les éléments respectivement via une fonction et via leur nom ou leur position, et qui n'appliquent la fonction de transformation qu'aux éléments sélectionnés.

Cela peut être particulièrement utile quand on l'applique à un tableau de données. Par exemple le code suivant utilise `modify_if()` pour transformer uniquement les colonnes de type `factor` de `hdv2003` en `character`, et laisser les autres inchangées.

```
hdv2003 %>% modify_if(is.factor, as.character)
```

On notera qu'on obtient le même résultat avec le code suivant qui utilise `across()` de `dplyr`.

```
hdv2003 %>%
  mutate(
    across(
      where(is.factor),
      as.character
    )
  )
```

`modify_at` permet d'appliquer une fonction à certaines variables à partir de leurs noms.

```
hdv2003 %>%
  modify_at(c("sexé", "qualif"), as.character)
```

En utilisant `vars()`, on peut sélectionner les variables avec toutes les possibilités offertes par la *tidy selection*.

```
hdv2003 %>%
  modify_at(vars(hard.rock:sport), as.character)
```

Là aussi, on peut obtenir le même résultat en utilisant `across()`.

```
hdv2003 %>%
  mutate(
    across(
      hard.rock:sport,
      as.character
    )
  )
```

18.5 `imap`

Imaginons que nous avons récupéré les données suivantes, qui représentent des évaluations obtenues par quatre restaurants, sous la forme d'une liste.

```
restos <- list(
  "La bonne fourchette" = c(3, 3, 2, 5, 2, 3, 2, 4, 1, 3),
  "La choucroute de l'amer" = c(4, 1, 2, 4, 2, 5, 2),
  "L'Hair de rien" = c(1, 5, 5, 1, 5, 3, 1, 5, 2),
  "La blanquette de Vaulx" = c(4, 1, 3, 1, 3, 3, 1, 4, 2, 5)
)
```

À partir de cette liste, on souhaite créer un tableau de données comportant la moyenne et l'écart-type des notes de chaque restaurant. Comme on l'a vu précédemment, cela peut se faire avec l'aide de `map_dfr()`.

```
restos %>% map_dfr(~ tibble(moyenne = mean(.x), ecart_type = sd(.x)) )
#> # A tibble: 4 x 2
#>   moyenne ecart_type
#>   <dbl>     <dbl>
#> 1     2.8      1.14
#> 2     2.86     1.46
#> 3     3.11     1.90
#> 4     2.7      1.42
```

On obtient le tableau souhaité, mais il manque une information : le nom du restaurant correspondant à chaque ligne. Cette information est incluse dans les noms des éléments de la liste `restos`, or la fonction passée à `map_dfr` n'y a pas accès, elle n'a accès qu'à leurs valeurs.

C'est pour ce type de cas de figure que `purrr` propose la famille de fonctions `imap()`. Celle-ci fonctionne de la même manière que `map()`, sauf que la fonction appliquée prend deux arguments : d'abord la valeur de l'élément courant, puis son nom.

Dans l'exemple suivant, on applique `imap()` à une liste simple et on affiche un message avec le nom et la valeur de chaque élément.

```
1 <- list(nom1 = 1, nom2 = 3)

12 <- 1 %>% imap(function(valeur, nom) {
  message("La valeur de ", nom, " est ", valeur)
})
#> La valeur de nom1 est 1
#> La valeur de nom2 est 3
```

On peut évidemment utiliser la notation “formule” de `purrr`, il faut juste se souvenir que dans ce cas `.x` correspond à la valeur, et `.y` au nom.

```
12 <- 1 %>% imap(~ {
  message("La valeur de ", .y, " est ", .x)
})
#> La valeur de nom1 est 1
#> La valeur de nom2 est 3
```

Tout comme `map()` proposait les variantes `map_int()`, `map_chr()` ou `map_dfr()`, on peut également utiliser `imap_dbl()`, `imap_chr()` et autres `imap_dfc()` pour forcer le type de résultat retourné.

Pour reprendre notre exemple de départ, on peut donc, en utilisant `imap`, récupérer le nom de l'élément courant de la liste `restos` et l'utiliser pour rajouter le nom du restaurant dans notre *tibble* de résultats.

On peut le faire avec une fonction anonyme “classique” :

```
restos %>% imap_dfr(function(notes, nom) {
  tibble(resto = nom, moyenne = mean(notes), ecart_type = sd(notes))
})
#> # A tibble: 4 x 3
#>   resto           moyenne   ecart_type
#>   <chr>          <dbl>      <dbl>
#> 1 La bonne fourchette    2.8       1.14
#> 2 La choucroute de l'amer 2.86      1.46
#> 3 L'Hair de rien        3.11      1.90
#> 4 La blanquette de Vaulx 2.7       1.42
```

On peut aussi utiliser la notation compacte de type formule, en se souvenant à nouveau que `.x` correspond à la valeur de l'élément courant, et `.y` à son nom.

```
restos %>% imap_dfr(~ {
  tibble(resto = .y, moyenne = mean(.x), ecart_type = sd(.x))
})
#> # A tibble: 4 x 3
#>   resto           moyenne   ecart_type
#>   <chr>          <dbl>      <dbl>
#> 1 La bonne fourchette    2.8       1.14
#> 2 La choucroute de l'amer 2.86      1.46
#> 3 L'Hair de rien        3.11      1.90
#> 4 La blanquette de Vaulx 2.7       1.42
```



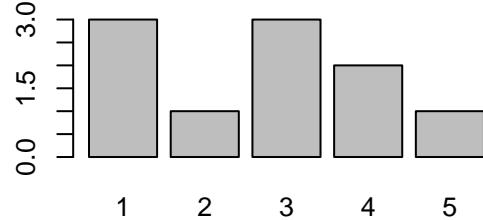
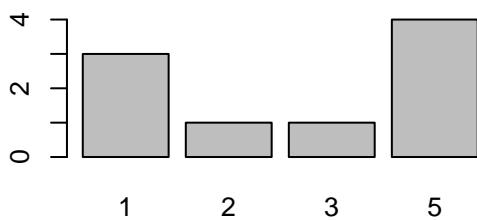
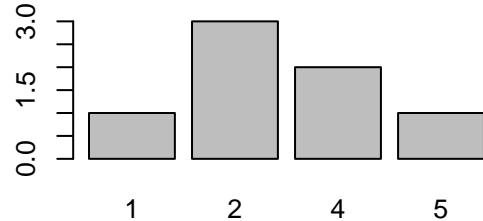
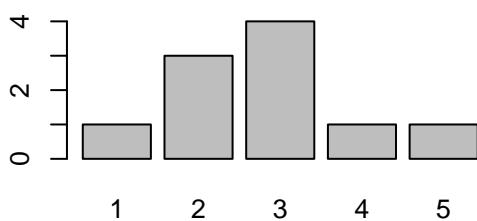
Si on utilise `imap()` sur une liste ou un vecteur qui n'a pas de noms, le deuxième argument passé à la fonction appliquée sera l'indice de l'élément courant : 1 pour le premier, 2 pour le deuxième, etc.

18.6 walk

`walk()` est une variante de `map()` qui a pour particularité de ne pas retourner de résultat. On l'utilise lorsqu'on souhaite parcourir un vecteur ou une liste et appliquer à ses éléments une fonction dont on ne souhaite conserver que les "effets de bord" : afficher un message, générer un graphique, enregistrer un fichier...

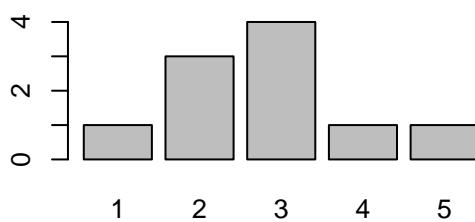
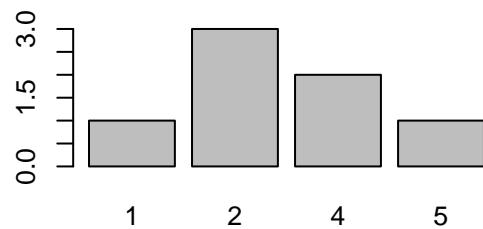
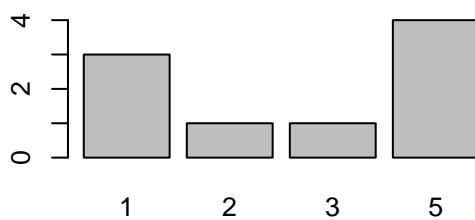
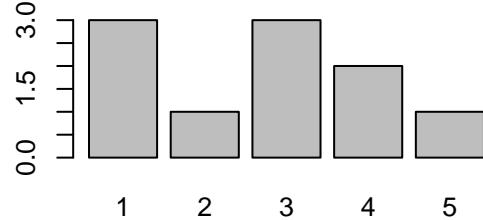
Par exemple, le code suivant génère quatre diagrammes en barre indiquant la répartition des notes des différents restaurants vus dans la section précédente.

```
walk(restos, ~ barplot(table(.x)) )
```



Comme pour `map()`, la variante `iwalk()` permet d'itérer à la fois sur les valeurs et sur les noms des éléments du vecteur ou de la liste. Ceci permet par exemple d'afficher le nom du restaurant comme titre de chaque graphique.

```
iwalk(restos, ~ barplot(table(.x), main = .y) )
```

La bonne fourchette**La choucroute de l'amer****L'Hair de rien****La blanquette de Vaulx**

Au final, on notera que l'utilisation de `walk()`, comme elle ne retourne pas de résultats, est très proche de celle d'une boucle `for`.

18.7 `map2` et `pmap` : itérer sur plusieurs vecteurs en parallèle

Supposons qu'un.e collègue, qui travaille avec nous sur le jeu de données `rp2018`, nous a envoyé une liste de variables dont elle voudrait connaître les corrélations. Cette liste a été saisie dans un tableau sur deux colonnes, chaque ligne indiquant deux variables pour lesquelles elle souhaite qu'on effectue ce calcul.

Après importation dans R on obtient le tableau de données suivant.

```
correlations <- tribble(
  ~var1,      ~var2,
  "dipl_sup", "dipl_aucun",
  "dipl_sup", "cadres",
  "hlm",      "cadres",
  "hlm",      "ouvr",
  "proprio",  "hlm"
)

correlations
#> # A tibble: 5 x 2
#>   var1     var2
#>   <chr>    <chr>
#> 1 dipl_sup dipl_aucun
#> 2 dipl_sup cadres
#> 3 hlm      cadres
#> 4 hlm      ouvr
#> 5 proprio  hlm
```

Pour pouvoir calculer les corrélations souhaitées, on doit itérer sur les deux vecteurs `var1` et `var2` *en parallèle*, et calculer la corrélation entre la colonne de `rp2018` correspondant à la valeur courante de `var1` et celle correspondant à la valeur courante de `var2`.

C'est précisément ce que fait la fonction `map2()`. Celle-ci prend trois arguments en entrée :

1. deux listes ou vecteurs qui seront itérés en parallèle
2. une fonction qui accepte deux arguments : ceux-ci prendront tour à tour les deux valeurs courantes des deux listes ou vecteurs itérés

On peut donc utiliser `map2` pour itérer parallèlement sur les deux colonnes `var1` et `var2` de notre tableau `correlations`, et calculer la corrélation des deux colonnes correspondantes de `rp2018`.

```
map2(
  correlations$var1,
  correlations$var2,
  ~ cor(rp2018[[.x]], rp2018[[.y]])
)
#> [[1]]
#> [1] -0.6146729
#>
#> [[2]]
#> [1] 0.9291504
#>
#> [[3]]
#> [1] -0.1067832
#>
#> [[4]]
#> [1] 0.2126366
#>
#> [[5]]
#> [1] -0.7786399
```

`map2()` propose les mêmes variantes `map2_int()`, `mapr2_chr()`, etc. que `map()`. On peut donc utiliser `map2_dbl()` pour récupérer un vecteur numérique plutôt qu'une liste, et l'utiliser par exemple pour rajouter une colonne à notre tableau de départ.

```
correlations$corr <- map2_dbl(
  correlations$var1,
  correlations$var2,
  ~ cor(rp2018[[.x]], rp2018[[.y]])
)

correlations
#> # A tibble: 5 x 3
#>   var1     var2       corr
#>   <chr>    <chr>    <dbl>
#> 1 dipl_sup dipl_aucun -0.615
#> 2 dipl_sup cadres     0.929
#> 3 hlm      cadres     -0.107
#> 4 hlm      ouvr       0.213
#> 5 proprio  hlm       -0.779
```

Si on souhaite uniquement capturer les effets de bord sans récupérer les résultats de la fonctions appliquée, on peut aussi utiliser la variante `walk2()`.

Supposons maintenant que notre collègue nous a envoyé, toujours sous la même forme, une liste de variables dont elle souhaite obtenir un nuage de points, mais en fournissant également un titre à ajouter au graphique. On obtient le tableau suivant :

```
nuages <- tribble(
  ~var1,      ~var2,      ~titre,
  "dipl_sup", "dipl_aucun", "Diplômés du supérieur x sans diplôme",
  "dipl_sup", "cadres",     "Pourcentage de cadres x diplômés du supérieur",
  "hlm",       "cadres",     "Pas facile de trouver un titre",
  "proprio",   "cadres",     "Oui non vraiment c'est pas simple"
)

nuages
#> # A tibble: 4 x 3
#>   var1     var2     titre
#>   <chr>    <chr>    <chr>
#> 1 dipl_sup dipl_aucun Diplômés du supérieur x sans diplôme
#> 2 dipl_sup cadres     Pourcentage de cadres x diplômés du supérieur
#> 3 hlm       cadres     Pas facile de trouver un titre
#> 4 proprio   cadres     Oui non vraiment c'est pas simple
```

On est dans une situation similaire à la précédente, sauf que cette fois on doit itérer sur trois vecteurs en parallèle. On va donc utiliser la fonction `pmap()` qui permet d'itérer sur autant de listes ou vecteurs que l'on souhaite. Plus précisément, comme on souhaite générer des graphiques on va utiliser la variante `pwalk()` qui ne retourne pas de résultat.

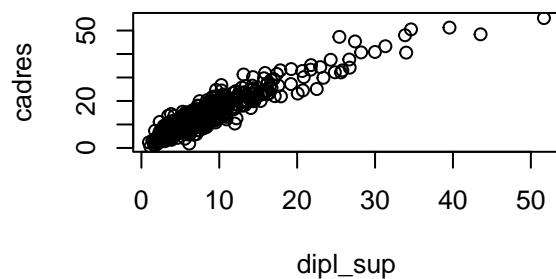
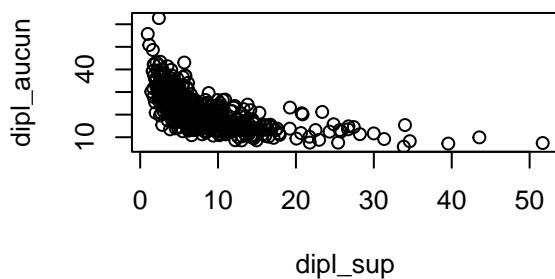
`pmap()` et `pwalk()` prennent deux arguments principaux :

- les vecteurs et listes sur lesquels itérer, eux-mêmes regroupés dans une liste
- une fonction acceptant autant d'arguments que de vecteur ou listes sur lesquels on itère

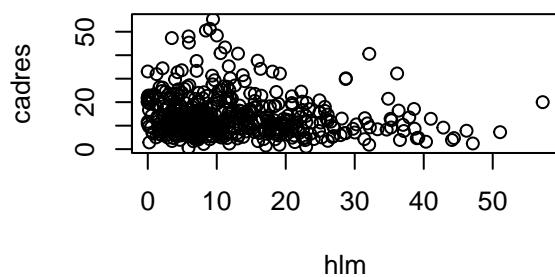
Dans notre exemple on aurait donc un appel de la forme suivante.

```
pwalk(
  list(nuages$var1, nuages$var2, nuages$titre),
  function(var1, var2, titre) {
    plot(
      rp2018[[var1]], rp2018[[var2]],
      xlab = var1, ylab = var2, main = titre
    )
  }
)
```

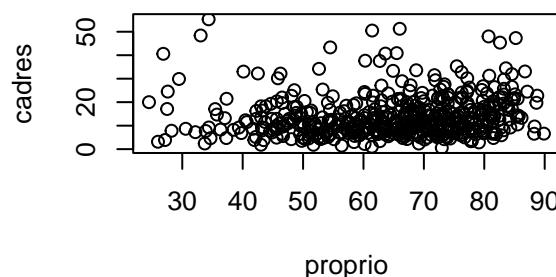
Diplômés du supérieur x sans diplôme oucentage de cadres x diplômés du supérieur



Pas facile de trouver un titre



Oui non vraiment c'est pas simple



Petite précision, si la liste est nommée, il faut que les noms des arguments de la fonction correspondent aux noms de la liste.

```
pwalk(
  list(v1 = nuages$var1, v2 = nuages$var2, titre = nuages$titre),
  function(v1, v2, titre) {
    plot(
      rp2018[[v1]], rp2018[[v2]],
      xlab = v1, ylab = v2, main = titre
    )
  }
)
```

À noter que comme `nuages` est un tableau de données, donc une liste dont les éléments sont ses colonnes, on obtient le même résultat avec :

```
pwalk(
  nuages,
  function(var1, var2, titre) {
    plot(
      rp2018[[var1]], rp2018[[var2]],
      xlab = var1, ylab = var2, main = titre
    )
  }
)
```

On peut utiliser la syntaxe “formule” pour la fonction anonyme, dans ce cas les arguments sont accessibles avec la notation `.1`, `.2`, etc. On notera que dans ce cas la syntaxe “formule” est sans doute moins lisible que la syntaxe classique avec `function()` qui permet de nommer les paramètres.

```
pwalk(
  nuages,
  ~ {
    plot(
      rp2018[..1], rp2018[..2],
      xlab = ..1, ylab = ..2, main = ..3
    )
  }
)
```

Comme pour `map()` et `map2()`, `pmap()` propose aussi les variantes `pmap_int()`, `pmap_chr()`, etc.

18.8 Répéter une opération

Les fonctions de `purrrr` peuvent être utilisées quand on souhaite juste répéter une opération un certain nombre de fois, à la place d'une boucle `for`.

Par exemple si on souhaite générer 10 vecteurs de 100 nombres aléatoires, on pourra remplacer la boucle suivante :

```
res <- list()
for (i in 1:10) {
  res <- c(res, rnorm(100))
}
```

Par un appel à `map()` :

```
res <- map(1:10, ~ rnorm(100))
```

Ce qui donne un code un peu plus compact et plus lisible.



De la même manière, si on s'intéresse juste aux effets de bord, on pourra éventuellement remplacer une boucle `for` par un appel à `walk()`.

18.9 Quand (ne pas) utiliser `map`

Une fois qu'on a compris la logique de `map()` et de ses variantes, on peut être tenté.es de l'appliquer un peu systématiquement. Il faut cependant garder en tête que son usage n'est pas toujours conseillé notamment dans les cas où il existe déjà une fonction vectorisée qui permet d'obtenir le même résultat.

Ainsi cela n'aurait pas de sens de faire :

```
v <- 1:5
map_dbl(v, ~ .x + 10)
#> [1] 11 12 13 14 15
```

Quand on peut simplement faire :

```
v + 10
#> [1] 11 12 13 14 15
```

Pour prendre un exemple un peu moins caricatural, de nombreuses fonctions de `stringr` sont vectorisées, il n'est donc pas utile de faire :

```
textes <- c("fantastique", "effectivement", "igloo")
map_int(textes, ~ str_count(.x, "f"))
#> [1] 1 2 0
```

Quand on peut faire simplement :

```
str_count(textes, "f")
#> [1] 1 2 0
```

Par contre `map()` est utile quand on souhaite appliquer une fonction qui n'est pas vectorisée à plusieurs valeurs, comme c'est le cas par exemple avec `read_csv()`, qui ne permet pas de charger plusieurs fichiers d'un coup :

```
fichiers <- c("fichier1.csv", "fichier2.csv")
l <- fichiers %>% map(read_csv)
```

Où quand on veut itérer sur un argument non vectorisé, par exemple ici sur l'argument `pattern` de `str_count()` :

```
voyelle <- c(a = "a", e = "e", i = "i")
textes <- c("brouette", "moto", "igloo")
voyelle %>% map(~ str_count(textes, pattern = .x))
#> $a
#> [1] 0 0 0
#>
#> $e
#> [1] 2 0 0
#>
#> $i
#> [1] 0 0 1
```

On l'utilise également quand on veut appliquer une fonction non pas à une liste, mais aux éléments qu'elle contient :

```
l <- list(1:3, c(2, 5))
l %>% map_int(length)
#> [1] 3 2
```

À noter qu'en termes de performance, `map()` n'est pas forcément plus rapide qu'une boucle `for`, puisque dans les deux cas on itère sur un ensemble de valeurs. Par contre une fonction vectorisée existante sera toujours (beaucoup) plus rapide.

18.10 purrr vs *apply

Les fonctions de *purrr* ont des équivalents dans R “de base”, ce sont notamment les fonctions de la famille *apply* : *lapply*, *sapply*, *mapply*...

L'avantage de *map()* et des autres fonctions fournies par *purrr* et qu'elles sont plus explicites : on a des fonctions différentes selon qu'on veut seulement appliquer une fonction (*map()*), générer des effets de bord (*walk*), modifier une liste sans changer son type (*modify()*), etc. *purrr* propose également de nombreuses fonctions utiles qui facilite le travail avec les vecteurs et listes.

Mais un des avantages principaux des fonctions de la famille *map()* est qu'elles sont consistantes et cohérentes dans le type de résultat qu'elles retournent : on est certain que *map()* ou *imap()* retourneront une liste, que *map_chr()* ou *map2_chr()* retourneront un vecteur de chaînes de caractères, etc.

Là encore, il n'est pas question de dire qu'il ne faut pas utiliser les fonctions **apply*. Si vous en avez l'habitude et qu'elles fonctionnent pour vous, il n'y a pas spécialement de raison de changer. Mais si vous n'avez pas l'habitude de ce type d'opérations ou si vous préférez une syntaxe plus cohérente et plus facile à retenir, les fonctions de *purrr* peuvent être intéressantes.

Si vous souhaitez en savoir plus, l'ouvrage en ligne *R for data science* contient [une comparaison plus détaillée](#) des deux familles de fonctions.

18.11 Ressources

Au-delà de celles présentées ici, *purrr* propose de nombreuses autres fonctions facilitant la manipulation et les itérations sur les listes et les vecteurs. On peut en trouver la [liste complète](#) et la documentation associée (en anglais) sur [le site de l'extension](#).

