

Notes de cours sélectionnées

STT-4230 / STT-6230

Références pour le défi technique 1

Extraits des notes de Sophie Baillargeon, Université Laval (2021). Sélection préparée le 22 septembre 2026. Consulter ces pages au besoin; l'aide-mémoire de six pages offre les repères rapides. Utiliser les signets du lecteur PDF ou Ctrl+F pour rechercher une notion.

Projets, importation et exportation : pages 2 à 10

Structures, types et conversions : pages 11 à 21

Transformations et textes : pages 22 à 38

Comparaisons et conditions logiques : pages 39 à 42

Résumés et valeurs manquantes : pages 43 à 45

Calculs par groupe : pages 46 à 50

Les passages retenus sont reproduits sans réécriture, avec leur ordre d'origine. Les sections non retenues sont omises et la mise en pages est adaptée. Des exemples mentionnent des chemins ou des fichiers de démonstration absents du poste : ils illustrent une syntaxe et ne sont pas des consignes d'exécution.

Les notes datent de 2021. Pour le comportement de la version installée, consulter l'aide locale. Les ressources et les consignes actuelles du défi prévalent. Attribution, liens d'origine et licence CC BY-NC-SA 4.0 : [SOURCES.html](#).

1 Accéder au système de fichiers

Afin d'accéder à des données à partir de R, il faut souvent aller les lire là où elles se trouvent. Si les données sont dans un fichier situé sur l'ordinateur local, dans un certain répertoire, il faut aller lire son contenu à cet endroit.

Dans toute commande R effectuant de la lecture ou de l'écriture dans un fichier, il faut identifier ce fichier. Il peut être identifié en utilisant son chemin d'accès complet (en anglais *full* ou *absolute path*), ou encore en utilisant un chemin d'accès relatif à un emplacement courant (en anglais *relative path*). Peu importe l'option choisie, le chemin doit se terminer par le nom complet du fichier, incluant son extension.

Par exemple, supposons que le fichier `data_ex.txt` se trouve dans le répertoire `C:\coursR` de l'ordinateur d'un utilisateur de R sous Windows. Le chemin d'accès complet de ce fichier est `C:\coursR\data_ex.txt`. Le chemin d'accès du fichier relatif au répertoire `C:\coursR` est pour sa part simplement le nom du fichier, soit `data_ex.txt`.

Remarques importantes

Bien que le caractère utilisé pour séparer les composantes dans un chemin d'accès sous Windows soit `\`, ce caractère est réservé en R pour les « séquences d'échappement ». Il s'agit de séquences ayant une signification particulière dans une chaîne de caractères (par exemple `"\t"` représente une tabulation, `"\n"` représente une nouvelle ligne, etc.). Dans un chemin d'accès en R, il faut plutôt séparer les composantes par le caractère `/`, ou encore par `\\`.

De plus, dans une commande R, un chemin d'accès doit toujours être présenté sous forme de chaîne de caractères, donc être encadré de guillemets.

Ainsi, dans une commande R, le chemin d'accès `C:\coursR\data_ex.txt` devrait être écrit ainsi : `"C:/coursR/data_ex.txt"`. Ce chemin pourrait aussi être produit avec la [fonction `file.path`](#) comme suit.

```
file.path("C:", "coursR", "data_ex.txt")
```

```
## [1] "C:/coursR/data_ex.txt"
```

1.1 Répertoire de travail

Lorsqu'un fichier est identifié par un chemin d'accès relatif dans une commande, R doit faire une supposition quant au répertoire à partir duquel chercher le fichier. C'est justement l'utilité du répertoire de travail (ou répertoire courant, en anglais *working directory*). Il s'agit de l'emplacement de base, celui à partir duquel débiter un chemin d'accès à un fichier lorsque le début du chemin n'est pas spécifié par l'utilisateur.

Pour connaître le répertoire de travail d'une session R, il suffit de soumettre la commande suivante :

```
getwd()
```

Pour accéder au fichier `data_ex.txt` en utilisant seulement son nom, il faut d'abord s'assurer que le répertoire de travail de la session R correspond au répertoire contenant le fichier. La fonction `setwd` permet de modifier le répertoire de travail. Par exemple, la commande suivante change le répertoire de travail pour `"C:/coursR"`.

```
setwd("C:/coursR")
```

Si le répertoire de travail avait été fixé à `"C:"`, le chemin d'accès relatif du fichier aurait été `"coursR/data_ex.txt"`.

Symboles reconnus dans les chemins d'accès

Dans les chemins d'accès à des fichiers en R, les symboles `.`, `..` et `~` ont les significations usuelles suivantes :

- `.` : répertoire de travail courant,
- `..` : répertoire parent du répertoire de travail courant,
- `~` : répertoire de travail par défaut (en anglais *home directory*).

Ainsi, si le répertoire