La section *Iteration* de l'ouvrage en ligne *R for data science* (en anglais) propose [une présentation de plusieurs fonctions de purrr](#).

RStudio propose une [antisèche](#) (en anglais, format PDF) qui résume les différentes fonctions de *purrr*.

Sur le blog en français de Lise Vaudor, on trouvera un billet [Itérer des fonctions avec purrr](#) et une suite [practice makes purrr-fect](#).

18.12 Exercices

18.12.1 map et ses variantes

Exercice 1.1

La liste suivante rassemble les notes obtenues par un élève dans différentes matières.

```
notes <- list(
  maths = c(12, 15, 8, 10),
  anglais = c(18, 11, 9),
  sport = c(5, 13),
  musique = 14
)
```

En utilisant *map()*, calculer une liste indiquant la moyenne dans chaque matière.

En utilisant une variante de *map()*, simplifier le résultat pour obtenir un vecteur numérique.

On a rajouté à la liste les notes obtenues en technologie, parmi lesquelles une note est manquante.

```
notes <- list(
  maths = c(12, 15, 8, 10),
  anglais = c(18, 11, 9),
  sport = c(5, 13),
  musique = 14,
  techno = c(12, NA)
)
```

Calculer à nouveau un vecteur numérique des moyennes par matière, mais sans tenir compte de la valeur manquante.

Calculer une liste qui contient pour chaque matière le nombre de notes, leur moyenne et leur écart-type.

Exercice 1.2

La liste suivante comporte les parcours biographiques de 5 personnes sous la forme de vecteurs indiquant leurs communes de résidence successives.

```
parcours <- list(
  c("Lyon", "F1ixeville", "Saint-Dié-en-Pouilly"),
  c("Sainte-Gabellie-sur-Sarthe"),
  c("Décines", "Meyzieu", "Demptezieu"),
  c("Meyzieu", "Lyon", "Paris", "F1ixeville", "Lyon"),
  c("La Bâtie-Divisin", "Versailles")
)
```

À l'aide de `map()`, calculer une nouvelle liste comportant le nombre de villes de résidence pour chaque parcours.

Utiliser une variante de `map()` pour simplifier le résultat et obtenir un vecteur numérique plutôt qu'une liste.

Déterminer pour chaque parcours le nombre de fois où la personne a résidé à Lyon.

On vient de repérer un problème dans les données : des caractères “l” ont été remplacés par des “1”. Utiliser `map()` pour corriger l'objet `parcours` en remplaçant tous les “1” par des “l”.

Exercice 1.3

Le vecteur suivant contient les adresses de deux fichiers CSV contenant les données de `rp2018` pour les départements de l'Ain et du Rhône :

```
urls <- c(
  "https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_01.csv",
  "https://raw.githubusercontent.com/juba/tidyverse/main/resources/data/rp2018/rp2018_69.csv"
)
```

Utiliser `map()` pour charger ces deux tableaux de données dans une liste nommée `dfs`.

Utiliser `map_dfr()` pour charger ces deux tableaux de données et les regrouper en une seule table `d`. Que constatez-vous ?

À l'aide de `map()`, afficher la variable `code_insee` des deux tableaux de `dfs`. D'où vient le problème ?

Trouver une solution pour corriger le problème.

18.12.2 `modify` et itération sur les colonnes d'un tableau

Exercice 2.1

Soit le tableau de données `d` suivant :

```
d <- tribble(
  ~prenom,           ~nom,           ~age, ~taille,
  "pierre-edmond", "multinivo",    19,   151,
  "YVONNE-HENRI",  "QUIDEU",       73,   182,
  "jean-adélaïde", "hacépé",      27,   NA
)
```

Utiliser `typeof()` et `map()` pour afficher le type de toutes les colonnes de `d`.

En utilisant `keep()` et `map()`, calculer la moyenne de toutes les variables numériques de `d`.

En utilisant `modify_if()`, appliquer `str_to_title()` à toutes les colonnes de type `character` pour corriger la capitalisation des noms et prénoms.

À l'aide de la fonction `discard()`, supprimer de `d` les colonnes qui comportent des `NA`.

18.12.3 imap et walk

Exercice 3.1

On reprend la liste de notes vue dans un exercice précédent.

```
notes <- list(
  maths = c(12, 15, 8, 10),
  anglais = c(18, 11, 9),
  sport = c(5, 13),
  musique = 14,
  techno = c(12, NA)
)
```

Sans utiliser de boucle `for`, afficher avec la fonction `message()` les valeurs de chaque moyenne tour à tour de manière à obtenir le résultat suivant :

```
#> 11.25
#> 12.6666666666667
#> 9
#> 14
#> 12
```

De la même manière afficher pour chaque matière les messages suivants :

```
#> Votre moyenne en maths est 11.25
#> Votre moyenne en anglais est 12.6666666666667
#> Votre moyenne en sport est 9
#> Votre moyenne en musique est 14
#> Votre moyenne en techno est 12
```

Construire un tableau de données avec une matière par ligne et deux colonnes contenant le nom de la matière et la valeur de la moyenne.

```
#> # A tibble: 5 x 2
#>   matiere moyenne
#>   <chr>     <dbl>
#> 1 maths      11.2
#> 2 anglais    12.7
#> 3 sport      9
#> 4 musique    14
#> 5 techno     12
```

Exercice 3.2

Lors de la présentation des boucles `for`, on a vu la fonction suivante qui affiche les dimensions de chaque tableau de données contenu dans une liste.

```
affiche_dimensions <- function(dfs) {
  for (i in seq_along(dfs)) {
    name <- names(dfs)[[i]]
    df <- dfs[[i]]
    message("Dimensions de ", name, " : ", nrow(df), "x", ncol(df))
  }
}

l <- list(
  hdv = hdv2003,
  rp = rp2018
)

affiche_dimensions(l)
#> Dimensions de hdv : 2000x20
#> Dimensions de rp : 5417x62
```

Réécrire `affiche_dimensions()` en utilisant `iwalk()`.

Exercice 3.3

Lors de la présentation des boucles, on a vu la fonction `summaries` suivante, qui prend en paramètre un tableau de données et une liste de noms de colonnes, et applique la fonction `summary()` à ces colonnes.

```
summaries <- function(d, vars) {
  for (var in vars) {
    message("--- ", var, " ---")
    print(summary(d[, var]))
  }
}

summaries(hdv2003, c("sexe", "age"))
```

Réécrire la fonction sans utiliser de boucle `for`.

Modifier cette nouvelle fonction en utilisant `keep()` ou `discard()`, pour qu'elle fonctionne sans erreur même si l'une des variables passées en arguments n'existe pas.

```
summaries(hdv2003, c("sexe", "igloo", "age"))
#> --- sexe ---
#>     sexe
```

```
#> Homme: 899
#> Femme: 1101
#> --- age ---
#>      age
#> Min.   :18.00
#> 1st Qu.:35.00
#> Median :48.00
#> Mean   :48.16
#> 3rd Qu.:60.00
#> Max.   :97.00
```

18.12.4 map2 et pmap

Exercice 4.1

Soit le tableau de données suivant, qui indique des couples de variables du jeu de données `hdv2003`.

```
croisements <- tribble(
  ~v1,      ~v2,
  "qualif", "clso",
  "qualif", "cinema",
  "sexe",    "clso",
  "sexe",    "cinema"
)

croisements
#> # A tibble: 4 x 2
#>   v1     v2
#>   <chr>  <chr>
#> 1 qualif clso
#> 2 qualif cinema
#> 3 sexe   clso
#> 4 sexe   cinema
```

À l'aide de `map2()`, calculer une liste qui contient les tableaux croisés des quatre couples de variables de `croisements`.

Modifier le code précédent pour obtenir une liste comportant, pour chaque couple de variables :

1. les noms des deux variables croisées
2. le résultat du test du χ^2 appliqué aux deux variables

Facultatif : filtrer la liste obtenue à l'étape précédente pour ne conserver que les éléments dont la p-value du test du χ^2 est inférieure à 0.001.

Exercice 4.2

Pour rappel, on peut simuler un tirage au sort de type “pile ou face” avec la fonction `sample()` de la manière suivante :

```
sample(
  c("pile", "face"),
  size = 10,
  replace = TRUE,
  prob = c(0.5, 0.5)
)
#> [1] "pile" "pile" "face" "face" "pile" "face" "face" "face" "pile" "pile"
```

Le tableau de données suivant définit trois types de tirages aléatoires différents. `elements` contient les éléments parmi lesquels on tire, `probas` les probabilités de chaque élément d'être tiré, et `n` le nombre de tirages.

```
tirages <- tribble(
  ~elements,           ~probas,      ~n,
  c("pile", "face"),   c(0.5, 0.5),  1000,
  c("rouge", "noire"), c(0.1, 0.9),  500,
  1:6,                 rep(1/6, 6),  800
)
```

À l'aide de `pmap()` et de `sample()`, créer une liste contenant les résultats des simulations de ces trois séries de tirages.

Représenter avec `barplot()` les résultats obtenus.

18.12.5 Répéter une opération

Exercice 5.1

La fonction suivante simule le lancer de `n` dés à 6 faces.

```
lancer_des <- function(n) {
  sample(1:6, size = n, replace = TRUE)
}

lancer_des(3)
#> [1] 1 2 5
```

On souhaite utiliser cette fonction pour générer 10 lancers de 4 dés. On essaie avec le code suivant :

```
map(1:10, lancer_des, 4)
```

Est-ce que cela fonctionne ? Pourquoi ?

Trouver une solution en corrigeant le code.

Exercice 5.2

Comme vu précédemment, la fonction suivante permet de simuler 20 lancers de pièces et de stocker le résultat dans un vecteur de chaînes de caractères.

```
sample(c("pile", "face"), size = 20, replace = TRUE)
#> [1] "pile" "pile" "pile" "pile" "face" "face" "face" "face" "pile" "pile"
#> [11] "pile" "face" "pile" "face" "pile" "pile" "pile" "pile" "face" "face"
```

À l'aide de `map()` et de ce code, créer une liste nommée `sims` contenant le résultat de 100 simulations de 20 lancers.

Toujours à l'aide de `map()`, calculer le nombre de “pile” obtenus pour chaque simulation de `sims`.

Déterminer le nombre minimal et le nombre maximal de “pile” obtenus parmi toutes les simulations de `sims`.

Facultatif : à l'aide de la fonction `head_while()`, déterminer pour chaque simulation de `sims` le nombre de “pile” consécutifs obtenus avant le premier “face”.

Facultatif : la fonction `rle()` permet de calculer les *run length encoding* d'un vecteur.

```
rle(c("pile", "face", "face", "face", "pile", "pile"))
#> Run Length Encoding
#>   lengths: int [1:3] 1 3 2
#>   values : chr [1:3] "pile" "face" "pile"
```

À l'aide de cette fonction et de `map_int()`, calculer pour chaque simulation la longueur de la plus grande série consécutive de “pile”.

Chapitre 19

Programmer avec le *tidyverse*

Dans les parties précédentes, nous avons vu comment créer et utiliser nos propres fonctions. Cependant, un lecteur.trice attentif.ve aura remarqué que nous avons rigoureusement évité d'utiliser des fonctions du *tidyverse*, notamment de `dplyr`, de `tidyr` et de `ggplot2`, dans les fonctions que nous avons créées jusqu'ici.

Nous allons voir dans cette section les spécificités liées à certaines fonctions du *tidyverse* et les manières de les utiliser pour programmer et les inclure dans nos propres fonctions.

On commence avant toute chose par charger le *tidyverse* et le jeu de données `starwars` de `dplyr`.

```
library(tidyverse)
data(starwars)
```



Les notions et fonctions abordées dans cette partie sont propres aux extensions du *tidyverse*. Elles ne sont en général pas utilisables ailleurs dans R.

19.1 Spécificités du *tidyverse*

Les extensions du *tidyverse* fournissent une syntaxe lisible et agréable à utiliser lorsqu'on les utilise de manière interactive. Par exemple, si on veut filtrer les lignes et les colonnes d'un tableau de données en base R, on utilise quelque chose comme :

```
starwars[starwars$species == "Droid" & starwars$eye_color == "red", c("height", "mass")]
```

Tandis qu'avec `dplyr` on peut faire :

```
starwars %>%
  filter(species == "Droid" & eye_color == "red") %>%
  select(height, mass)
```

L'avantage de `dplyr` est double :

1. dans `filter`, on sait qu'on travaille à l'intérieur du tableau de données, on indique donc juste `species` et pas `starwars$species`
2. dans `select`, on peut indiquer les noms de colonnes en omettant les guillemets

Cette simplification pour une utilisation interactive, la plus fréquente, entraîne cependant une complexification lorsqu'on souhaite utiliser ces fonctions pour programmer, notamment quand on veut les utiliser à l'intérieur d'autres fonctions.

19.1.1 *data masking*

On part du tableau de données d'exemple suivant¹.

```
restos <- tibble(
  nom = c("Chez Jojo", "Vertige des sens", "Le Crousse", "Le bouchon coréen", "Le lampad'hair"),
  style = c("tradi", "gastro", "tradi", "gastro", "coiffure"),
  ville = c("Ecully", "Lyon", "Lyon", "Lyon", "Ecully"),
  evaluation = c(4.6, 3.2, 3.3, 4.1, 1.2),
  places = c(28, 32, 94, 18, 8),
  note = c("Pas mal", "Cher", "Ambiance jeune", "Original", "Euh ?")
)

restos
#> # A tibble: 5 x 6
#>   nom       style     ville  evaluation places note
#>   <chr>     <chr>    <chr>      <dbl>   <dbl> <chr>
#> 1 Chez Jojo   tradi    Ecully      4.6     28 Pas mal
#> 2 Vertige des sens  gastro   Lyon        3.2     32 Cher
#> 3 Le Crousse   tradi    Lyon        3.3     94 Ambiance jeune
#> 4 Le bouchon coréen  gastro   Lyon        4.1     18 Original
#> 5 Le lampad'hair  coiffure Ecully      1.2      8 Euh ?
```

Avec `dplyr`, on peut sélectionner une colonne du tableau avec :

```
restos %>% select(note)
#> # A tibble: 5 x 1
#>   note
#>   <chr>
#> 1 Pas mal
#> 2 Cher
#> 3 Ambiance jeune
#> 4 Original
#> 5 Euh ?
```

Dans cette expression, `note` ne se réfère pas à un objet `note` de notre environnement, mais à une variable du tableau de données `restos`.

Que se passe-t-il si on passe à `select` non pas une colonne du tableau mais bien un objet de notre environnement ?

```
x <- "note"
restos %>% select(x)
#> # A tibble: 5 x 1
#>   note
#>   <chr>
#> 1 Pas mal
#> 2 Cher
#> 3 Ambiance jeune
#> 4 Original
#> 5 Euh ?
```

Ça fonctionne aussi : en l'absence de colonne nommée `x`, `select` va aller chercher le `x` de notre environnement et utiliser sa valeur pour sélectionner la colonne par nom.

¹Effectivement, un salon de coiffure s'est à nouveau glissé dans ce jeu de données.

Mais que se passe-t-il s'il existe à la fois une colonne du tableau et un objet de notre environnement du même nom ?

```
note <- "nom"
restos %>% select(note)
#> # A tibble: 5 x 1
#>   note
#>   <chr>
#> 1 Pas mal
#> 2 Cher
#> 3 Ambiance jeune
#> 4 Original
#> 5 Euh ?
```

Dans ce cas, c'est la colonne du tableau qui a la priorité. On dit que `dplyr` fait du *data masking* : les objets de notre environnement sont “masqués” par les colonnes du même nom de notre tableau de données. On retrouve ce *data masking* dans d'autres fonctions comme `filter()`, `mutate()` ou `summarise()`, mais aussi dans la sélection des variables avec `aes()` pour les fonctions de `ggplot2`.

Dans certains cas de figure, on peut vouloir outrepasser ce *data masking*. Par exemple, dans le cas suivant, la nouvelle colonne `note_michelin` n'est pas créée à partir des nouvelles données de l'objet `note`, mais à partir de celles de la colonne `note`.

```
note <- c(12, 14, 9, 15, NA)
restos %>% mutate(note_michelin = note)
#> # A tibble: 5 x 7
#>   nom          style    ville  evaluation places note      note_michelin
#>   <chr>        <chr>    <chr>     <dbl>    <dbl> <chr>      <chr>
#> 1 Chez Jojo    tradi    Ecully     4.6      28 Pas mal    Pas mal
#> 2 Vertige des sens  gastro  Lyon      3.2      32 Cher      Cher
#> 3 Le Crousse   tradi    Lyon      3.3      94 Ambiance jeu~ Ambiance jeu~
#> 4 Le bouchon coréen  gastro  Lyon      4.1      18 Original   Original
#> 5 Le lampad'hair coiffure Ecully     1.2      8 Euh ?      Euh ?
```

“Il suffit de changer le nom de l'objet `note` pour qu'il ne corresponde à aucune colonne !”, s'exclamera la lectrice ou le lecteur attentif.ve. Mais, outre que cela peut être source d'erreur, il est des cas où on ne connaît pas le nom des colonnes du tableau, par exemple quand l'opération se déroule dans une fonction et que le tableau est passé en paramètre :

```
filtre_nom <- function(df, valeurs) {
  df %>% filter(nom %in% valeurs)
}
```

Rien ne nous assure dans ce cas que le tableau `df` ne contient pas déjà une colonne nommée `valeurs` qui “masquerait” l'objet `valeurs` passé en argument...

Pour pallier à ce problème, à chaque fois qu'on est dans un environnement où du *data masking* se produit, on peut utiliser deux “pronoms” spécifiques nommés `.data` et `.env` :

- `.data$var` ou `.data[["var"]]` pointe vers l'objet `var` correspondant à une colonne du tableau de données
- `.env$var` ou `.env[["var"]]` pointe vers l'objet `var` correspondant à un objet de notre environnement

Avec ces deux outils, on peut donc explicitement choisir d'où viennent les données qu'on utilise.

```

note <- c(12, 14, 9, 15, NA)
restos %>% mutate(note_michelin = .env$note)
#> # A tibble: 5 x 7
#>   nom          style  ville  evaluation places note      note_michelin
#>   <chr>        <chr>  <chr>    <dbl>  <dbl> <chr>           <dbl>
#> 1 Chez Jojo    tradi   Ecully     4.6    28 Pas mal       12
#> 2 Vertige des sens  gastro  Lyon      3.2    32 Cher        14
#> 3 Le Crousse   tradi   Lyon      3.3    94 Ambiance je~       9
#> 4 Le bouchon coréen  gastro  Lyon      4.1    18 Original      15
#> 5 Le lampad'hair coiffure Ecully     1.2     8 Euh ?        NA
restos %>% mutate(note = str_to_upper(.data$note))
#> # A tibble: 5 x 6
#>   nom          style  ville  evaluation places note
#>   <chr>        <chr>  <chr>    <dbl>  <dbl> <chr>
#> 1 Chez Jojo    tradi   Ecully     4.6    28 PAS MAL
#> 2 Vertige des sens  gastro  Lyon      3.2    32 CHER
#> 3 Le Crousse   tradi   Lyon      3.3    94 AMBIANCE JEUNE
#> 4 Le bouchon coréen  gastro  Lyon      4.1    18 ORIGINAL
#> 5 Le lampad'hair coiffure Ecully     1.2     8 EUH ?

```

En utilisant `.env`, on peut donc s’assurer que notre fonction `filtre_nom()` ci-dessus va bien prendre les valeurs dans notre environnement, donc dans l’argument passé à la fonction, et pas dans une éventuelle colonne qui porterait le même nom.

```

filtre_nom <- function(df, valeurs) {
  df %>% filter(nom %in% .env$valeurs)
}

filtre_nom(restos, c("Chez Jojo", "Le Crousse"))
#> # A tibble: 2 x 6
#>   nom          style  ville  evaluation places note
#>   <chr>        <chr>  <chr>    <dbl>  <dbl> <chr>
#> 1 Chez Jojo    tradi   Ecully     4.6    28 Pas mal
#> 2 Le Crousse   tradi   Lyon      3.3    94 Ambiance jeune

```

19.1.2 *tidy selection*

Une autre spécificité de certaines fonctions du *tidyverse* réside dans le mode de sélection des colonnes basé sur un “mini-langage” permettant des expressions comme :

```

c(height, mass)
q1:q10
where(is.numeric) & !contains(id)

```

Cette méthode de sélection de colonnes est appelée *tidy selection* et on la retrouve dans plusieurs fonctions de `dplyr` et `tidyverse`, comme `select()`, `across()`, `c_across()`, `pull()`, `pivot_longer()`, etc.



La *tidy selection* est implémentée par le package `tidyselect` et on peut retrouver les différentes possibilités de ce mini-langage dans la vignette [Selection language](#).

On notera que la *tidy selection* fait appel au *data masking*, tout en y ajoutant des fonctions spécifiques.

19.1.3 Utilisation dans des fonctions

Une difficulté liée au *data masking* survient quand les colonnes du tableau ne sont pas saisies directement mais proviennent d'un argument de fonction.

Soit la fonction suivante qui prend en entrée un tableau de données et une colonne et retourne le résultat d'un `summarise`.

```
summarise_min <- function(df, col) {
  df %>% summarise(min = min(col))
}
```

On voudrait pouvoir appeler cette fonction de la même manière qu'on utilise `summarise`, mais cela ne fonctionne pas :

```
summarise_min(restos, evaluation)
#> Error: Problem with `summarise()` column `min`.
#> i `min = min(col)`.
#> x objet 'evaluation' introuvable
```

Le message d'erreur nous dit que l'objet `evaluation` est introuvable. La raison n'est pas triviale, elle repose sur plusieurs mécanismes assez complexes liés à l'évaluation des expressions dans R, mais on pourrait résumer en disant que les fonctions du *tidyverse* utilisent leur propre mécanisme d'évaluation qui tient compte notamment du *data masking*. Or, quand on utilise l'argument `col` dans le `summarise` de notre fonction, c'est l'évaluation “normale” de R qui est utilisée : le *data masking* n'étant pas pris en compte, l'objet `evaluation` est recherché dans notre environnement plutôt que dans notre tableau, ce qui génère une erreur puisqu'aucun objet de ce nom n'existe en-dehors du tableau.

Comme la *tidy selection* fait appel au *data masking*, elle génère le même type d'erreur :

```
select_col <- function(df, col) {
  df %>% select(col)
}

select_col(restos, evaluation)
#> Error: objet 'evaluation' introuvable
```

C'est pourquoi les packages du *tidyverse* fournissent un opérateur permettant de “forcer” l'évaluation d'expressions selon la manière qu'elles attendent. Cet opérateur prend la forme de double accolades `{}{ }` et se nomme *curly curly*.

Pour résoudre le problème de notre fonction `summarise_min()`, on peut donc simplement faire passer notre argument `col` dans l'opérateur *curly curly* :

```
summarise_min <- function(df, col) {
  df %>% summarise(min = min({{ col }}))
}

summarise_min(restos, evaluation)
#> # A tibble: 1 x 1
#>       min
#>   <dbl>
#> 1     1.2
```

À noter que *curly curly* permet de passer en argument toute expression qui serait acceptée directement par les fonctions appelées. On peut donc combiner plusieurs colonnes, effectuer des opérations, etc.

```
summarise_min(restos, evaluation * 4)
#> # A tibble: 1 x 1
#>   min
#>   <dbl>
#> 1 4.8
```

Et on peut même utiliser les pronoms `.data` et `.env` :

```
evaluation <- 0:5
summarise_min(restos, .env$evaluation)
#> # A tibble: 1 x 1
#>   min
#>   <int>
#> 1 0
```

19.2 Programmer avec dplyr et tidyr

19.2.1 Utiliser une colonne passée en argument

Une opération courante quand on utilise les fonctions de `dplyr` ou `tidyr` dans une fonction est de prendre en argument une colonne à laquelle on souhaite accéder. Dans ce cas on doit utiliser l'opérateur *curly curly* et entourer les utilisations de l'argument contenant la colonne par une paire d'accolades.

On a déjà vu un exemple précédemment avec `summarise()`.

```
resume <- function(df, col) {
  df %>% summarise(
    moyenne = mean({{ col }}),
    min = min({{ col }}),
    max = max({{ col }})
  )
}

resume(restos, evaluation)
#> # A tibble: 1 x 3
#>   moyenne   min   max
#>   <dbl> <dbl> <dbl>
#> 1 3.28   1.2   4.6
```

C'est le cas dans toutes les fonctions qui font du *data masking*, comme `group_by()` :

```
resume_groupe <- function(df, col_group, col_var) {
  df %>%
    group_by({{ col_group }}) %>%
    summarise(
      moyenne = mean({{ col_var }}),
      min = min({{ col_var }}),
      max = max({{ col_var }})
```

```

        )
}

resume_groupe(restos, style, evaluation)
#> # A tibble: 3 x 4
#>   style     moyenne    min    max
#>   <chr>      <dbl> <dbl> <dbl>
#> 1 coiffure    1.2    1.2    1.2
#> 2 gastro     3.65   3.2    4.1
#> 3 tradi      3.95   3.3    4.6

```

19.2.2 Utiliser une sélection de colonnes passée en argument

Dans la section précédente, on a utilisé à chaque fois une seule colonne. Si on souhaite grouper ou appliquer une fonction sur une série de colonnes, il faut alors utiliser `across()`.

```

resume_groupe <- function(df, cols_group, cols_var) {
  df %>%
    group_by(
      across({{ cols_group }}))
  ) %>%
    summarise(
      across(
        {{ cols_var }},
        mean
      )
    )
}

```

On peut du coup utiliser tous les modes de sélection de colonnes permises par la *tidy selection*.

```

resume_groupe(restos, c(style, ville), where(is.numeric))
#> `summarise()` has grouped output by 'style'. You can override using the `~.groups` argument.
#> # A tibble: 4 x 4
#> # Groups:   style [3]
#>   style     ville  evaluation places
#>   <chr>     <chr>      <dbl>   <dbl>
#> 1 coiffure  Ecully     1.2       8
#> 2 gastro    Lyon      3.65     25
#> 3 tradi     Ecully     4.6      28
#> 4 tradi     Lyon      3.3      94

```

De la même manière, si on utilise un argument de fonction pour sélectionner des variables avec `select()`, on doit l'entourer avec l'opérateur *curly curly*, et on peut dès lors utiliser toutes les possibilités de la *tidy selection*.

```

select_cols <- function(df, cols) {
  df %>% select({{ cols }})
}

restos %>% select_cols(where(is.character) & !c(nom, note))
#> # A tibble: 5 x 2
#>   style     ville

```

```
#>   <chr>    <chr>
#> 1 tradi    Ecully
#> 2 gastro   Lyon
#> 3 tradi    Lyon
#> 4 gastro   Lyon
#> 5 coiffure Ecully
```

19.2.3 Nommer de nouvelles colonnes à partir d'un argument

On a vu comment utiliser des colonnes passées en argument pour accéder à leur contenu. On peut aussi vouloir passer en argument des noms de colonnes qu'on souhaite créer, par exemple avec un `mutate()` ou un `summarise()`.

On pourrait essayer directement de la manière suivante, mais cela ne fonctionne pas car dans ce cas la colonne créée s'appelle “`col_new`”, et pas la valeur de l'argument `col_new`.

```
calcule_pourcentage <- function(df, col_new, col_var) {
  df %>%
    mutate(
      col_new = {{ col_var }} / sum({{ col_var }}) * 100
    )
}

calcule_pourcentage(restos, prop_places, places)
#> # A tibble: 5 x 7
#>   nom          style  ville  evaluation places note      col_new
#>   <chr>        <chr>  <chr>     <dbl>   <dbl> <chr>      <dbl>
#> 1 Chez Jojo   tradi   Ecully     4.6     28 Pas mal    15.6
#> 2 Vertige des sens  gastro  Lyon      3.2     32 Cher     17.8
#> 3 Le Crousse   tradi   Lyon      3.3     94 Ambiance jeune  52.2
#> 4 Le bouchon coréen  gastro  Lyon      4.1     18 Original   10
#> 5 Le lampad'hair coiffure Ecully     1.2      8 Euh ?     4.44
```

Dans ce cas de figure, la syntaxe à utiliser est un peu plus complexe :

- on remplace l'opérateur `=` du `mutate()` par l'opérateur `:=` (appelé *walrus operator*)
- on place à gauche du `:=` une chaîne de caractères dans laquelle notre argument contenant le nom de la nouvelle variable est entouré d'une paire d'accolades.

Voici ce que ça donne pour l'exemple ci-dessus :

```
calcule_pourcentage <- function(df, col_new, col_var) {
  df %>%
    mutate(
      "{{col_new}}" := {{ col_var }} / sum({{ col_var }}) * 100
    )
}

calcule_pourcentage(restos, prop_places, places)
#> # A tibble: 5 x 7
#>   nom          style  ville  evaluation places note      prop_places
#>   <chr>        <chr>  <chr>     <dbl>   <dbl> <chr>      <dbl>
#> 1 Chez Jojo   tradi   Ecully     4.6     28 Pas mal    15.6
#> 2 Vertige des sens  gastro  Lyon      3.2     32 Cher     17.8
```

```
#> 3 Le Crousse      tradi    Lyon      3.3    94 Ambiance jeune    52.2
#> 4 Le bouchon coréen gastro  Lyon      4.1    18 Original        10
#> 5 Le lampad'hair  coiffure Ecully   1.2    8 Euh ?           4.44
```

Cette syntaxe est un peu complexe de prime abord, mais elle à l'avantage d'être souple : en particulier, on peut placer le texte que l'on souhaite dans la chaîne de caractères en plus des noms de variables entre double accolades.

Cela permet par exemple de générer le nom d'une nouvelle variable automatiquement à partir de l'ancienne.

```
calcule_pourcentage <- function(df, col_var) {
  df %>%
    mutate(
      "prop_{col_var}" := {{ col_var }} / sum({{ col_var }}) * 100
    )
}

calcule_pourcentage(restos, places)
#> # A tibble: 5 x 7
#>   nom          style  ville  evaluation places note      prop_places
#>   <chr>        <chr>  <chr>     <dbl>    <dbl> <chr>        <dbl>
#> 1 Chez Jojo    tradi   Ecully     4.6     28 Pas mal    15.6
#> 2 Vertige des sens gastro  Lyon      3.2     32 Cher     17.8
#> 3 Le Crousse   tradi   Lyon      3.3     94 Ambiance jeune 52.2
#> 4 Le bouchon coréen gastro  Lyon      4.1     18 Original   10
#> 5 Le lampad'hair coiffure Ecully   1.2     8 Euh ?      4.44
```

Ou de personnaliser les noms de colonnes dans un `summarise()`.

```
resume <- function(df, col) {
  df %>% summarise(
    "{{col}}_moyenne" := mean({{ col }}),
    "{{col}}_min" := min({{ col }}),
    "{{col}}_max" := max({{ col }})
  )
}

resume(restos, places)
#> # A tibble: 1 x 3
#>   places_moyenne places_min places_max
#>       <dbl>        <dbl>        <dbl>
#> 1            36           8           94
```

19.2.4 Désambiguïser `data` et `env`

Lorsqu'on utilise des fonctions de `dplyr` ou `tidyverse` dans d'autres fonctions, il peut être utile de préciser, quand on accède à un objet dont on connaît le nom (c'est-à-dire dont le nom n'est pas passé en argument), si c'est un objet de type “data” (une colonne du tableau de données dans lequel on travaille) ou de type “env” (un objet de l'environnement dans lequel on travaille).

Dans la fonction suivante, on calcule la moyenne d'une colonne numérique de `restos` selon les valeurs de la colonne `ville`, et on ne conserve que les villes pour lesquelles cette moyenne est supérieure à un certain seuil.

```
stat_par_ville <- function(col_var, seuil = NULL) {
  res <- restos %>%
    group_by(.data$ville) %>%
    summarise(moyenne = mean({{col_var}})) %>%
    filter(moyenne > .env$seuil)
}

stat_par_ville(evaluation, seuil = 0)
stat_par_ville(evaluation, seuil = 3)
```

On veut que l'objet `ville` du `group_by` soit toujours la colonne du tableau `restos` nommée `ville`, on peut donc s'assurer que c'est bien le cas en l'explicitant avec `.data$ville`. À l'inverse, on veut que la valeur `seuil` du `filter` soit celle de l'argument du même nom, donc d'un objet de l'environnement. On peut s'en assurer en indiquant `.env$seuil`.

Cette explicitation ne paraît pas forcément utile à première vue, mais elle peut éviter des problèmes à terme, notamment si on ajoute de nouvelles colonnes à un tableau de données et qu'on finit par avoir des objets “data” et des objets “env” avec le même nom.

19.2.5 Quand les arguments sont des chaînes de caractères

Jusqu'ici, on a passé les arguments de fonction sous la forme d'expressions ou de symboles.

```
resume(restos, places)
summarise_min(restos, evaluation * 4)
```

Mais il arrive que des noms de colonnes soient passés plutôt sous forme de chaînes de caractères.

```
resume(restos, "places")
summarise_min(restos, "evaluation")
```

On ne peut pas dans ce cas utiliser l'opérateur *curly curly*, par contre on peut utiliser le pronom `.data` pour accéder aux colonnes à partir de leur nom.

```
summarise_min <- function(df, col) {
  df %>% summarise(min = min(.data[[col]]))
}

summarise_min(restos, "evaluation")
#> # A tibble: 1 x 1
#>       min
#>   <dbl>
#> 1 1.2
```

Si dans l'exemple précédent on souhaite personnaliser le nom de la colonne créée en utilisant la valeur de notre paramètre `evaluation`, on place le nom de notre objet dans une chaîne de caractère en l'entourant d'accolades simples, et on utilise le *walrus operator* `:=`.

```
summarise_min <- function(df, col) {
  df %>% summarise("min_{col}" := min(.data[[col]]))
}

summarise_min(restos, "evaluation")
#> # A tibble: 1 x 1
#>   min_evaluation
#>   <dbl>
#> 1 1.2
```

Quand on veut plutôt sélectionner des colonnes avec `select()` ou `across()` et qu'on récupère les noms de ces colonnes dans un vecteur de chaînes de caractères, on doit utiliser les fonctions `all_of()` ou `any_of()`.

```
evaluation_par_groupe <- function(cols_group) {
  restos %>%
    group_by(
      across(all_of(cols_group)))
  ) %>%
    summarise(evaluation = mean(.data$evaluation))
}

evaluation_par_groupe("ville")
#> # A tibble: 2 x 2
#>   ville evaluation
#>   <chr>     <dbl>
#> 1 Ecully     2.9
#> 2 Lyon       3.53

evaluation_par_groupe(c("ville", "style"))
#> `summarise()` has grouped output by 'ville'. You can override using the `groups` argument.
#> # A tibble: 4 x 3
#> # Groups:   ville [2]
#>   ville style   evaluation
#>   <chr> <chr>     <dbl>
#> 1 Ecully coiffure     1.2
#> 2 Ecully tradi       4.6
#> 3 Lyon   gastro      3.65
#> 4 Lyon   tradi       3.3
```

La différence entre `all_of()` et `any_of()` est que `all_of()` produira une erreur si l'une des variables n'est pas trouvée.

```
select_all_cols <- function(cols) {
  restos %>% select(all_of(cols))
}

select_all_cols(c("ville", "evaluation", "igloo"))
#> Error: Can't subset columns that don't exist.
#> #> x Column `igloo` doesn't exist.
```

Tandis qu'`any_of()` renverra uniquement les colonnes existantes, sans générer d'erreur.

```
select_any_cols <- function(cols) {
  restos %>% select(any_of(cols))
}

select_any_cols(c("ville", "evaluation", "igloo"))
#> # A tibble: 5 x 2
#>   ville     evaluation
#>   <chr>      <dbl>
#> 1 Ecully     4.6
#> 2 Lyon       3.2
#> 3 Lyon       3.3
#> 4 Lyon       4.1
#> 5 Ecully     1.2
```

19.3 Programmer avec ggplot2

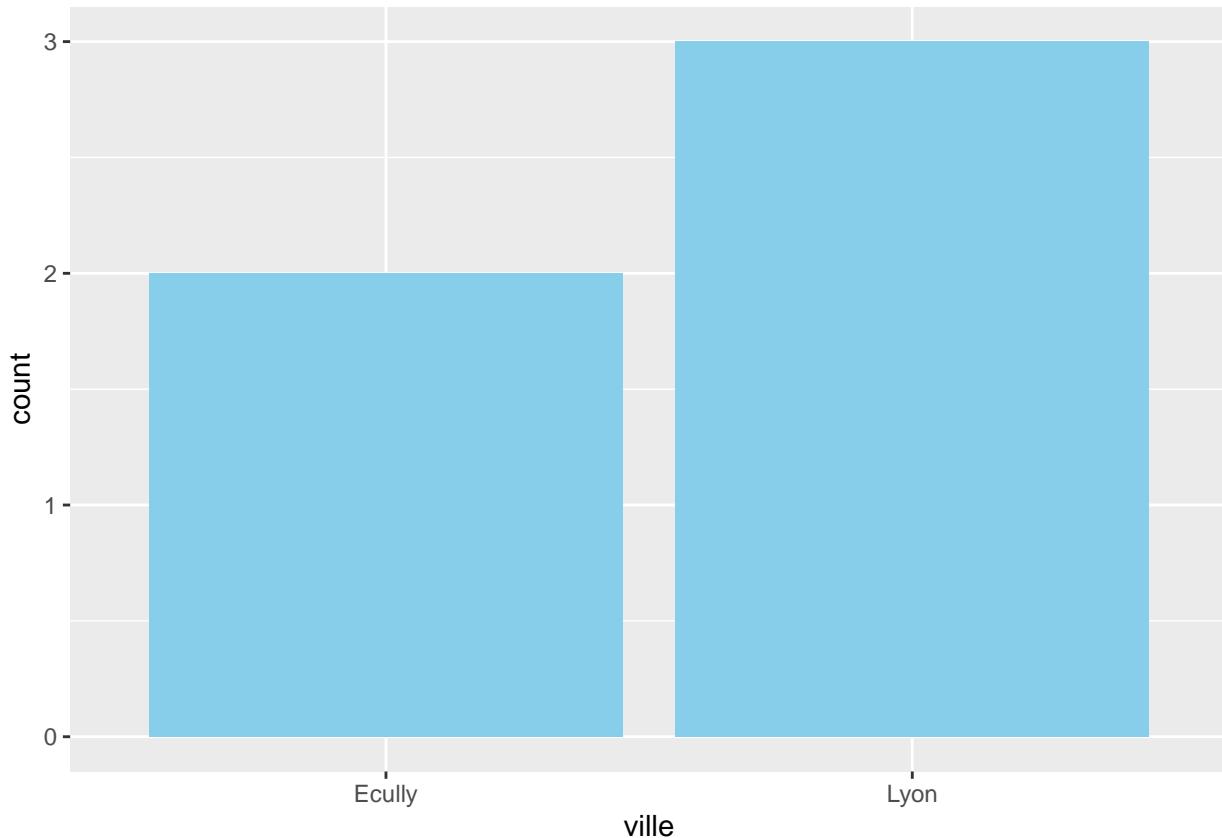
19.3.1 Sélection de colonnes avec aes()

Les règles pour sélectionner des colonnes dans un graphique ggplot2 à partir d'arguments passés à une fonction sont les mêmes que celles vues précédemment pour `dplyr` et `tidyverse` :

- si les noms sont passés sous forme de symboles ou d'expressions, on utilise l'opérateur *curly curly* (`{`{`}`}`)

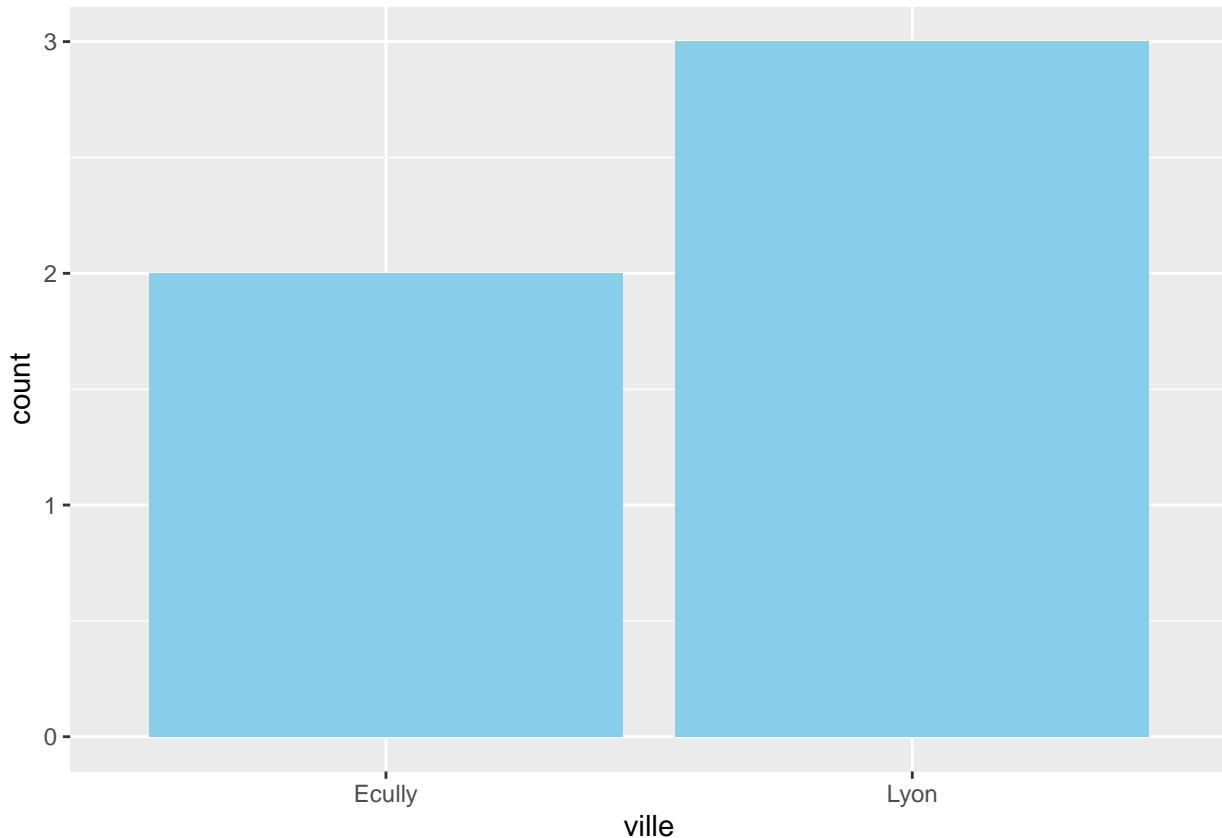
```
bar_graph <- function(df, col) {
  ggplot(df) +
    geom_bar(aes(x = {{ col }}), fill = "skyblue")
}

bar_graph(restos, ville)
```



- si les noms sont passés sous forme de chaînes de caractères, on utilise le pronom `.data`

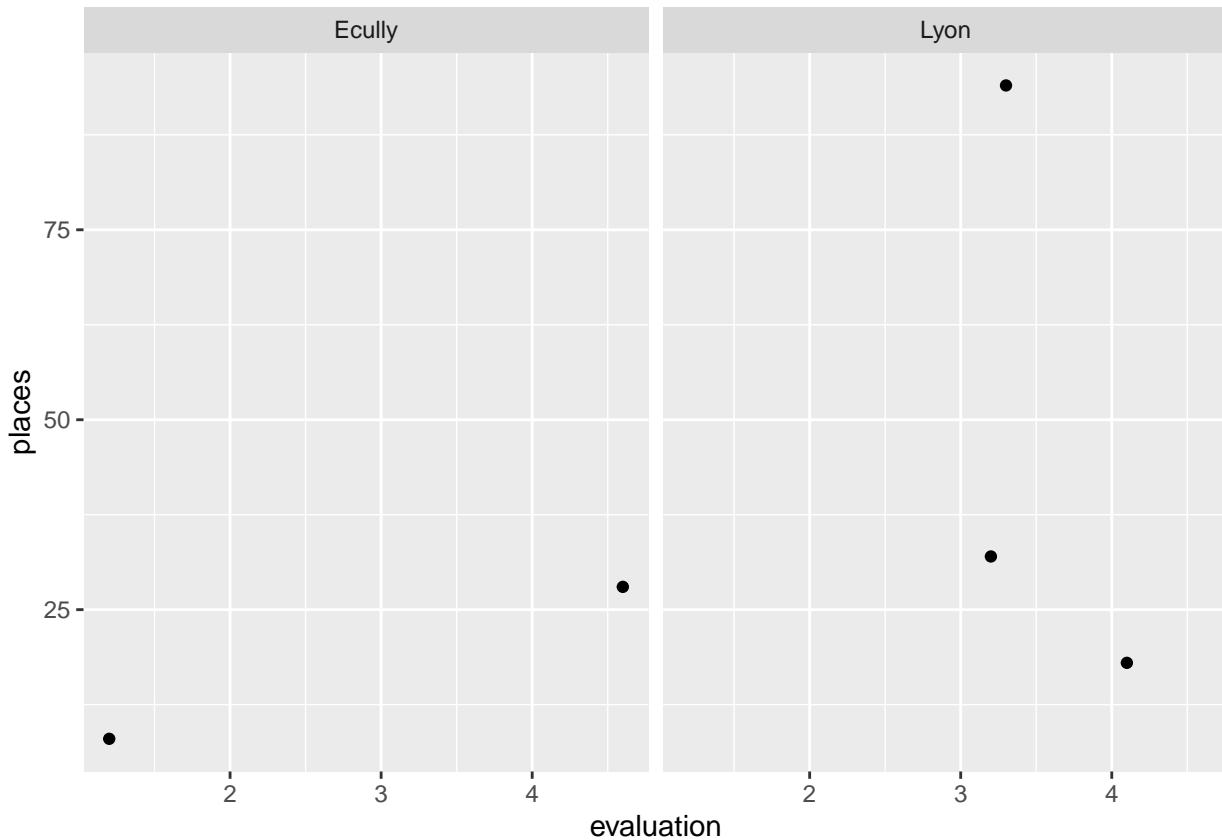
```
bar_graph <- function(df, col) {  
  ggplot(df) +  
    geom_bar(aes(x = .data[[col]]), fill = "skyblue")  
}  
  
bar_graph(restos, "ville")
```



19.3.2 Faceting

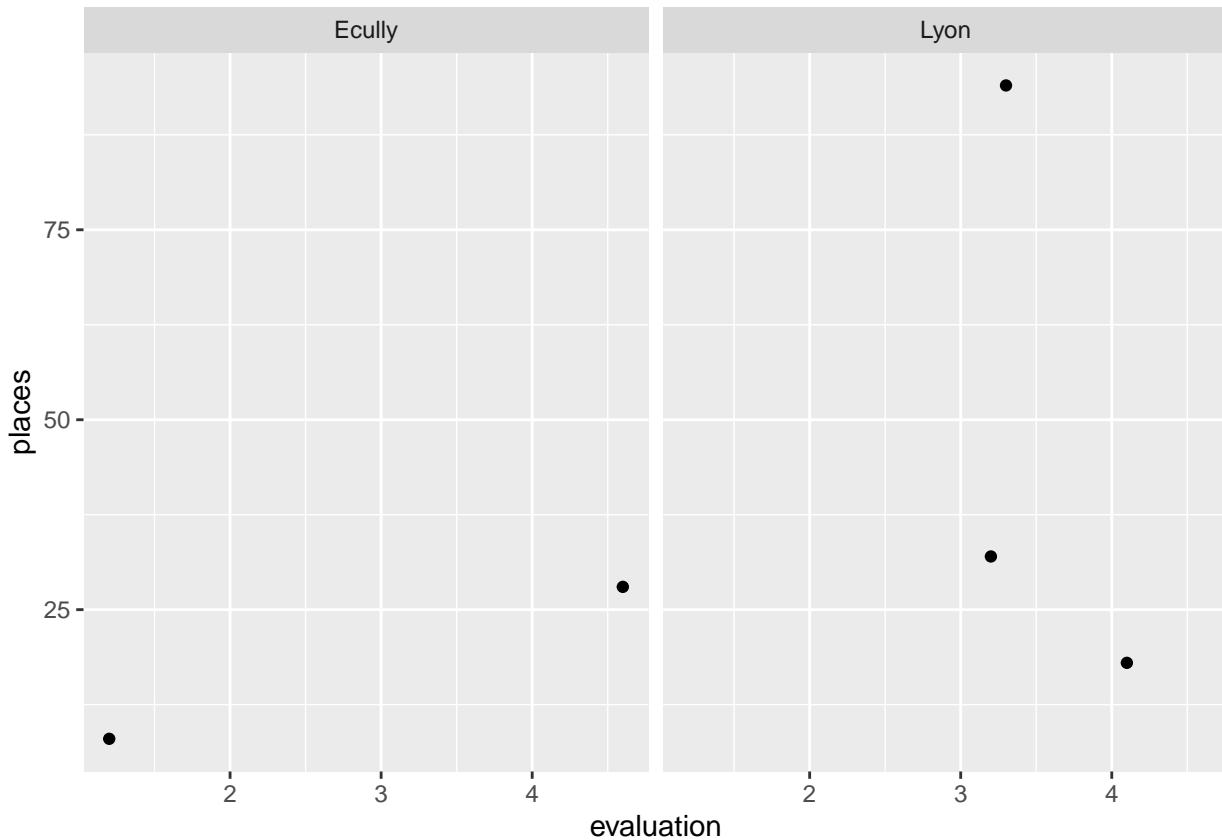
Quand on utilise `facet_wrap()` ou `facet_grid()`, si la variable de faceting est donnée sous forme d'un symbole, on utilise `vars()` en conjonction avec l'opérateur *curly curly*.

```
facet_points <- function(facet) {  
  ggplot(restos) +  
    geom_point(aes(x = .data$evaluation, y = .data$places)) +  
    facet_wrap(vars({{ facet }}))  
}  
  
facet_points(ville)
```



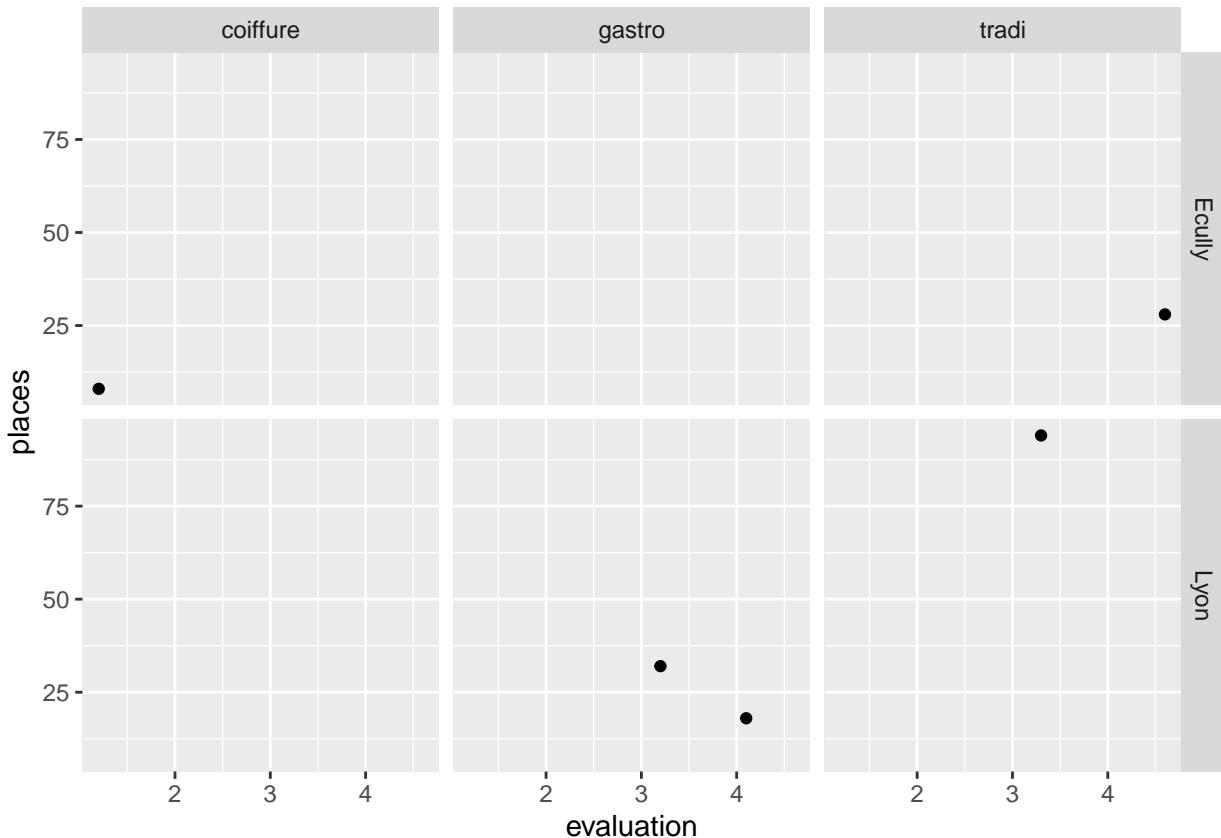
Si la variable de facetting est donnée sous forme d'une chaîne de caractères , on utilise `vars()` en conjonction avec le pronom `.data`.

```
facet_points <- function(facet) {  
  ggplot(restos) +  
    geom_point(aes(x = .data$evaluation, y = .data$places)) +  
    facet_wrap(vars(.data[[facet]]))  
}  
  
facet_points("ville")
```



Si on utilise `facet_grid()`, la fonction doit accepter deux variables de facetting comme arguments. Celles-ci peuvent ensuite être passées à `facet_grid()` via ses paramètres `rows` et `cols`.

```
facet_points <- function(facet_row, facet_col) {  
  ggplot(restos) +  
    geom_point(aes(x = .data$evaluation, y = .data$places)) +  
    facet_grid(  
      rows = vars({{ facet_row }}),  
      cols = vars({{ facet_col }}))  
}  
  
facet_points(ville, style)
```



19.4 Aide-mémoire

On essaie de récapituler ici les points importants pour pouvoir les retrouver facilement.



Premier point important : les spécificités vues ici ne s'appliquent que quand on veut utiliser certaines fonctions du *tidyverse* (`dplyr`, `tidyr`, `ggplot2`) à l'intérieur d'autres fonctions. Plus spécifiquement, elles sont à prendre en compte quand on souhaite passer en argument d'une fonction des noms de colonnes qui seront utilisées par des fonctions du *tidyverse*.

Elles ne s'appliquent pas si on passe en arguments d'autres paramètres comme le tableau de données qu'on souhaite utiliser, des valeurs numériques ou des chaînes de caractères qu'on souhaite récupérer telles quelles, etc.

1. Dans le cas où deux objets du même nom pourraient exister à la fois comme colonne de notre tableau de données (objet `data`) et comme objet de notre environnement (objet `env`), on peut expliciter lequel on souhaite utiliser avec les pronoms `.data$var` et `.env$var`
2. Si un argument est une colonne passée sous la forme d'un symbole (`var`), on doit l'encadrer de l'opérateur *curly curly*.

```
summarise_col <- function(df, col) {
  df %>% summarise(moyenne = mean({{ col }}), na.rm = TRUE)
}

summarise_col(starwars, height)
```

3. Si un argument est un nom de colonne passé sous la forme d'une chaîne de caractères ("var"), on y accède avec le pronom `.data` :

```
summarise_col <- function(df, col_name) {
  df %>% summarise(moyenne = mean(.data[[col_name]], na.rm = TRUE))
}

summarise_col(starwars, "height")
```

4. Si on utilise la *tidy selection* dans un `select()`, un `across()` ou une autre fonction, on l'encadre de l'opérateur *curly curly* :

```
select_cols <- function(df, cols) {
  df %>% select({{ cols }})
}

select_cols(starwars, !where(is.list))
```

5. Si on indique les noms de plusieurs colonnes sous la forme d'un vecteur de chaînes de caractères pour utilisation dans un `select()`, un `across()` ou une autre fonction acceptant la *tidy selection*, on utilise `all_of()` ou `any_of()` :

```
select_cols <- function(df, col_names) {
  df %>% select(all_of(col_names))
}

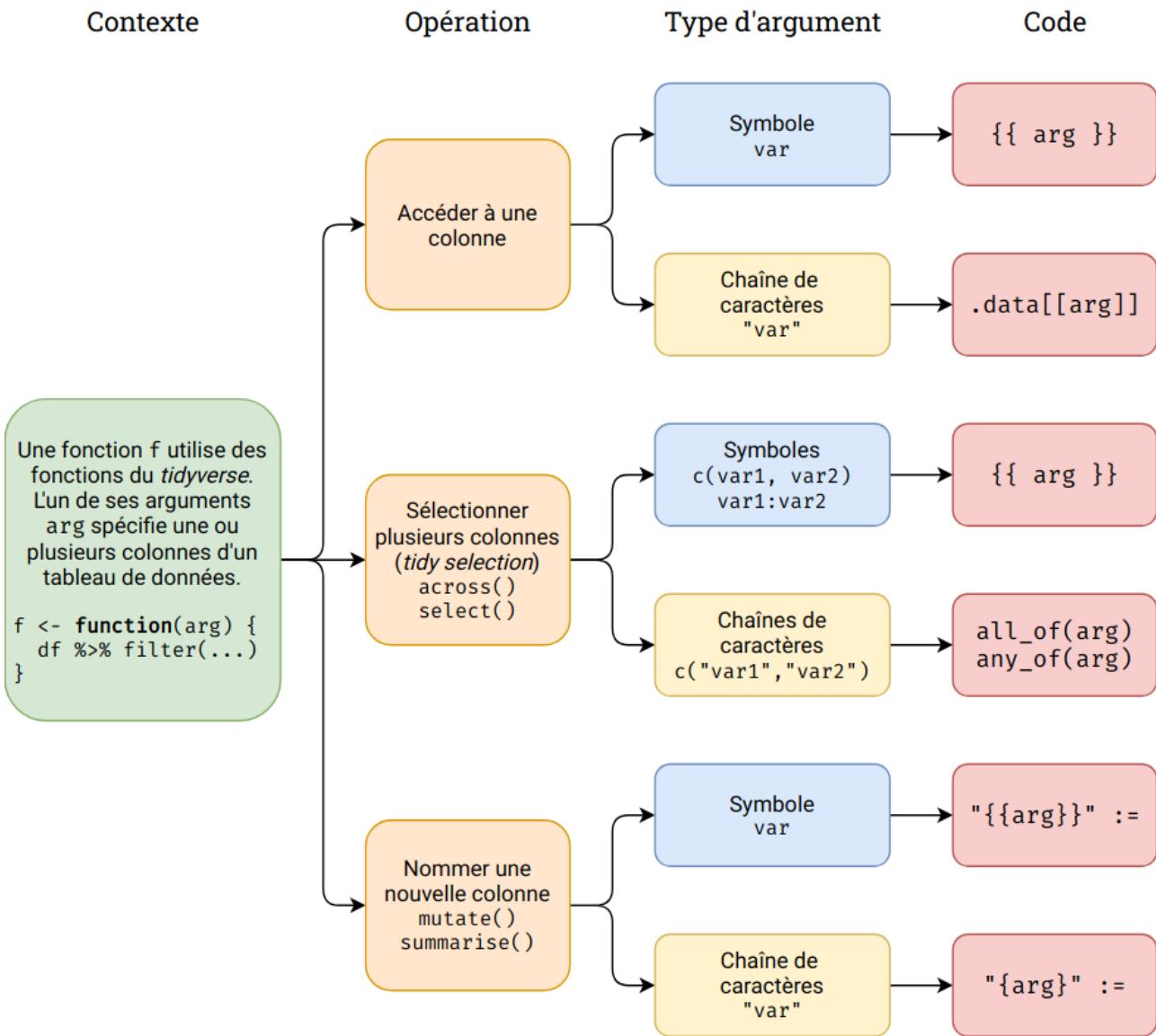
select_cols(starwars, c("height", "mass"))
```

6. Si on souhaite créer une nouvelle colonne à partir de la valeur d'un argument, on l'utilise sous la forme d'une chaîne de caractères avec l'opérateur *walrus* `:=`. Si l'argument est un symbole on l'entoure avec `{`{`}`, si c'est une chaîne de caractères on l'entoure avec `{`}`` :

```
add_mean_by_species <- function(col_var) {
  starwars %>%
    group_by(species) %>%
    mutate("moyenne_{{col_var}}" := mean('{{ col_var }}'))
}

add_mean_by_species(height)
```

Le schéma suivant récapitule les points précédents :



19.5 Ressources

Pour plus de détails sur la programmation avec les fonctions de `dplyr`, on pourra se reporter à la vignette [Programming with dplyr](#).

Pour l'utilisation de `ggplot2` dans des fonctions, on trouvera plus de détails dans la vignette [Using ggplot2 in packages](#).

Les mécanismes d'évaluation et de métaprogrammation propres aux packages du *tidyverse* sont implémentés en grande partie dans le package `rlang`.

Enfin, pour une présentation approfondie des possibilités de métaprogrammation dans R en général, on pourra consulter le chapitre [Metaprogramming](#) de l'ouvrage en ligne *Advanced R*.

19.6 Exercices

19.6.1 dplyr et tidyr

Exercice 1.1

Créer une fonction `my_table` qui prend en arguments un tableau de données `df` et une variable `var` et qui renvoie le résultat de `count()` sur cette variable.

Vérifier avec :

```
my_table(starwars, gender)
#> # A tibble: 3 x 2
#>   gender      n
#>   <chr>     <int>
#> 1 feminine    17
#> 2 masculine   66
#> 3 <NA>        4
```

Modifier `my_table` pour qu'elle accepte trois arguments `df`, `var1` et `var2`, et qu'elle retourne le résultat du `count` appliqué à `var1` et à `var2`.

```
my_table(starwars, gender, sex)
#> # A tibble: 6 x 3
#>   gender   sex      n
#>   <chr>   <chr>   <int>
#> 1 feminine female    16
#> 2 feminine none      1
#> 3 masculine hermaphroditic  1
#> 4 masculine male     60
#> 5 masculine none      5
#> 6 <NA>     <NA>     4
```

Modifier `my_table` pour qu'elle accepte deux arguments `df` et `vars`, et qu'elle applique le `count` à toutes les variables indiquées dans `vars` en utilisant la *tidy selection*.

```
my_table(starwars, c(gender, sex))
#> # A tibble: 6 x 3
#>   gender   sex      n
#>   <chr>   <chr>   <int>
#> 1 feminine female    16
#> 2 feminine none      1
#> 3 masculine hermaphroditic  1
#> 4 masculine male     60
#> 5 masculine none      5
#> 6 <NA>     <NA>     4
```

Exercice 1.2

Le code suivant ajoute une nouvelle colonne `diff_height` au tableau `starwars`, qui contient la différence entre la valeur de `height` et sa moyenne selon les valeurs de `species`.

```
starwars %>%
  group_by(species) %>%
  mutate(diff_height = height - mean(height, na.rm = TRUE))
```

En utilisant ce code, créer une fonction `diff_mean_height` qui accepte un seul argument `by` et qui ajoute à `starwars` une colonne `diff_height` calculée de la même manière, mais en appliquant le `group_by` au `by` passé en paramètre.

Modifier la fonction précédente en une nouvelle fonction `diff_mean`, qui prend deux arguments `by` et `var`, et qui applique la même transformation non pas à `height` mais à la variable passée dans `var`.

Modifier la fonction pour que le nom de la colonne ajoutée ne soit pas `diff_mean` mais le nom de la variable passée dans `var` suivi du suffixe “`_diff_mean`”.

Modifier à nouveau `diff_mean` pour pouvoir passer à `by` plusieurs variables en utilisant la *tidy selection*.

Exercice 1.3

La fonction `unnest()` de `tidyverse` permet de transformer une colonne contenant des valeurs de type liste en colonne “normale” en dupliquant les lignes autant de fois qu'il y a d'éléments dans chaque liste :

```
starwars %>%
  unnest(films) %>%
  select(name, films)
#> # A tibble: 173 x 2
#>   name           films
#>   <chr>          <chr>
#> 1 Luke Skywalker The Empire Strikes Back
#> 2 Luke Skywalker Revenge of the Sith
#> 3 Luke Skywalker Return of the Jedi
#> 4 Luke Skywalker A New Hope
#> 5 Luke Skywalker The Force Awakens
#> 6 C-3PO           The Empire Strikes Back
#> 7 C-3PO           Attack of the Clones
#> 8 C-3PO           The Phantom Menace
#> 9 C-3PO           Revenge of the Sith
#> 10 C-3PO          Return of the Jedi
#> # ... with 163 more rows
```

Créer une fonction `freq_liste()` qui prend en entrée un argument `var` et retourne grâce à `count` le tri à plat de toutes les valeurs de la variable correspondant à `var` dans `starwars`, même si cette variable contient des listes.

```
freq_liste(films)
#> # A tibble: 7 x 2
#>   films            n
#>   <chr>           <int>
#> 1 A New Hope      18
#> 2 Attack of the Clones 40
#> 3 Return of the Jedi 20
#> 4 Revenge of the Sith 34
#> 5 The Empire Strikes Back 16
#> 6 The Force Awakens    11
#> 7 The Phantom Menace   34
```

19.6.2 Noms de colonnes en chaînes de caractères

Exercice 2.1

La fonction `readline()` permet de lire une chaîne de caractères entrée au clavier par l'utilisateur :

```
v <- readline("Votre choix : ")
```

Créer une fonction `affiche_planete` qui affiche le message “Nom de la planète”, lit la réponse de l'utilisateur, et affiche à l'aide d'un `filter()` les lignes du tableau `starwars` pour lesquelles la variable `homeworld` correspond au nom saisi.

Créer une fonction `affiche_barplot` qui affiche le message “Nom de la variable”, lit la réponse de l’utilisateur, et affiche avec `ggplot2` le diagramme en barres de la variable correspondante du tableau `starwars`.

Créer une fonction `affiche_colonnes` qui affiche le message “Noms des variables séparés par des virgules”, lit la réponse de l’utilisateur, et affiche uniquement les colonnes saisies du tableau `starwars`.

Conseil : pour récupérer un vecteur de colonnes à partir de la saisie de l’utilisateur, on pourra utiliser les fonctions `str_split()` et `str_trim()` de `stringr`.

19.6.3 Ambiguïté `data` / `env`

Exercice 3.1

Créer une fonction `filter_height()` qui accepte un argument nommé `height`, et qui retourne les lignes de `starwars` pour lesquelles les valeurs de la variable `height` sont supérieures à celles de l’argument `height`. Créer cette fonction sans utiliser `dplyr` mais avec l’opérateur `[,]`.

```
filter_height(200)
#> # A tibble: 16 x 14
#>   name    height mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>       <chr>      <chr>        <dbl> <chr> <chr>
#> 1 Darth ~    202   136 none       white      yellow       41.9 male  masculin
#> 2 Chewba~    228   112 brown     unknown     blue        200 male  masculin
#> 3 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> 4 Roos T~   224    82 none      grey       orange      NA male  masculin
#> 5 Rugor ~   206    NA none      green      orange      NA male  masculin
#> 6 Yarael~   264    NA none      white      yellow      NA male  masculin
#> 7 Lama Su   229    88 none      grey       black       NA male  masculin
#> 8 Taun We   213    NA none      grey       black       NA female feminin
#> 9 Grievo~   216   159 none      brown, whi~ green, y~ NA male  masculin
#> 10 Tarfful  234   136 brown     brown      blue        NA male  masculin
#> 11 Tion M~  206    80 none      grey       black       NA male  masculin
#> 12 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> 13 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> 14 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> 15 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> 16 <NA>       NA    NA <NA>       <NA>       <NA>        NA <NA> <NA>
#> # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

Améliorer la fonction pour qu’elle ne retourne pas les lignes pour lesquelles la variable correspondant à `height_var` vaut `NA`.

```
filter_height(200)
#> # A tibble: 10 x 14
#>   name    height mass hair_color skin_color eye_color birth_year sex   gender
#>   <chr>     <int> <dbl> <chr>       <chr>      <chr>        <dbl> <chr> <chr>
#> 1 Darth ~    202   136 none       white      yellow       41.9 male  masculin
#> 2 Chewba~    228   112 brown     unknown     blue        200 male  masculin
#> 3 Roos T~   224    82 none      grey       orange      NA male  masculin
#> 4 Rugor ~   206    NA none      green      orange      NA male  masculin
#> 5 Yarael~   264    NA none      white      yellow      NA male  masculin
#> 6 Lama Su   229    88 none      grey       black       NA male  masculin
#> 7 Taun We   213    NA none      grey       black       NA female feminin
#> 8 Grievo~   216   159 none      brown, whi~ green, y~ NA male  masculin
#> 9 Tarfful  234   136 brown     brown      blue        NA male  masculin
#> 10 Tion M~  206    80 none      grey       black      NA male  masculin
```

```
#> # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
#> #   vehicles <list>, starships <list>
```

Écrire la même fonction, cette fois en utilisant `filter()` à la place de `[,]`.

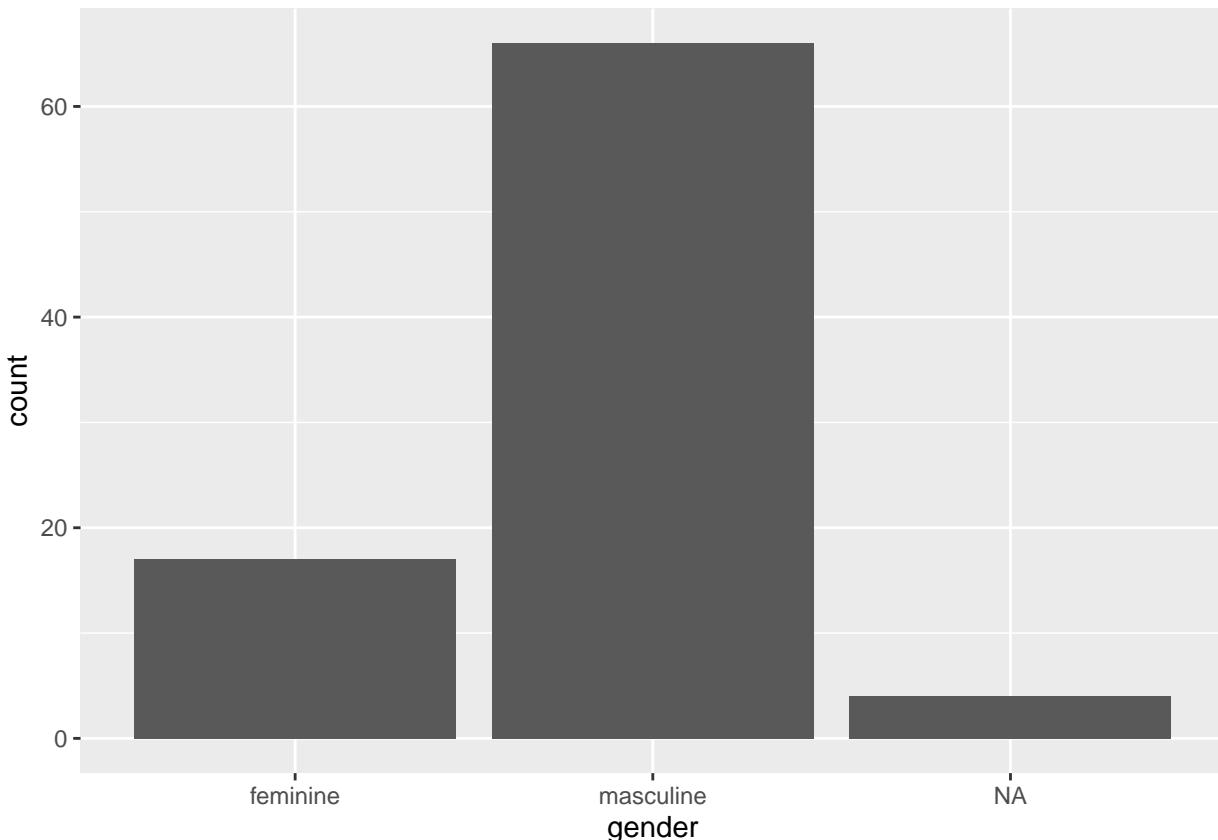
Quels sont les avantages et inconvénients des deux méthodes ?

19.6.4 ggplot2

Exercice 4.1

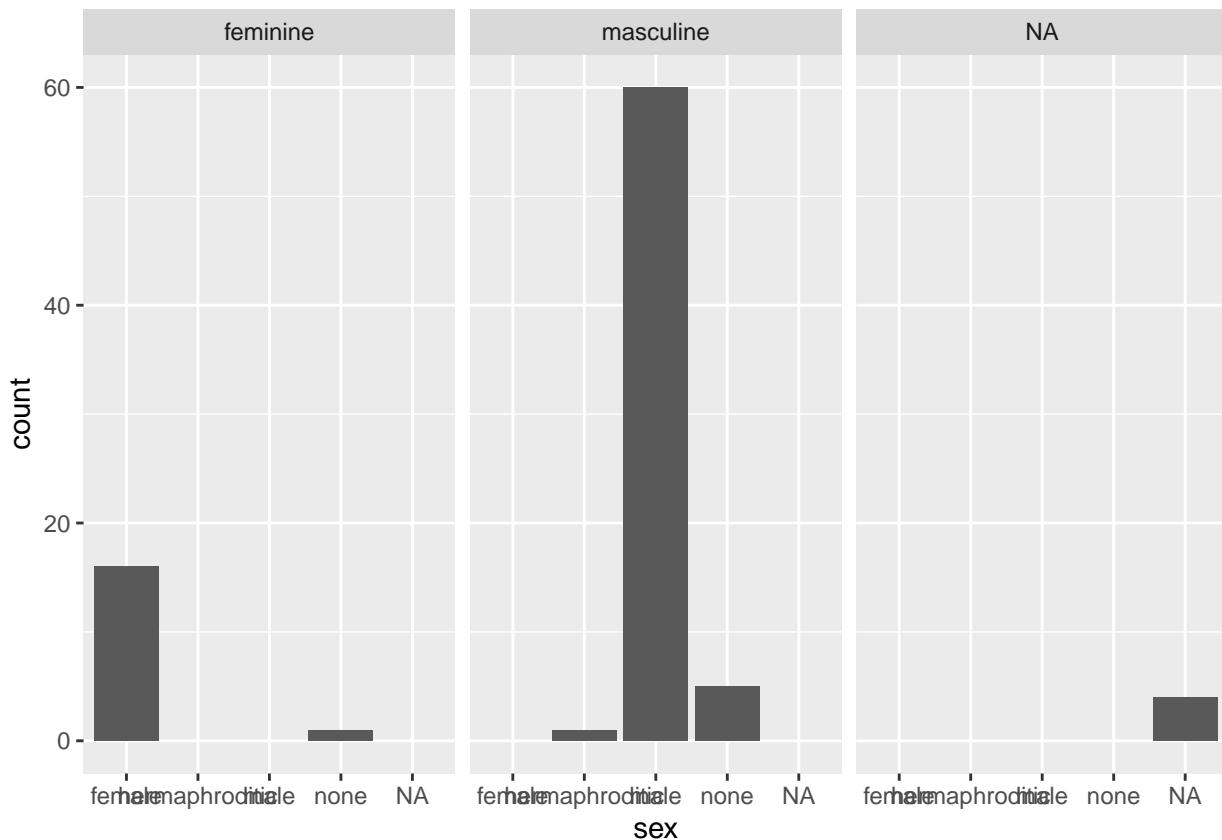
Créer une fonction `graph_bar` qui prend en argument un tableau de données `df` et une variable `var` et qui retourne le diagramme en barre de la variable correspondante généré avec `ggplot2`.

```
graph_bar(starwars, gender)
```



Modifier la fonction `graph_bar` pour qu'elle accepte un troisième argument nommé `facet_var` et qu'elle retourne le diagramme en barre de `var` pour chaque valeur de `facet_var`.

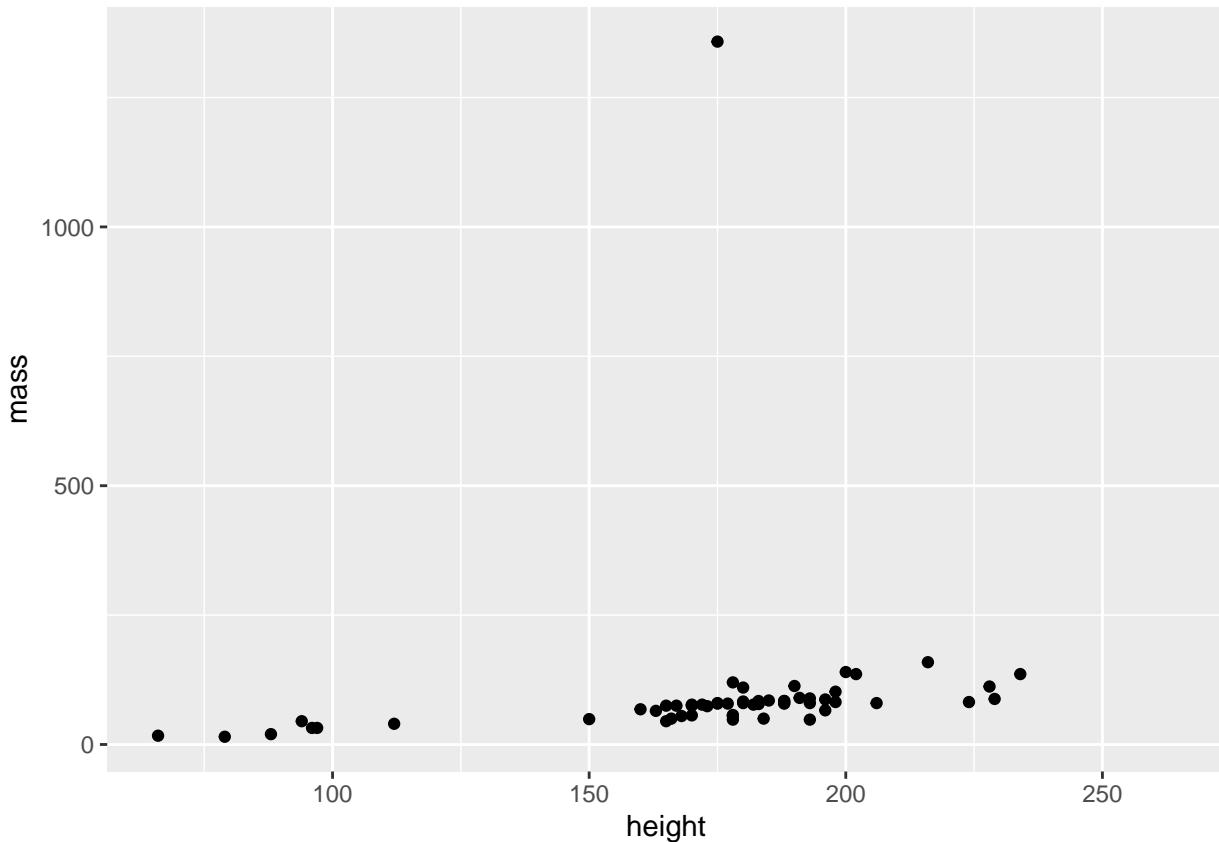
```
graph_bar(starwars, sex, gender)
```



Exercice 4.2

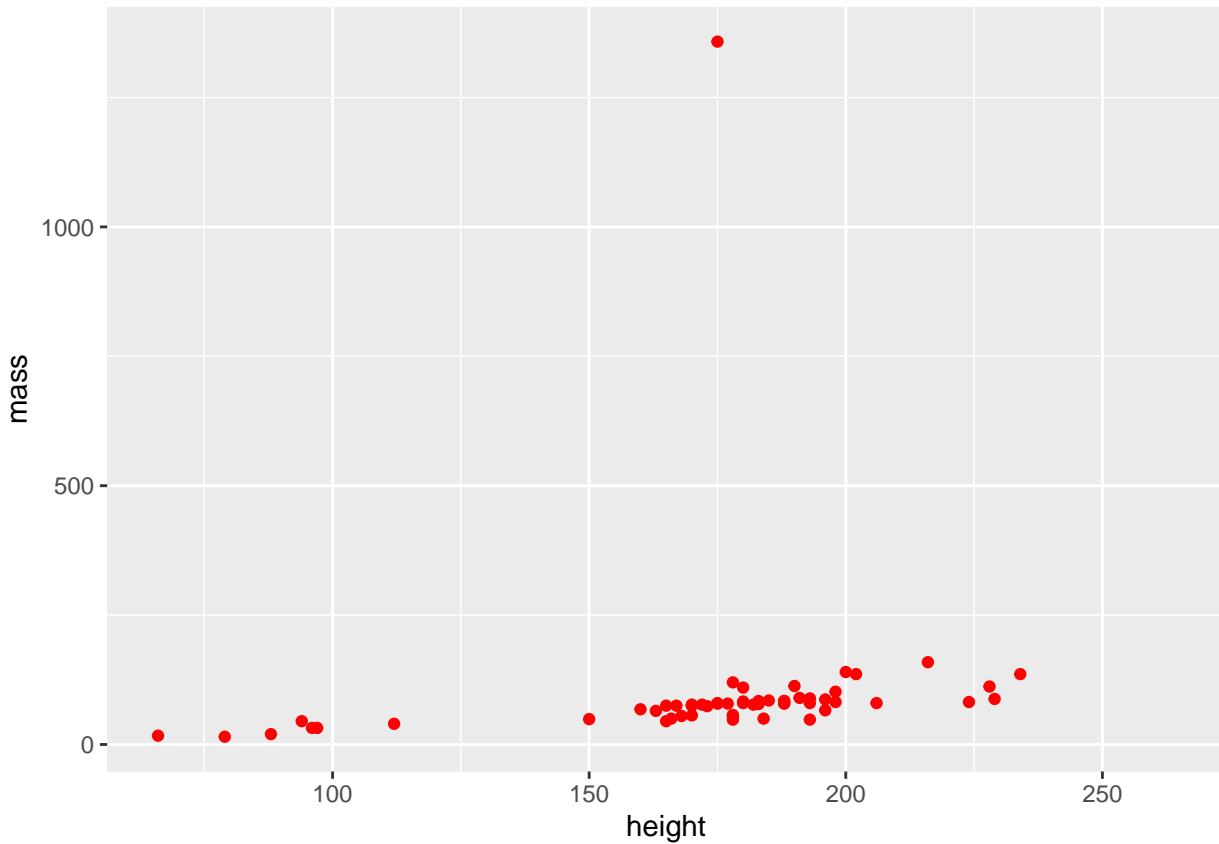
Créer une fonction `graph_points` qui prend en arguments un tableau de données et deux variables, et affiche le nuage de points de ces deux variables généré avec `ggplot2`.

```
graph_points(starwars, height, mass)
#> Warning: Removed 28 rows containing missing values (geom_point).
```



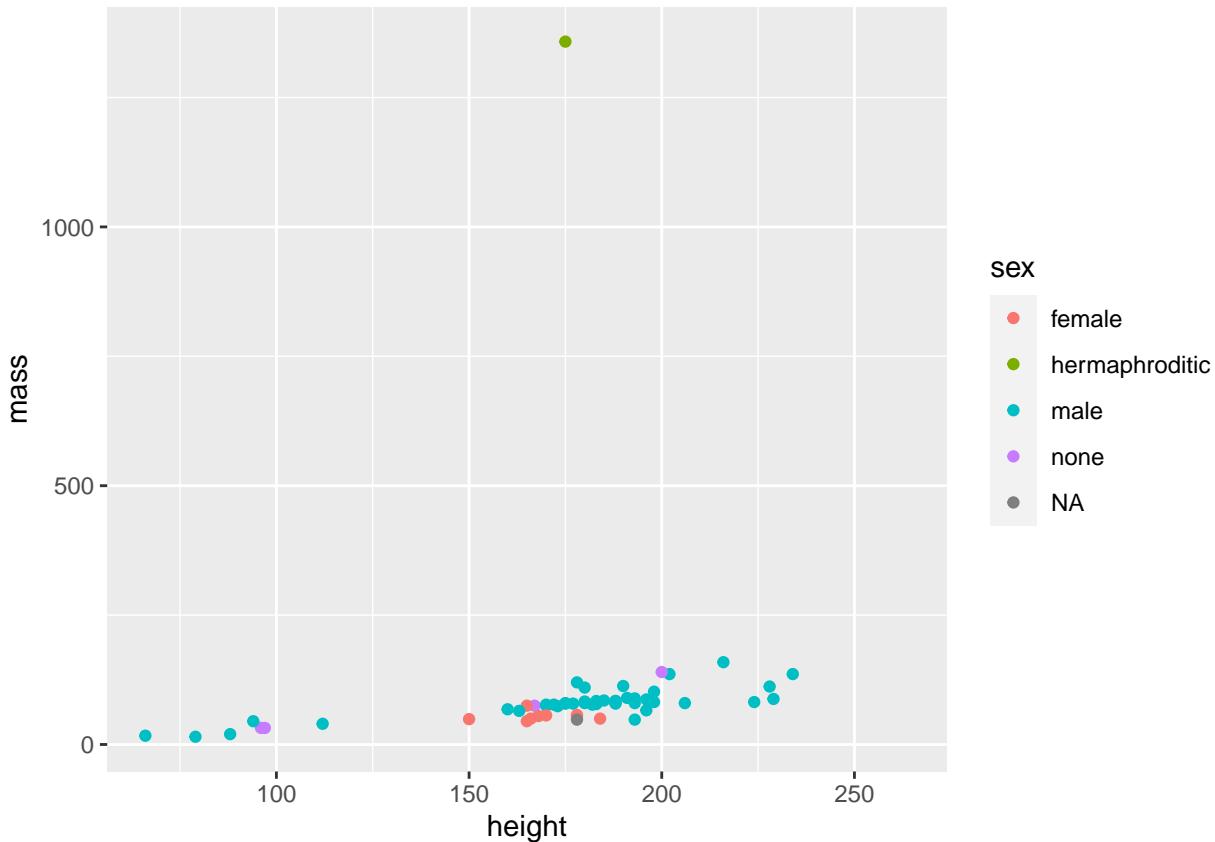
Ajouter un quatrième argument `color` à la fonction `graph_points()`, qui permet de spécifier la couleur des points (la même couleur pour tous).

```
graph_points(starwars, height, mass, color = "red")
#> Warning: Removed 28 rows containing missing values (geom_point).
```



Modifier `graph_points()` pour que l'argument `color` corresponde non pas à une couleur fixe mais à une variable dont les valeurs déterminent la couleur des points.

```
graph_points(starwars, height, mass, color = sex)
#> Warning: Removed 28 rows containing missing values (geom_point).
```



Facultatif : modifier `graph_points()` pour que l'argument `color` puisse accepter aussi bien une variable qu'une couleur fixe.

Indication : en s'a aidant du résultat de la fonction suivante, on pourra tester si la valeur passée à `color` est bien une colonne du tableau de données.

```
f <- function(x) {  
  deparse(substitute(x))  
}  
f(igloo)  
#> [1] "igloo"
```


Chapitre 20

Déboggage et performance

20.1 Déboguer une fonction

Lorsqu'on commence à écrire des fonctions un peu complexes, avec des `if`, des `for`, et autres joyeusetés, arrive forcément un moment où ça ne fonctionne pas comme on le souhaiterait et où on ne comprend pas pourquoi. Bref, *il y a un bug*.

Trouver la cause d'un bug n'est pas toujours évident, mais il existe plusieurs méthodes et outils permettant de faciliter un peu les choses.

On commence par charger les extensions et les jeux de données dont on aura besoin par la suite.

```
library(tidyverse)
library(questionr)
data(hdv2003)
data(starwars)
```

20.1.1 `print()`

L'outil le plus simple, rudimentaire et parfois décrié mais qui reste efficace, est de rajouter un `print()` dans le code de notre fonction pour examiner le contenu d'un objet à un moment donné.

Prenons un exemple. La fonction suivante prend en argument un tableau de données `df` et un nom de variable `var`, et retourne la moyenne et l'écart-type de cette variable.

```
indicateurs <- function(df, var) {
  valeurs <- df[, var]
  if (!is.numeric(valeurs)) return(NA)
  list(
    moyenne = mean(valeurs, na.rm = TRUE),
    sd = sd(valeurs, na.rm = TRUE)
  )
}
```

On teste notre fonction sur une variable du jeu de données `hdv2003`. Tout semble fonctionner.

```
indicateurs(hdv2003, "age")
#> $moyenne
#> [1] 48.157
```

```
#>
#> $sd
#> [1] 16.94181
```

On teste à nouveau, cette fois sur une variable du jeu de données `starwars` de `dplyr`.

```
indicateurs(starwars, "height")
#> [1] NA
```

Sapristi, on obtient `NA` en résultat alors que notre variable `height` est bien numérique : on devrait donc obtenir la moyenne et l'écart-type. Comment expliquer ce résultat ?

Comme on n'a pas d'explication immédiate juste en relisant le code de notre fonction, on va ajouter temporairement une instruction `print()` qui va afficher le contenu de `valeurs` juste après qu'elle soit calculée.

```
indicateurs <- function(df, var) {
  valeurs <- df[, var]
  print(valeurs)
  if (!is.numeric(valeurs)) return(NA)
  list(
    moyenne = mean(valeurs, na.rm = TRUE),
    sd = sd(valeurs, na.rm = TRUE)
  )
}
```

On lance cette fonction modifiée sur `starwars` :

```
indicateurs(starwars, "height")
#> # A tibble: 87 x 1
#>   height
#>   <int>
#> 1 172
#> 2 167
#> 3 96
#> 4 202
#> 5 150
#> 6 178
#> 7 165
#> 8 97
#> 9 183
#> 10 182
#> # ... with 77 more rows
#> [1] NA
```

Le `print()` nous indique que `valeurs` n'est pas un vecteur mais un tableau de données à une seule colonne. Or dans ce cas, le test `is.numeric`, qui est sensé s'appliquer à un vecteur atomique, renvoie `FALSE`.

```
is.numeric(starwars[, "height"])
#> [1] FALSE
```

C'est donc la raison pour laquelle on obtient `NA` comme résultat.

Certes, mais alors pourquoi cela fonctionne-t-il dans notre exemple avec `hdv2003` ?

```

indicateurs(hdv2003, "age")
#> [1] 28 23 59 34 71 35 60 47 20 28 65 47 63 67 76 49 62 20 70 39 30 30 37 79 20
#> [26] 74 31 35 35 30 54 29 49 59 41 41 53 19 77 56 54 35 40 62 68 77 39 57 43 34
#> [51] 60 41 54 23 26 23 74 60 23 57 55 71 39 56 54 27 40 42 44 41 30 48 59 46 33
#> [76] 70 67 75 22 51 41 53 70 79 31 34 72 59 63 77 48 25 62 23 66 90 43 32 54 19
#> [101] 61 57 54 31 33 86 84 74 49 47 25 52 71 56 58 66 70 51 54 73 35 64 51 68 42
#> [126] 49 55 49 60 71 75 36 78 19 23 81 41 58 37 49 55 51 58 29 47 78 31 55 42 35
#> [151] 50 78 56 40 35 46 50 69 73 32 38 18 43 62 35 28 51 65 47 66 48 27 71 38 61
#> [176] 39 69 79 57 64 59 47 24 54 50 32 60 57 50 23 48 49 70 27 52 19 54 45 64 71
#> [ reached getOption("max.print") -- omitted 1800 entries ]
#> $moyenne
#> [1] 48.157
#>
#> $sd
#> [1] 16.94181

```

Ah ! Dans ce cas là `valeurs` est bien un vecteur, et on obtient donc le résultat attendu. Pourquoi cette différence ? On est en fait tombé sur une différence de comportement entre les *data frames* et les *tibbles*, mentionnée section 16.3.3 : lorsqu'on sélectionne une seule colonne avec l'opérateur `[,]`, un *data frame* retourne un vecteur tandis qu'un *tibble* retourne un tableau à une colonne. Or, ici, `hdv2003` est un *data frame*, et `starwars` un *tibble*.

Comment résoudre ce problème ? Il y a différentes manières, mais la plus simple est sans doute de remplacer `[,]` par `[[]]`, qui lui a le même comportement dans les deux cas. On peut donc modifier notre fonction (en n'oubliant pas d'enlever le `print()`) et vérifier que ça fonctionne désormais.

```

indicateurs <- function(df, var) {
  valeurs <- df[[var]]
  if (!is.numeric(valeurs)) return(NA)
  list(
    moyenne = mean(valeurs, na.rm = TRUE),
    sd = sd(valeurs, na.rm = TRUE)
  )
}

indicateurs(hdv2003, "age")
#> $moyenne
#> [1] 48.157
#>
#> $sd
#> [1] 16.94181
indicateurs(starwars, "height")
#> $moyenne
#> [1] 174.358
#>
#> $sd
#> [1] 34.77043

```



Quand on ajoute des `print()` pour essayer d'identifier un problème, il faut bien penser à les supprimer une fois ce problème résolu, sinon on risque de se retrouver avec des messages “parasites” dans la console.

20.1.2 Localiser une erreur

`print()` peut aussi être utile pour savoir à quel moment une erreur se produit, notamment lorsqu'on utilise une boucle.

Dans l'exemple suivant, on crée une fonction `min_freq_values()` qui prend en argument un tableau de données `df` et un effectif `n_min` et retourne une liste des modalités de variables qui apparaissent dans au moins `n_min` lignes de `df` (on utilise ici une boucle `for`, mais on aurait aussi pu utiliser `map()`).

```
min_freq_values <- function(df, n_min) {
  res <- list()
  for (col in names(df)) {
    # Tri à plat des valeurs de la colonne
    freq <- table(df[[col]])
    # On conserve les modalités avec effectif >= n
    freq <- freq[freq >= n_min]
    # On ajoute la colonne au résultat s'il y a au moins une modalité
    if (length(freq) > 0) res[[col]] <- freq
  }
  res
}
```

Si on applique `min_freq_values()` à `hdv2003` avec une valeur de `n` à 1500, on obtient toutes les modalités correspondant à au moins 1500 observations.

```
min_freq_values(hdv2003, 1500)
#> $hard.rock
#> Non
#> 1986
#>
#> $lecture.bd
#> Non
#> 1953
#>
#> $peche.chasse
#> Non
#> 1776
```

Essayons maintenant d'appliquer `min_freq_values()` au jeu de données `starwars` de `dplyr`.

```
min_freq_values(starwars, 50)
#> Error in table(df[[col]]): tous les arguments doivent avoir la même longueur
```

On obtient un message quelque peu sybillin : apparemment une erreur se produit au moment de faire le tri à plat des valeurs avec `table()`. Mais comme ce tri à plat s'effectue dans une boucle, on ne sait pas quelle variable de `starwars` est à l'origine du problème.

On va donc rajouter un `print(col)` comme première instruction de la boucle `for`.

```
min_freq_values <- function(df, n_min) {
  res <- list()
  for (col in names(df)) {
    print(col)
    # Tri à plat des valeurs de la colonne
    freq <- table(df[[col]])
    # On conserve les modalités avec effectif >= n
    freq <- freq[freq >= n_min]
    # On ajoute la colonne au résultat s'il y a au moins une modalité
  }
  res
}
```

```

        if (length(freq) > 0) res[[col]] <- freq
    }
    res
}

min_freq_values(starwars, 50)
#> [1] "name"
#> [1] "height"
#> [1] "mass"
#> [1] "hair_color"
#> [1] "skin_color"
#> [1] "eye_color"
#> [1] "birth_year"
#> [1] "sex"
#> [1] "gender"
#> [1] "homeworld"
#> [1] "species"
#> [1] "films"
#> Error in table(df[[col]]): tous les arguments doivent avoir la même longueur

```

Ce `print(col)` nous permet de voir que tout se passe bien pour les premières variables du tableau, et que l'erreur survient au moment de traiter la variable `films`. On regarde donc à quoi ressemble cette variable.

```

head(starwars$films, 3)
#> [[1]]
#> [1] "The Empire Strikes Back" "Revenge of the Sith"
#> [3] "Return of the Jedi"      "A New Hope"
#> [5] "The Force Awakens"
#>
#> [[2]]
#> [1] "The Empire Strikes Back" "Attack of the Clones"
#> [3] "The Phantom Menace"      "Revenge of the Sith"
#> [5] "Return of the Jedi"      "A New Hope"
#>
#> [[3]]
#> [1] "The Empire Strikes Back" "Attack of the Clones"
#> [3] "The Phantom Menace"      "Revenge of the Sith"
#> [5] "Return of the Jedi"      "A New Hope"
#> [7] "The Force Awakens"

```

Damn, cette variable n'est pas un vecteur atomique mais une liste ! Un tableau de données peut en effet contenir ce que l'on appelle des *colonnes-listes*. Comme ça n'est pas courant, on ne l'avait clairement pas prévu au moment de la création de la fonction.

On a désormais identifié le problème, on peut donc le corriger par exemple en ignorant les colonnes de type liste. Pour cela on utilise `is.list()` et l'instruction `next`.

```

min_freq_values <- function(df, n_min) {
  res <- list()
  for (col in names(df)) {
    # On passe à la colonne suivante si la colonne est une colonne-liste
    if (is.list(df[[col]])) next
    # Tri à plat des valeurs de la colonne
    freq <- table(df[[col]])
    # On conserve les modalités avec effectif >= n
  }
}

```

```

freq <- freq[freq >= n_min]
# On ajoute la colonne au résultat s'il y a au moins une modalité
if (length(freq) > 0) res[[col]] <- freq
}
res
}

min_freq_values(starwars, 50)
#> $sex
#> male
#> 60
#>
#> $gender
#> masculine
#>       66

```

Ça fonctionne ! Et cela nous permet de repérer au passage une “légère” sur-représentation des personnages masculins...

20.1.3 browser()

Il existe des alternatives à `print()` plus efficaces pour identifier et résoudre des bugs. La plus utilisée est la fonction `browser()` : celle-ci peut s’insérer à n’importe quel endroit du code, et lorsqu’elle est rencontrée par R celui-ci s’interrompt et affiche une invite de commande permettant notamment d’inspecter des objets.

On reprend l’exemple précédent en remplaçant le `print(col)` que nous avions inséré pour déboguer la fonction `min_freq_values` par un appel à `browser()`.

```

min_freq_values <- function(df, n_min) {
  res <- list()
  for (col in names(df)) {
    browser()
    # Tri à plat des valeurs de la colonne
    freq <- table(df[[col]])
    # On conserve les modalités avec effectif >= n
    freq <- freq[freq >= n_min]
    # On ajoute la colonne au résultat s'il y a au moins une modalité
    if (length(freq) > 0) res[[col]] <- freq
  }
  res
}

min_freq_values(starwars, 50)

```

Lorsqu’on lance ce code, R s’interrompt et affiche l’invite de commande suivant :

```

Called from: min_freq_values(starwars, 50)

Browse[1]>

```

Cette invite de commande offre plusieurs possibilités. La première est d’indiquer du code R qui sera exécuté dans le contexte au moment de l’interruption : si on indique un nom d’objet, on pourra donc afficher sa valeur. Ainsi, si on tape `col`, R nous affiche la valeur actuelle de `col`, donc au moment de la première itération de la boucle.

```
Browse[1]> col
[1] "name"
```

On peut également fournir des commandes spécifiques : si on tape `n`, R va passer à l'instruction suivante puis s'interrompre à nouveau¹.

```
Browse[1]> n
debug à #6 : freq <- table(df[[col]])
```

Si on tape `c`, R va relancer l'exécution jusqu'à la fin, ou jusqu'à la prochaine rencontre d'un `browser()`. Dans notre cas, cela signifie qu'on va continuer jusqu'au `browser()` de la deuxième itération de la boucle.

```
Browse[2]> c
Called from: min_freq_values(starwars, 50)

Browse[1]> col
[1] "height"
```

Si on souhaite sortir de cette invite de commande et tout interrompre, il suffit de taper `Q`.

`browser()` est donc un peu plus complexe à utiliser que des `print()` ajoutés manuellement, mais c'est aussi un outil plus souple et plus puissant.



Cette section n'offre qu'un petit aperçu des possibilités de déboggage. R propose d'autres fonctionnalités, et les environnements de développement comme RStudio ou Visual Studio Code proposent également leurs propres outils.

Pour plus d'informations on pourra se reporter aux ressources en fin de chapitre.

20.2 benchmarking : mesurer et comparer les temps d'exécution

Lorsqu'on commence à créer ses propres fonctions et de manière générale à écrire du code de plus en plus complexe ou à travailler sur des données de plus en plus volumineuses, on peut arriver sur des problèmes de performance : les temps d'exécution deviennent longs et on aimerait essayer de les optimiser.

Il est alors parfois utile de faire du *benchmarking*, c'est-à-dire comparer plusieurs manières différentes de faire la même chose en mesurant leurs différences de vitesse d'exécution.

L'instruction la plus simple pour cela est sans doute `system.time()`. Celle-ci prend en argument une expression R, et affiche en retour le temps d'exécution en secondes.

```
system.time(runif(1000000))
#> utilisateur      système    écoulé
#>     0.035        0.000    0.035
```

L'expression peut comporter plusieurs instructions, dans ce cas on les entoure d'accolades.

¹Si on a un objet `n` dont on souhaite afficher le contenu, on doit faire `print(n)`.

```
system.time({
  v <- runif(1000000)
  moy <- mean(v)
})
#> utilisateur      système      écoulé
#>    0.037        0.000      0.037
```

`system.time()` est cependant très limitée : on ne peut exécuter qu'une expression à la fois, on ne peut donc pas en comparer plusieurs directement, et surtout le temps d'exécution peut varier assez sensiblement selon l'utilisation du processeur de la machine au moment où on lance la commande.

Il existe donc en complément plusieurs extensions dédiées au *benchmarking*. On va utiliser ici l'extension `bench`, installable avec :

```
install.packages("bench")
```

`bench` propose une fonction principale, nommée `mark()`, qu'on peut donc appeler directement avec `bench::mark()`². On passe à cette fonction plusieurs expressions, et `bench::mark()` va effectuer les actions suivante :

- elle vérifie que les expressions retournent bien le même résultat (si on souhaite comparer des expressions qui renvoient des résultats différents, il faut ajouter l'argument `check = FALSE`)
- elle lance chacune de ces expressions plusieurs fois et mesure à chaque fois leur temps d'exécution
- elle affiche un résumé de ces temps d'exécution en indiquant notamment leur minimum et leur médiane

Exécuter chaque expression plusieurs fois permet de prendre en compte les fluctuations liées à l'activité du processeur de la machine. Par défaut, `bench::mark()` exécute chaque instruction au minimum une fois et au moins suffisamment de fois pour atteindre 0,5s d'exécution (ces valeurs peuvent être modifiées via les paramètres `min_iterations` et `min_time`).

Dans l'exemple suivant, on crée deux fonctions qui font la même chose : ajouter 10 à tous les éléments d'un vecteur passé en argument. Dans la première fonction `plus10_for` on utilise une boucle `for` qui ajoute 10 à tous les éléments l'un après l'autre (ce qu'il ne faut évidemment pas faire !), tandis que la deuxième fonction `plus10_vec` utilise la forme vectorisée `x + 10`.

```
plus10_for <- function(x) {
  for (i in seq_along(x)) {
    x[i] <- x[i] + 10
  }
  x
}

plus10_vec <- function(x) {
  x + 10
}
```

On lance un benchmark avec `bench::mark()` et on stocke le résultat dans un objet.

```
x <- 1:10000
bnch <- bench::mark(
  plus10_for(x),
  plus10_vec(x)
)
```

²L'avantage d'utiliser `bench::mark()` est qu'on n'a pas besoin d'ajouter un `library(bench)` dans notre script.

On affiche les résultats obtenus :

```
bnch
#> # A tibble: 2 x 6
#>   expression      min    median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>     <dbl>
#> 1 plus10_for(x)  737.7us  804.2us    1240.    97.6KB      0
#> 2 plus10_vec(x)  17.3us   19.7us    25500.   117.3KB     7.65
```

La colonne la plus importante est sans doute la colonne `median`, qui affiche le temps médian d'exécution des deux expressions. Attention, l'unité de temps n'est pas forcément la même, ici l'exécution de `plus10_for` est affichée en millisecondes ("ms"), tandis que celle de `plus10_vec` l'est en microsecondes ("μs") donc avec une unité mille fois plus petite.

Si on préfère, on peut afficher les performances relatives des deux fonctions avec :

```
summary(bnch, relative = TRUE)
#> # A tibble: 2 x 6
#>   expression      min    median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <dbl> <dbl>     <dbl> <dbl>      <dbl>
#> 1 plus10_for(x)  42.6   40.8      1        1       NaN
#> 2 plus10_vec(x)  1       1      20.6     1.20      Inf
```

Ces résultats nous permettent de voir que la version `vec` est très nettement plus rapide que la version `for` ! Ce qui confirme le fait qu'il faut toujours prioriser l'utilisation de fonctions vectorisées lorsqu'elles existent.

On prend un second exemple : on souhaite mesurer si l'utilisation de `map` est plus rapide ou non qu'une boucle `for` pour une tâche équivalente.

Pour cela on commence par créer artificiellement une liste de 200 tableaux de données en dupliquant 100 fois une liste composée des tableaux `rp2018` et `hdv2003`.

```
dfs <- list(rp2018, hdv2003)
dfs <- rep(dfs, 100)
```

Puis on lance un benchmark sur une boucle `for` et un `map` qui retournent chacun les nombres de ligne des 200 tableaux. Cette fois on ne crée pas de fonctions : on passe le code directement à `bench::mark()`, avec des noms permettant de les identifier plus facilement dans les résultats.

```
bench::mark(
  boucle_for = {
    res <- list()
    for (df in dfs) {
      res <- c(res, nrow(df))
    }
    res
  },
  map = map(dfs, ~ nrow(.x) )
)
#> # A tibble: 2 x 6
#>   expression      min    median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>     <dbl>
#> 1 boucle_for    3.62ms  3.77ms    265.   177.4KB     2.08
#> 2 map          937.37us  1.01ms    983.   6.8KB      6.37
```

La lecture du résultat indique que le `map` est environ 4 fois plus rapide que la boucle `for`.

Attention cependant en lisant ces résultats : dans la comparaison précédente les temps d'exécution étant en millisecondes, même une différence du type “4 fois plus rapide” peut être quasiment imperceptible. Dans l'exemple suivant on compare de la même manière `for` et `map`, mais cette fois en effectuant une action plus coûteuse en temps de calcul (on génère des vecteurs de nombres aléatoires).

```
bench::mark(
  boucle_for = {
    res <- numeric()
    for (i in 1:100) {
      res <- c(res, mean(runif(50000)))
    }
    res
  },
  map = map_dbl(1:100, ~ mean(runif(50000))),
  check = FALSE
)
#> # A tibble: 2 x 6
#>   expression     min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>     <dbl>
#> 1 boucle_for    185ms    186ms      5.24    38.5MB      0
#> 2 map           205ms    205ms      4.88    38.4MB      0
```

Dans ce cas la différence entre `for` et `map` devient négligeable par rapport aux temps de calcul des opérations effectuées dans les boucles : au final les temps d'exécution sont presque identiques.

20.3 Quelques conseils d'optimisation

20.3.1 Privilégier les fonctions vectorisées

On l'a déjà dit, redit et reredit, une des forces de R est de proposer un grand nombre de fonctions vectorisées, c'est-à-dire prévues et optimisées pour s'appliquer à tous les éléments d'un vecteur. Quand une fonction vectorisée existe, il est donc toujours préférable de l'utiliser plutôt qu'une boucle ou un `map`.

```
v <- rep(c("Pomme", "Poire", "Fraise"), 100)
bench::mark(
  boucle = {
    map_int(v, str_count, "m")
  },
  vec = str_count(v, "m")
)
#> # A tibble: 2 x 6
#>   expression     min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>     <dbl>
#> 1 boucle        3.14ms   3.26ms     307.    1.22KB     4.26
#> 2 vec          71.7us  79.01us   12621.   1.22KB      0
```

20.3.2 Mettre le minimum d'opérations dans les boucles

Si vous utilisez une boucle, toute opération qu'elle contient, comme elle va être répétée, peut devenir rapidement très coûteuse. Il ne faut donc y mettre que les opérations réellement nécessaires.

La fonction suivante prend en entrée un tableau de données `d` et une chaîne de caractères `chr`, et retourne un vecteur nommé indiquant le nombre de fois où cette valeur apparaît dans chaque colonne du tableau. On a

décidé de convertir en minuscules à la fois la valeur de `chr` et l'ensemble des colonnes du tableau pour que le comptage ne prenne pas en compte les différences de majuscules/minuscules.

```
nb_chaine1 <- function(d, chr) {
  res <- numeric()
  for (var in names(d)) {
    # Conversion de chr et des colonnes de d en minuscules
    d <- d %>% modify(str_to_lower)
    chr <- str_to_lower(chr)
    # Comptage des occurrences de chr dans la colonne var
    res[[var]] <- sum(d[[var]] == chr, na.rm = TRUE)
  }
  res
}

d <- hdv2003[, c("hard.rock", "qualif", "clso", "bricol")]
nb_chaine1(d, "oui")
#> hard.rock      qualif      clso     bricol
#>       14          0        936       853
```

Ça fonctionne, mais si on regarde un peu plus attentivement le code de la fonction on peut vite se rendre compte d'un problème : les conversions en minuscules sont faites à l'intérieur de la boucle, et sont donc répétées pour chaque colonne de `d`. Or ceci n'est absolument pas nécessaire puisque cette conversion ne dépend pas des colonnes et qu'elle peut être faite une seule et unique fois en début de fonction.

On décide donc de sortir les conversions en minuscules de la boucle.

```
nb_chaine2 <- function(d, chr) {
  res <- numeric()

  # Conversion de chr et des colonnes de d en minuscules
  d <- d %>% modify(str_to_lower)
  chr <- str_to_lower(chr)

  for (var in names(d)) {
    # Comptage des occurrences de chr dans la colonne var
    res[[var]] <- sum(d[[var]] == chr, na.rm = TRUE)
  }
  res
}

nb_chaine2(d, "oui")
#> hard.rock      qualif      clso     bricol
#>       14          0        936       853
```

Le résultat des deux fonctions est identique. On compare leur temps d'exécution à l'aide de `bench::mark()`.

```
bench::mark(
  nb_chaine1(d, "oui"),
  nb_chaine2(d, "oui"),
)
#> # A tibble: 2 x 6
#>   expression      min   median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr> <bch:tm> <bch:tm>    <dbl> <bch:byt>    <dbl>
#> 1 nb_chaine1(d, "oui")  4.38ms   4.5ms     222.    345KB     0
#> 2 nb_chaine2(d, "oui")  1.24ms   1.34ms    746.    157KB     0
```

Ce qui permet de constater que la deuxième version est quatre fois plus rapide (et elle sera d'autant plus rapide que le nombre de colonnes du tableau sera grand).

20.3.3 Choisir un format de fichier adapté

Si on travaille sur des fichiers de données volumineux, les temps de chargement et de sauvegarde des données peuvent devenir importants.

Il existe différents formats de fichiers qui peuvent être plus ou moins rapides et pratiques selon l'utilisation qu'on en fait. Il n'est malheureusement pas toujours simple de s'y retrouver d'autant que les formats et les performances peuvent évoluer assez rapidement, mais on peut quand même donner quelques indications :

Le format **CSV** est pratique pour échanger des données tabulaires d'un système ou d'un logiciel à un autre, mais il présente plusieurs inconvénients :

- les fichiers sont volumineux
- les temps de lecture/écriture sont assez longs
- ils n'incluent pas de métadonnées comme les types des colonnes, et nécessitent donc des opérations de conversion plus ou moins "risquées"

À noter que plusieurs extensions et fonctions existent pour lire et/ou écrire les fichiers CSV. Outre `read.csv` de R base et `read_csv` de `readr`, on pourra mentionner `fread` de `data.table`, particulièrement rapide, ou l'extension `vroom`, qui ne charge pas toutes les informations d'un coup mais y accède "à la demande".

Les formats **Rdata** (utilisé par `load()` et `save()`) et **RDS** (utilisé par `readRDS()` et `saveRDS()`), propres à R, permettent de sauvegarder des objets dans un fichier. Ils ont des temps de lecture et d'écriture rapides, et permettent d'enregistrer n'importe quel objet R et d'être sûr de le retrouver à peu près à l'identique. À noter que ces formats permettent de faire varier le niveau de compression des données et donc jouer sur le rapport entre espace disque utilisé et temps d'accès. Ces formats sont intéressants notamment pour enregistrer des résultats intermédiaires. À noter que le format **fst**, proposé par [l'extension du même nom](#), offre des performances encore supérieures mais ne gère que les données tabulaires.

Pour des données vraiment volumineuses, le package **arrow** permet la lecture et l'enregistrement dans différents formats, dont **arrow** et **parquet**. Très optimisés mais limités aux données tabulaires, ils permettent également des échanges avec d'autres langages de programmation comme Python, JavaScript, etc.

À titre indicatif, le billet [a shallow benchmark of R data frame export/import methods](#) propose un comparatif entre les performances de différents formats de fichiers pour des données tabulaires. Attention cependant, il date de 2019 et les performances des fonctions testées ont pu évoluer depuis.

20.3.4 Repérer les opérations coûteuses avec du *profiling*

Quand on commence à écrire du code un peu long, il n'est pas toujours évident de savoir quelles sont les opérations qui sont les plus coûteuses en termes de performance. Il peut alors être utile d'utiliser un outil de *profiling*, qui consiste à exécuter du code tout en mesurant les temps d'exécution de chaque instruction, ce qui permet ensuite de visualiser et repérer les opérations qui prennent le plus de temps et qui seraient donc les plus intéressantes à optimiser (si c'est possible).

L'extension **profvis** fournit un outil de *profiling* sous R pratique et utile. On passe des instructions ou un script entier à la fonction `profvis()`, et celle-ci fournit une visualisation interactive des différentes opérations, de leur temps d'exécution, de leur utilisation de la mémoire...

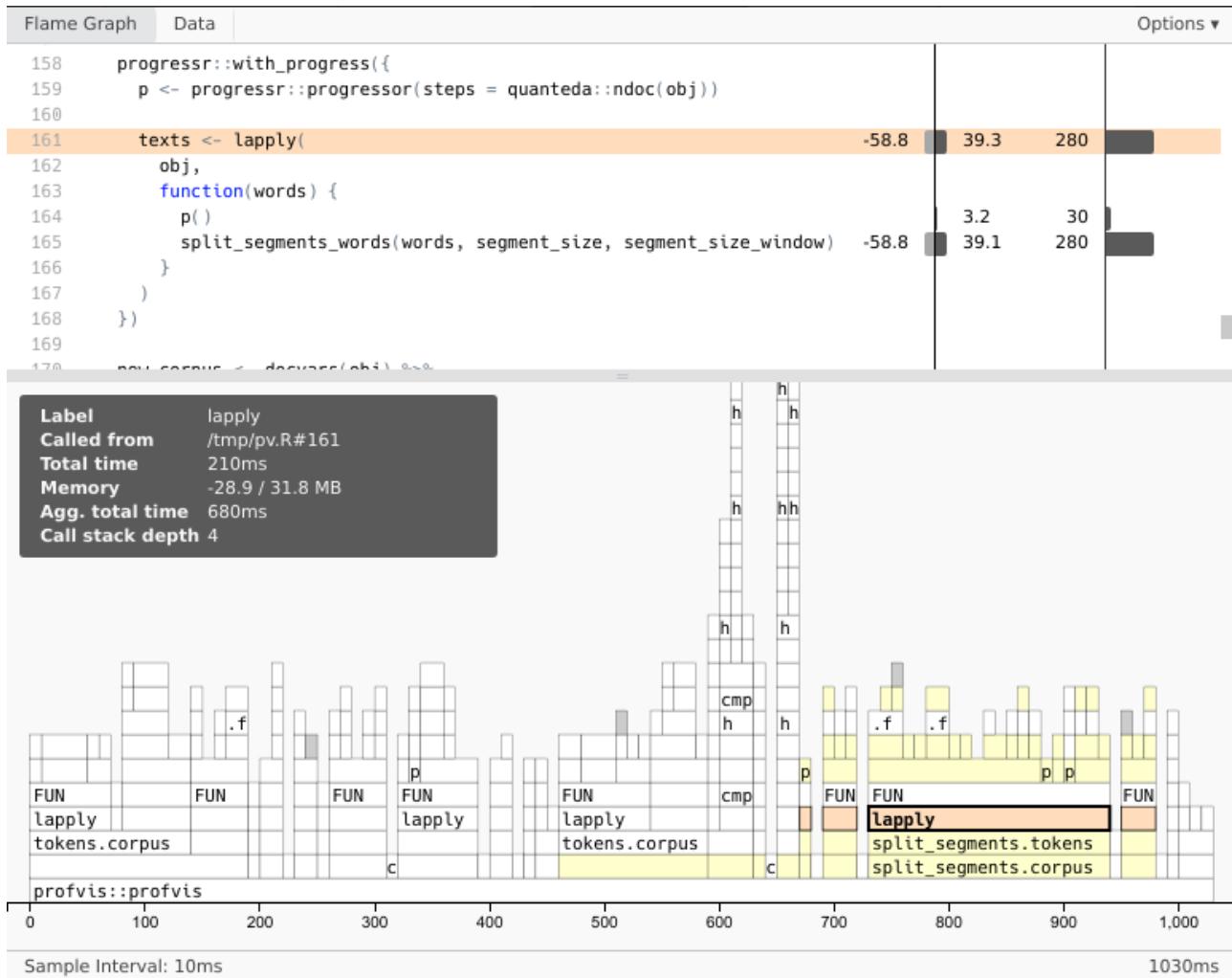


Figure 20.1: Exemple de visualisation générée par profvis()

L'utilisation de `profvis()` dépasse le cadre de ce document, mais on pourra trouver des explications (en anglais) dans la partie `profiling` d'Advanced R.

20.3.5 Mettre en cache des résultats intermédiaires

Selon la taille des données et le type d'opérations réalisées, les temps de calculs même optimisés peuvent demeurer longs. Dans ce cas il peut être intéressant de "mettre en cache" des résultats intermédiaires dans des fichiers pour pouvoir les charger directement sans avoir à tout recalculer.

Par exemple, si on travaille sur un corpus de données textuelles, on peut enregistrer dans un fichier `RDS` ou `Rdata` le résultat de toutes les opérations de prétraitement du corpus : si on souhaite faire des analyses par la suite on aura juste à charger les données depuis ce fichier sans avoir à relancer le calcul de tous les prétraitements.

L'extension `targets`, présentée section 21, est particulièrement adaptée pour les projets de ce type car elle gère automatiquement les dépendances entre les étapes d'un projet et la mise en cache des résultats, et permet ainsi d'optimiser les temps de traitement.

20.3.6 Utiliser d'autres extensions

Enfin, si ce document est basé sur les extensions du `tidyverse`, qui offrent une syntaxe cohérente et plus facile d'accès, il existe d'autres extensions notamment dans le domaine de la manipulation des données qui permettent des opérations plus rapides.

C'est en particulier le cas de l'extension `data.table`, qui utilise une syntaxe moins accessible que celle de `dplyr`, mais propose des performances en général [assez nettement supérieures](#). On trouvera la documentation détaillée de `data.table` (en anglais) [sur le site de l'extension](#).

20.4 Ressources

L'ouvrage *Advanced R* (en anglais) consacre un chapitre entier [au déboggage](#), un autre à [la mesure de la performance](#), et un dernier à différentes techniques d'[optimisation du code](#).

Le site de RStudio propose une page entière détaillant ses [fonctionnalités de debugging](#) (en anglais). Pour les utilisateurs de Visual Studio Code, on pourra se référer à l'extension [VSCode-R-Debugger](#).

Chapitre 21

Organiser un projet avec `targets`

`targets` est une extension développée par [Will Landau](#) qui permet d'organiser un projet sous la forme d'un *pipeline* de traitements, composé de différentes étapes, et gérant automatiquement les dépendances entre celles-ci¹. Cette organisation a plusieurs avantages :

- elle permet une description de toutes les étapes du pipeline dans un fichier dédié, et force à séparer ces différentes étapes dans des fonctions à part, ce qui facilite la lisibilité et la maintenance du projet
- elle facilite la reproductibilité des traitements, car elle garantit que toutes les étapes ont bien été effectuées dans le bon ordre et dans un nouvel environnement
- elle optimise les temps de calcul, car en cas de modification seules les étapes qui le nécessitent sont relancées

L'utilisation de `targets` dans des petits projets peut être vue comme une complexité supplémentaire pas toujours très utile, mais elle peut être très bénéfique pour des projets plus complexes ou comportant des temps de calculs importants à certaines étapes.

`targets` fait partie de l'initiative [rOpenSci](#).

21.1 Définition du pipeline

21.1.1 Projet d'exemple

On part d'un projet très simple : à partir du [fichier national des prénoms donnés à la naissance](#), diffusé par l'INSEE, on souhaite produire un document indiquant les prénoms ayant les évolutions les plus fortes (à la hausse ou à la baisse) entre 2019 et 2020.

Le dossier de notre projet s'organise de la manière suivante :

```
data/
  nat2020.csv
R/
  fonctions_recode.R
  fonctions_calculs.R
_targets.R
```

À noter que `targets` n'impose aucune structure de projet particulière en-dehors de la présence du fichier `_targets.R`. On aurait donc pu avoir une organisation tout à fait différente.

¹Pour les personnes habituées au développement, il s'agit d'un équivalent à [GNU Make](#) pour R.

Le fichier `data/nat2020.csv` contient les données brutes [téléchargées depuis le site de l'INSEE](#).

Le fichier `R/fonctions_recode.R` contient deux fonctions de traitement et de remise en forme des données.

```
# On conserve uniquement 2019 et 2020 et on
# filtre les lignes des prénoms rares regroupés
filter_data <- function(df) {
  df %>%
    filter(annais %in% c("2019", "2020")) %>%
    filter(preusuel != "_PRENOMS_RARES")
}

# Passage d'un format avec les années en ligne à un
# format avec les années en colonnes
pivot_2019_2020 <- function(df) {
  df %>%
    tidyr::pivot_wider(names_from = annais, values_from = nombre) %>%
    relocate(`2020`, .after = `2019`)
}
```

Le fichier `R/fonctions_calculs.R` contient une seule fonction qui calcule les variables d'évolution 2019-2020.

```
# Calcul des indicateurs d'évolution en effectifs et pourcentages
# pour les prénoms dont la fréquence est > à min_n
calcule_evo <- function(df, min_n = 200) {
  df %>%
    filter(`2020` > min_n | `2019` > min_n) %>%
    mutate(
      evo = (`2020` - `2019`),
      `evo%` = round(evo / `2019` * 100, 2)
    ) %>%
    drop_na(evo)
}
```

21.1.2 `_targets.R`

C'est dans le fichier `_targets.R`, situé à la racine du dossier, qu'on va définir le pipeline constitué de toutes les étapes de notre traitement : chargement et manipulation des données, calculs, génération de rapports, etc. Ces étapes sont également appelées *cibles* (*targets*).



La syntaxe présentée ici est celle proposée par l'extension `tarchetypes`, qui est un peu plus facile à prendre en main et plus lisible que la syntaxe native de `targets`.

Le fichier `_targets.R` commence par charger à la fois `targets` et `tarchetypes`.

```
# Packages nécessaires pour ce script
library(targets)
library(tarchetypes)
```

On va ensuite utiliser `source()` pour charger le contenu des deux fichiers `R/fonctions_recode.R` et `R/fonctions_calculs.R`, et pouvoir utiliser par la suite les fonctions qu'ils définissent.

```
# Chargement des fonctions
source("R/fonctions_recode.R")
source("R/fonctions_calculs.R")
```

On définit ensuite des options globales pour le pipeline. L'option `packages` de `tar_option_set()`, permet de spécifier une liste d'extensions à charger systématiquement avant le lancement de chaque étape. Ici on s'assure que l'extension `tidyverse` est bien chargée et disponible, et on positionne l'option `tidyverse.quiet` à `TRUE` pour supprimer le message qu'elle affiche systématiquement au chargement.

```
# Options pour les différentes étapes
options(tidyverse.quiet = TRUE)
tar_option_set(packages = "tidyverse")
```

Vient enfin la définition du pipeline proprement dit. Celle-ci se fait via la fonction `tar_plan()` de `tarchetypes`.

```
tar_plan(
)
```

La première opération que l'on souhaite effectuer est de charger les données contenues dans `data/nat2020.csv`. Pour cela on va d'abord créer une première étape qui consiste à *référencer* notre fichier CSV à l'aide de la fonction `tar_file()`.

```
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv")
)
```

Cette première étape définit une *cible (target)*, nommée `csv_file`, qui pointe vers notre fichier CSV.

On ajoute une seconde étape qui charge les données à l'aide de `read_csv2()`.

```
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv"),
  donnees_brutes = read_csv2(csv_file),
)
```

Cette nouvelle étape définit une deuxième cible nommée `donnees_brutes`. Cette cible correspond au nom d'une étape, mais aussi à un objet : dans ce qui suit, `donnees_brutes` correspond au tableau de données résultat du `read_csv2()`.

On va utiliser cet objet `donnees_brutes` dans une troisième étape nommée `donnees` qui lui applique les deux fonctions de filtrage et transformation définies dans `R/fonctions_recode.R`.

```
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv"),
  donnees_brutes = read_csv2(csv_file),
```

```
# Mise en forme des données
donnees = donnees_brutes %>%
  filter_data() %>%
  pivot_2019_2020()
)
```

Ici aussi, `donnees` est à la fois le nom d'une cible et un objet contenant nos données retravaillées. On utilise cet objet dans une étape supplémentaire qui utilise la fonction de R/`fonctions_calculs.R` pour calculer les variables d'évolution.

```
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv"),
  donnees_brutes = read_csv2(csv_file),

  # Mise en forme des données
  donnees = donnees_brutes %>%
    filter_data() %>%
    pivot_2019_2020(),

  # Calcul indicateurs
  donnees_evo = donnees %>%
    calcule_evo(min_n = 1000)
)
```



On notera que les cibles doivent toutes avoir des noms différents. Si on exécute plusieurs étapes de transformation ou de calcul sur un tableau de données, on devra donner un nom distinct à ces cibles et aux objets qui correspondent.

Au final, notre fichier `_targets.R` est donc le suivant :

```
# Packages nécessaires pour ce script
library(targets)
library(tarchetypes)

# Chargement des fonctions
source("R/fonctions_recode.R")
source("R/fonctions_calculs.R")

# Options pour les différentes étapes
options(tidyverse.quiet = TRUE)
tar_option_set(packages = "tidyverse")

# Définition du pipeline
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv"),
  donnees_brutes = read_csv2(csv_file),

  # Mise en forme des données
  donnees = donnees_brutes %>%
    filter_data() %>%
```

```

pivot_2019_2020(),

# Calcul indicateurs
donnees_evo = donnees %>%
  calcule_evo(min_n = 1000)

)

```



`targets` offre aussi la possibilité de définir notre pipeline directement dans un fichier RMarkdown, ce qui peut permettre notamment de mieux le documenter. Pour plus d'information on pourra se référer au chapitre [Target Markdown](#) du manuel en ligne.

21.2 Exécution du pipeline

Une fois notre pipeline défini, `targets` fournit des outils permettant de visualiser sa structure et son état, notamment la fonction `tar_visnetwork()`.

```
tar_visnetwork()
```

Les différentes cibles apparaissent sous forme de cercles, et les fonctions qui leur sont appliquées sous forme de triangles. Les flèches indiquent que `targets` a automatiquement créé un *réseau de dépendances* entre cibles et fonctions : ainsi la cible `donnees` dépend des fonctions `filter_data`, `pivot_2019_2020` et de la cible `donnees_brutes`, qui elle-même dépend de la cible `csv_file`.

La couleur des différents éléments montrent que ceux-ci sont à l'état *outdated* : ils ne sont pas à jour.

On va donc exécuter notre pipeline, en utilisant la fonction `tar_make()`.

```

tar_make()
#> • start target csv_file
#> • built target csv_file
#> • start target donnees_brutes
#>   Using ',',',' as decimal and '.!' as grouping mark. Use `read_delim()` for more control.
#>   Rows: 667364 Columns: 4
#>     Column specification
#>   Delimiter: ;"
#>   chr (2): preusuel, annais
#>   dbl (2): sexe, nombre
#>
#>   Use `spec()` to retrieve the full column specification for this data.
#>   Specify the column types or set `show_col_types = FALSE` to quiet this message.
#> • built target donnees_brutes
#> • start target donnees
#> • built target donnees
#> • start target donnees_evo
#> • built target donnees_evo
#> • end pipeline

```

Lorsqu'on utilise `tar_make()`, `targets` lance une nouvelle session R (pour éviter tout problème ou conflit lié à l'état de notre session actuelle), charge les extensions définies via `tar_option_set()`, et exécute les cibles définies avec `tar_plan()`.

On visualise le nouvel état de notre pipeline, et on voit que toutes les cibles sont passées à l'état *up to date*.

```
tar_visnetwork()
```

À chaque étape, `targets` crée et stocke dans un cache chacun des objets correspondants aux différentes cibles (`donnees_brutes`, `donnees`, etc.). On peut charger à tout moment ces objets dans notre session avec la fonction `tar_load()`².

```
tar_load(donnees_evo)
donnees_evo
#> # A tibble: 128 x 6
#>   sexe preusuel `2019` `2020`   evo `evo%`
#>   <dbl> <chr>     <dbl> <dbl> <dbl> <dbl>
#> 1     1 AARON      2443  2312 -131  -5.36
#> 2     1 ADAM       3670  3386 -284  -7.74
#> 3     1 ALEXANDRE  1154  1039 -115  -9.97
#> 4     1 AMIR        1588  1360 -228 -14.4
#> 5     1 ANTOINE     1786  1455 -331 -18.5
#> 6     1 ARTHUR      4008  3800 -208  -5.19
#> 7     1 AUGUSTIN    1546  1379 -167  -10.8
#> 8     1 AXEL        1809  1683 -126  -6.97
#> 9     1 AYDEN        1524  1786  262  17.2
#> 10    1 BAPTISTE     1403  1202 -201  -14.3
#> # ... with 118 more rows
```

On peut aussi utiliser `tar_read()`, qui lit et retourne les résultats d'une des cibles, permettant de les stocker dans un nouvel objet.

```
evo <- tar_read(donnees_evo)
```

21.3 Modification du pipeline

Essayons de lancer à nouveau notre pipeline :

```
tar_make()
#> skip target csv_file
#> skip target donnees_brutes
#> skip target donnees
#> skip target donnees_evo
#> skip pipeline
```

On voit que toutes les cibles ont été “skippées” : quand on lance `tar_make()`, seules les cibles qui sont à l'état *outdated* sont recalculées. Les résultats des autres sont conservés tels quels.

On va maintenant modifier légèrement notre fichier `R/fonctions_calculs.R` : plutôt que d'arrondir les évolutions en pourcentages à deux décimale, on n'en conserve plus qu'une.

```
# Calcul des indicateurs d'évolution en effectifs et pourcentages
# pour les prénoms dont la fréquence est > à min_n
calcule_evo <- function(df, min_n = 200) {
```

²Ces données sont stockées dans le répertoire `_targets` à la racine du projet.

```
df %>%
  filter(`2020` > min_n | `2019` > min_n) %>%
  mutate(
    evo = (`2020` - `2019`),
    `evo%` = round(evo / `2019` * 100, 1)
  ) %>%
  drop_na(evo)
}
```

On visualise à nouveau l'état de notre pipeline :

```
tar_visnetwork()
```

Grâce à sa gestion interne des dépendances entre les cibles, `targets` a vu que la fonction `calcule_evo` a été modifiée (elle est passée en statut *outdated*), et comme la cible `donnees_evo` dépend de cette fonction, celle-ci a également été placée en *outdated*. On peut obtenir directement une liste des cibles qui ne sont plus à jour à l'aide de la fonction `tar_outdated()` :

```
tar_outdated()
#> [1] "donnees_evo"
```

On relance notre pipeline :

```
tar_make()
#> skip target csv_file
#> skip target donnees_brutes
#> skip target donnees
#> • start target donnees_evo
#> • built target donnees_evo
#> • end pipeline
```

On voit que les cibles `csv_file`, `donnees_brutes` et `donnees` ont été “skippées” : `targets` est allé prendre directement leurs valeurs déjà stockées en cache. Par contre `donnees_evo` a bien été recalculée.

On peut vérifier que notre pipeline est désormais entièrement à jour :

```
tar_visnetwork()
```



À noter que `targets` gère aussi les modifications des fichiers externes. Ainsi, si on modifie le contenu de `nat2020.csv`, la cible `csv_file` passerait en *outdated*, tout comme l'ensemble des autres cibles puisqu'elles dépendent directement ou indirectement de celle-ci. Dans ce cas, un `tar_make()` aurait pour effet de recalculer l'intégralité du pipeline.

21.4 RMarkdown

Imaginons maintenant qu'on souhaite générer un rapport à partir d'un document RMarkdown en utilisant les données d'évolution calculées par notre pipeline. On crée donc un nouveau fichier `evolution.Rmd` dans un dossier `reports`.

```
data/
  nat2020.csv
R/
  fonctions_recode.R
  fonctions_calculs.R
reports/
  evolution.Rmd
_targets.R
```

Quand on utilise un document RMarkdown dans un pipeline, on doit accéder aux données en utilisant les fonctions `tar_read()` ou `tar_load()` : ceci permet de s'assurer qu'on récupère les données “à jour”, et cela permet aussi à `targets` de déterminer un lien de dépendance entre le document et les données.

Comme on souhaite utiliser les données de `donnees_evo`, on devra donc utiliser quelque chose comme :

```
d <- tar_read(donnees_evo)
```

Au final, le contenu de notre fichier RMarkdown est le suivant :

```
---
title: "Évolutions des prénoms 2019-2020"
date: "`r Sys.Date()`"
output:
  html_document:
    df_print: paged
---

```{r setup, include = FALSE}
knitr::opts_chunk$set(echo = FALSE)
d <- tar_read(donnees_evo)
```

## Plus fortes hausses

```{r}
d %>%
 arrange(desc(`evo%`)) %>%
 head(10)
```

## Plus fortes baisses

```{r}
d %>%
 arrange(`evo%`) %>%
 head(10)
```

```

Pour ajouter ce rapport à notre pipeline, on crée une nouvelle cible dans le `tar_plan()` de `_targets.R`. Comme il s'agit d'un document RMarkdown, on utilise la fonction `tar_render()`.

```

# Packages nécessaires pour ce script
library(targets)
library(tarchetypes)

# Chargement des fonctions
source("R/fonctions_recode.R")
source("R/fonctions_calculs.R")

# Options pour les différentes étapes
options(tidyverse.quiet = TRUE)
tar_option_set(packages = "tidyverse")

# Définition du pipeline
tar_plan(
  # Chargement du fichier CSV
  tar_file(csv_file, "data/nat2020.csv"),
  donnees_brutes = read_csv2(csv_file),

  # Mise en forme des données
  donnees = donnees_brutes %>%
    filter_data() %>%
    pivot_2019_2020(),

  # Calcul indicateurs
  donnees_evo = donnees %>%
    calcule_evo(min_n = 1000),

  # Génération rapport
  tar_render(report_evo, "reports/evolution.Rmd")
)

```

Visualisons notre pipeline modifié :

```
tar_visnetwork()
```

On voit que notre nouvelle cible `report_evo` a bien été prise en compte, qu'elle dépend bien de `donnees_evo` et qu'elle est à l'état *outdated*.

Si on exécute notre pipeline :

```

tar_make()
#> skip target csv_file
#> skip target donnees_brutes
#> skip target donnees
#> skip target donnees_evo
#> • start target report_evo
#> • built target report_evo
#> • end pipeline

```

La cible `report_evo` a bien été calculée, et on devrait retrouver notre rapport compilé au format HTML dans le dossier `reports`.

21.5 Gestion des données en cache

`targets` garde une copie des objets correspondant aux cibles du pipeline dans un cache, en fait sous forme de fichiers placés dans un sous-dossier `_targets`.

On a vu qu'on peut récupérer ces objets dans notre session via les fonctions `tar_read()` et `tar_load()`. `targets` propose également plusieurs fonctions pour gérer les données et métadonnées en cache :

- `tar_destroy()` supprime la totalité du répertoire `_targets`. Elle permet donc de “repartir de zéro”, sans aucun cache et avec toutes les cibles à recalculer.
- `tar_delete(donnees)` supprime l'objet `donnees` du cache et place l'état de la cible correspondante à *outdated*. Elle permet de forcer le recalculation d'une cible et de celles qui en dépendent. À noter qu'on peut sélectionner plusieurs cibles en utilisant la syntaxe de la *tidy selection*.
- `tar_prune()` permet de supprimer les cibles qui ne sont plus présentes dans le pipeline. Elle permet donc de “faire le ménage” quand on a supprimé des étapes dans `_targets.R`.

21.6 Avantages et limites

21.6.1 Avantages

On peut voir dans cette introduction rapide que l'utilisation de `targets` présente de nombreux avantages :

1. le fichier `_targets.R` fournit une description détaillée des étapes du projet. Cela facilite les choses quand on revient dessus après un certain temps et qu'on n'a plus tous les détails en tête, ou si on le partage avec quelqu'un.
2. chaque cible du pipeline est définie via des fonctions, ce qui garantit une séparation et une encapsulation des différentes étapes.
3. l'utilisation de `tar_make()` garantit que toutes les cibles du pipeline sont recalculées dans le bon ordre : pas de risque de lancer un script sur des données qui ne seraient pas complètement à jour parce qu'on a oublié de relancer certains recodages par exemple.
4. `tar_make()` s'exécute toujours dans un environnement vide, ce qui élimine les problèmes liés à l'état de notre session en cours et garantit la reproductibilité des résultats.
5. comme `targets` conserve une copie des résultats des cibles en cache, pas besoin de tout recalculer quand on relance le projet, on peut récupérer directement les résultats et savoir si ils sont à jour.
6. `tar_make()` ne calcule que les cibles qui le nécessitent, les temps de calcul et d'exécution sont optimisés.

Parmi les inconvénients liés à l'utilisation de `targets`, on notera que le débogage est un peu plus complexe, même si l'extension [fournit plusieurs outils](#) pour faciliter le travail.

21.6.2 Interactivité et développement du pipeline

Une des limitations de `targets` est que le pipeline ne permet pas l'utilisation de fonctions “interactives”. Par exemple, on pourrait ajouter une étape affichant un graphique dans `tar_plan()` :

```
graphique = ggplot(donnees_evo) + geom_histogram(aes(x = evo))
```

Ceci fonctionne, mais ne provoque pas l'affichage du graphique. Il faut faire un `tar_read(graphique)` pour pouvoir le visualiser.

De la même manière, on ne peut pas utiliser d'interfaces interactives comme celles vues pour faciliter les recodages de variables (par exemple section 9.3.2.1). Il est donc souvent pratique de commencer à développer des transformations, calculs ou analyses de façon “interactive”, via un script classique dans lequel on importe les données nécessaires via `tar_read()`. Une fois qu'on obtient le résultat souhaité, on transforme ce code en une ou plusieurs fonctions et on les intègre au pipeline de `targets`.



On notera que les documents RMarkdown s'utilisent très bien avec `targets` : du moment qu'on charge les données avec `tar_read()` ou `tar_load()`, ils permettent à la fois une utilisation “interactive” pendant leur écriture, et une intégration directe dans un pipeline avec `tar_render()` sans avoir besoin de les modifier.

21.7 Ressources

Nous n'avons vu ici qu'un petit aperçu des fonctionnalités de **targets**, qui est une extension extrêmement riche. Celle-ci propose de nombreuses autres possibilités, comme la parallélisation des calculs, la gestion des versions de paquets via **renv**, la création programmatique de cibles...

Le *package* bénéficie d'une excellente documentation en anglais. On pourra donc se référer aux sites officiels de **targets** et **tarchetypes**, mais surtout à l'ouvrage en ligne [The targets R Package User Manual](#), très clair et très complet.

Le groupe des utilisateurs de R de Lille a accueilli une intervention (toujours en anglais) de Will Landau, l'auteur de **targets**. Celle-ci est disponible [en vidéo sur YouTube](#).

Appendix A

Ressources

A.1 Aide

A.1.1 Aide de R et RStudio

Il est possible d'obtenir à tout moment de l'aide (en anglais) sur une fonction en tapant `help()` avec comme argument le nom de la fonction dans la console :

```
help("mean")
```

Vous pouvez aussi aller dans l'onglet *Help* de l'interface de RStudio (dans le quart de l'écran en bas à droite) et utiliser le moteur de recherche intégré.

Chaque page d'aide est très complète mais pas toujours très accessible. Elle est structurée selon différentes sections, notamment :

- *Description* : donne un résumé en une phrase de ce que fait la fonction
- *Usage* : indique la ou les manières de l'utiliser
- *Arguments* : détaille les arguments possibles et leur signification
- *Value* : indique la forme du résultat renvoyé par la fonction
- *Details* : apporte des précisions sur le fonctionnement de la fonction
- *See Also* : renvoie vers d'autres fonctions semblables ou liées, ce qui peut être très utile pour découvrir ou retrouver une fonction dont on a oublié le nom
- *Examples* : donne une série d'exemples d'utilisation

Les exemples d'une page d'aide peuvent être exécutés directement dans la console avec la fonction `example` :

```
example("mean")
```

L'onglet *Help* de RStudio permet d'afficher mais aussi de naviguer dans les pages d'aide de R et dans d'autres ressources :

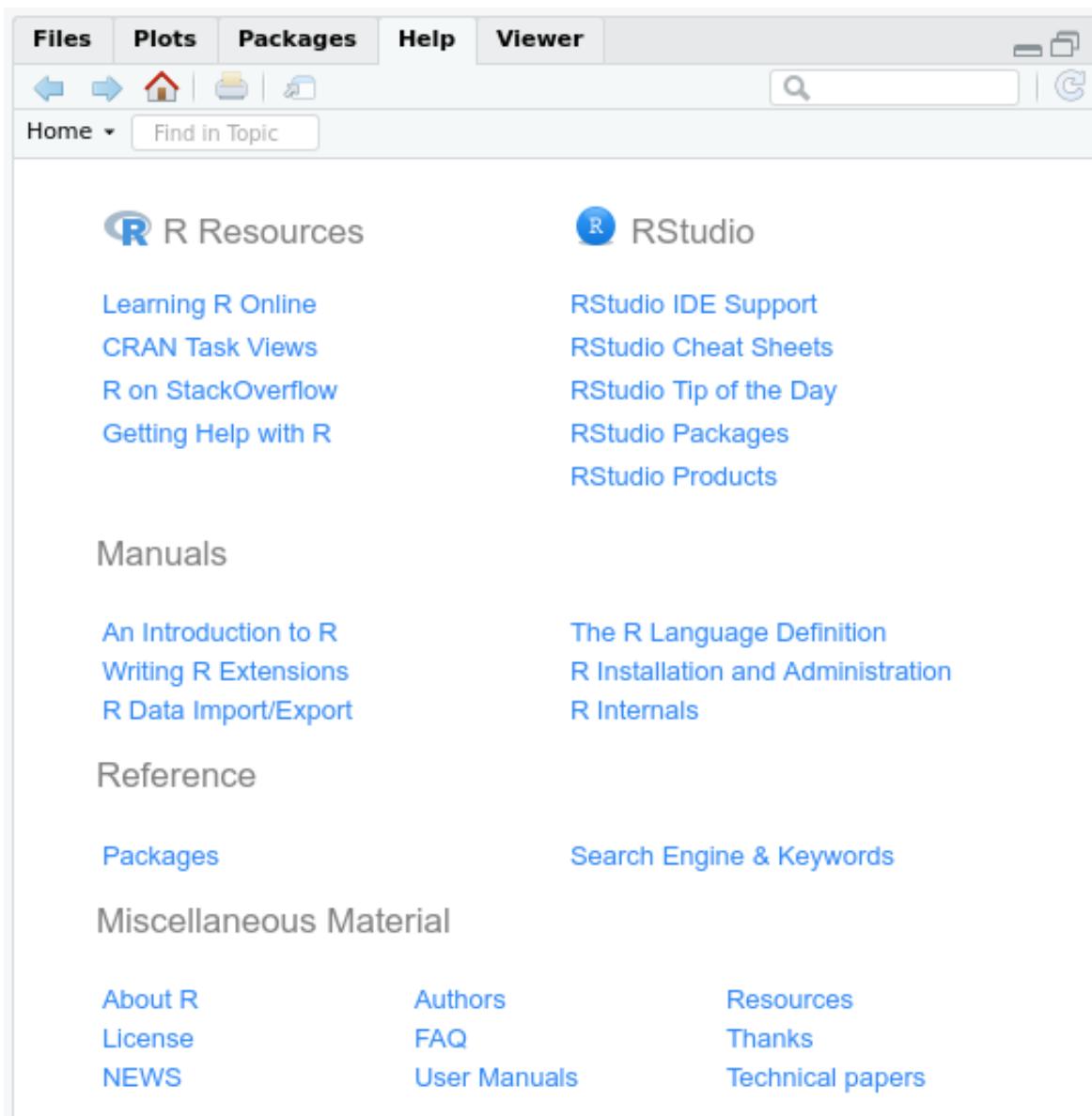


Figure A.1: Onglet *Help* de Rstudio

Cette page permet d'accéder aux manuels officiels de R (section *Manuals*), qui abordent différents aspects plus ou moins techniques du langage et du logiciel, en anglais. On citera notamment les documents *An Introduction to R* et *R Data Import/Export*. Elle propose également un lien vers la FAQ officielle.

A.1.2 Aide en ligne

Plusieurs sites proposent une interface permettant de naviguer et rechercher dans l'aide de R et de l'ensemble des extensions existantes.

C'est notamment le cas de rdrr.io.

A.1.3 Antisèches

RStudio propose plusieurs *cheat sheets* (antisèches) en anglais qui proposent sur deux pages une synthèse compacte de fonctions et de leur usage selon différentes thématiques, notamment :

- Manipulation des données avec `dplyr`
- Visualisation avec `ggplot2`

- Création de documents avec RMarkdown
- etc.

La liste complète est disponible en ligne :

<https://www.rstudio.com/resources/cheatsheets/>

Ou directement depuis RStudio, via le menu *Help*, puis *Cheatsheets*.

A.1.4 Où poser des questions

Outre l'aide intégrée au logiciel, il existe de nombreuses ressources en ligne, forums, listes de discussions, pour poser ses questions et échanger avec des utilisateurs et utilisatrices de R.

Le meilleur moyen d'obtenir une réponse est de poser la question de manière à ce qu'il soit aussi facile que possible d'y répondre. Ce qui implique de donner le maximum d'informations possibles et, si possible, de fournir un exemple de reproductible (un extrait de code et de données permettant de reproduire un problème ou de montrer le résultat qu'on souhaite obtenir). Pour des conseils sur les bonnes pratiques pour poser une question, on pourra se référer au billet [Reprex, ou comment demander de l'aide efficacement](#) sur le blog de ThinkR.

A.1.4.1 Discussion instantanée

Grrr (“pour quand votre R fait Grrr”) est un groupe Slack (plateforme de discussion instantanée) francophone dédié aux échanges et à l’entraide autour de R. Il est ouvert à tous et se veut accessible aux débutants. Vous pouvez même utiliser un pseudonyme si vous préférez.

Pour rejoindre la discussion, il suffit de suivre le lien d’invitation suivant :

https://join.slack.com/t/r-grrr/shared_invite/zt-46utbgb9-uv0_bg5cbux0V~H10YUX8w

A.1.4.2 Listes de discussion

La liste **R-soc** est une liste francophone spécialement dédiée aux utilisateurs et utilisatrices de R en sciences sociales. Toutes les questions y sont les bienvenues, et les réponses sont en général assez rapides. Il suffit de s’y abonner pour pouvoir ensuite poster sa question :

- <https://groupes.renater.fr/sympa/subscribe/r-soc>

La liste **semin-r** est la liste de discussion du groupe des utilisateurs et utilisatrices de R animé par le Muséum national d’Histoire naturelle. Elle est ouverte à tous et les questions y sont bienvenues :

- <https://listes.mnhn.fr/wws/subscribe/semin-r>

Il existe aussi une liste officielle anglophone baptisée **R-help**. Elle est cependant à réservier aux questions les plus pointues, et dans tous les cas il est nécessaire d’avoir en tête et de respecter les [bonnes pratiques](#) avant de poster sur la liste :

- <https://stat.ethz.ch/mailman/listinfo/r-help>

A.1.4.3 Sur le Web

Pour les anglophones, la ressource la plus riche concernant R est certainement le site [StackOverflow](#). Sous forme de questions/réponses, il comporte un très grand nombre d’informations sur R et les réponses y sont très rapides. Avant de poster une question il est fortement recommandé de faire une recherche sur le site, car il y a de fortes chances que celle-ci ait déjà été posée :

- <https://stackoverflow.com/questions/tagged/r>

Pour les francophones, on pourra citer le forum du CIRAD, qui comporte une section *questions en cours* assez active. Là aussi, pensez à faire une recherche sur le forum avant de poser votre question :

- <http://forums.cirad.fr/logiciel-R/>

A.2 Ouvrages, blogs, MOOCs...

A.2.1 Francophones

Parmi les ouvrages en français, on peut citer notamment :

- Les [formations R](#) très complètes développées par les agents de plusieurs ministère.
- [R et espace](#), manuel d'initiation à la programmation avec R appliquée à l'analyse de l'information géographique, librement téléchargeable en ligne.
- [utilitR](#), un ouvrage en ligne de formation à R à destination principalement des agents de l'INSEE mais qui aborde un grand nombre de sujets.

Le pôle bioinformatique lyonnais (PBIL) propose depuis longtemps une somme très importante de documents, qui comprend des cours complets de statistiques utilisant R :

- <https://pbil.univ-lyon1.fr/R/>

Plusieurs blogs francophones autour de R sont également actifs, parmi lesquels :

- [ElementR](#), le blog du groupe du même nom, qui propose de nombreuses ressources sur R en général et en particulier sur la cartographie ou l'analyse de réseaux.
- [R-atique](#), blog animé par Lise Vaudor, propose régulièrement des articles intéressants et accessibles sur des méthodes d'analyse ou sur des extensions R.

Pour des formations en ligne, le site *France Université Numérique* propose régulièrement des sessions de cours, parmi lesquels une [Introduction à la statistique avec R](#) et un cours sur [l'Analyse des données multidimensionnelles](#).

Enfin, le projet [Rzine](#) effectue un important travail de recensement des ressources sur R en particulier pour les sciences humaines et sociales.

A.2.2 Anglophones

Les ressources anglophones sont évidemment très nombreuses.

On citera essentiellement l'ouvrage en ligne [R for data science](#), très complet, et qui fournit une introduction très complète et progressive à R, et aux packages du *tidyverse*. Il existe également en version papier.

Pour aborder des aspects beaucoup plus avancés, l'ouvrage également en ligne [Advanced R](#), d'Hadley Wickham, est extrêmement bien fait et très complet.

On notera également l'existence du [R journal](#), revue en ligne consacrée à R, et qui propose régulièrement des articles sur des méthodes d'analyse, des extensions, et l'actualité du langage.

La plateforme [R-bloggers](#) agrège les contenus de plusieurs centaines de blogs parlant de R, très pratique pour suivre l'actualité de la communauté.

Enfin, sur Twitter, les échanges autour de R sont regroupés autour du *hashtag #rstats*.

A.3 Extensions

A.3.1 Où trouver des extensions intéressantes ?

Il existe plusieurs milliers d'extensions pour R, et il n'est pas toujours facile de savoir laquelle choisir pour une tâche donnée.

Si un des meilleurs moyens reste le bouche à oreille, on peut aussi se reporter à la page [CRAN Task view](#) qui liste un certain nombre de domaines (classification, sciences sociales, séries temporelles...) et indique, pour chacun d'entre eux, une liste d'extensions potentiellement intéressantes accompagnées d'une courte description.

Une autre possibilité est de consulter la page [listant l'ensemble des packages existant](#). S'il n'est évidemment pas possible de passer en revue les milliers d'extensions une à une, on peut toujours effectuer une recherche dans la page avec des mots-clés correspondant aux fonctionnalités recherchées.

Un autre site intéressant est [Awesome R](#), une liste élaborée collaborativement des extensions les plus utiles ou les plus populaires classées par grandes catégories : manipulation des données, graphiques interactifs, etc.

La page [frfrenchies](#) liste des packages pouvant être utiles pour des utilisateurs et utilisatrices francophones (géolocalisation, traitement du langage, accès à des API...), ainsi que des ressources en français.

Enfin, certaines extensions fournissent des “galeries” permettant de repérer ou découvrir certains *packages*. C'est notamment le cas de [R Markdown](#) ou de [htmlwidgets](#), qui propose une [galerie d'extensions proposant des graphiques interactifs](#).

A.3.2 L'extension `questionr`

`questionr` est une extension utilisée régulièrement dans ce document et comprenant quelques fonctions utiles pour l'utilisation du logiciel en sciences sociales, ainsi que différents jeux de données. Elle est développée en collaboration avec François Briatte et Joseph Larmarange.

L'installation se fait soit via le bouton *Install* de l'onglet *Packages* de RStudio, soit en utilisant la commande suivante dans la console :

```
install.packages("questionr")
```

Il est possible d'installer la version de développement à l'aide de la fonction `install_github` de l'extension `remotes` :

```
remotes::install_github("juba/questionr")
```

`questionr` propose à la fois des fonctions, des interfaces interactives et des jeux de données d'exemple.

A.3.2.1 Fonctions et utilisation

Pour plus de détails sur la liste des fonctions de l'extension et son utilisation, on pourra se reporter au site [Web de l'extension](#), hébergé sur GitHub.

L'onglet [Reference](#) liste l'ensemble des fonctions de `questionr`, tandis que l'onglet [Articles](#) propose une [présentation des trois interfaces interactives \(Addins\)](#) visant à faciliter le recodage de certaines variables.

Ces interfaces sont également abordées dans la partie 9.

A.3.2.2 Jeu de données `hdv2003`

`hdv2003` est un extrait comportant 2000 individus et 20 variables provenant de l'enquête *Histoire de Vie* réalisée par l'INSEE en 2003.

L'extract est tiré du fichier détail [mis à disposition librement](#) (ainsi que de nombreux autres) par l'INSEE. On trouvera une [documentation complète](#) à la même adresse.

Les variables retenues ont été parfois partiellement recodées. La liste des variables est la suivante :

| Variable | Description |
|--------------------|-------------------------------|
| <code>id</code> | Identifiant (numéro de ligne) |
| <code>poids</code> | Variable de pondération |
| <code>age</code> | Âge |
| <code>sexe</code> | Sexe |

| Variable | Description |
|---------------|--|
| nivetud | Niveau d'études atteint |
| occup | Occupation actuelle |
| qualif | Qualification de l'emploi actuel |
| freres.soeurs | Nombre total de frères, sœurs, demi-frères et demi-sœurs |
| clso | Sentiment d'appartenance à une classe sociale |
| relig | Pratique et croyance religieuse |
| trav.imp | Importance accordée au travail |
| trav.satisf | Satisfaction ou insatisfaction au travail |
| hard.rock | Écoute du Hard rock ou assimilés |
| lecture.bd | Lecture de bandes dessinées |
| peche.chasse | Pêche ou chasse pour le plaisir au cours des 12 derniers mois |
| cuisine | Cuisine pour le plaisir au cours des 12 derniers mois |
| bricol | Bricolage ou mécanique pour le plaisir au cours des 12 derniers mois |
| cinema | Cinéma au cours des 12 derniers mois |
| sport | Sport ou activité physique pour le plaisir au cours des 12 derniers mois |
| heures.tv | Nombre moyen d'heures passées à regarder la télévision par jour |



Comme il s'agit d'un extrait du fichier, la variable de pondération n'a en toute rigueur aucune valeur statistique. Elle a été tout de même incluse à des fins "pédagogiques".

A.3.2.3 Jeu de données rp2018

rp2018 est un jeu de données issu du [recensement de la population de 2018](#) de l'INSEE. Il comporte une petite partie des résultats pour l'ensemble des communes françaises de plus de 2000 habitants, soit au final 5417 communes et 62 variables.

Liste de quelques variables du fichier :

| Variable | Description |
|-------------|---|
| commune | nom de la commune |
| code_insee | Code de la commune |
| pop_tot | Population totale |
| pop_act_15p | Population active de 15 ans et plus |
| log_rp | Nombre de résidences principales |
| agric | Part des agriculteurs dans la population active |
| indep | Part des artisans, commerçants et chefs d'entreprises |
| cadres | Part des cadres |
| interm | Part des professions intermédiaires |
| empl | Part des employés |
| ouvr | Part des ouvriers |
| chom | Part des chômeurs |
| etud | Part des étudiants |
| dipl_sup | Part des diplômés de niveau Bac+5 ou plus |
| dipl_aucun | Part des personnes sans diplôme |
| proprio | Part des propriétaires parmi les résidences principales |
| hlm | Part des logements HLM parmi les résidences principales |
| locataire | Part des locataires parmi les résidences principales |
| maison | Part des maisons parmi les résidences principales |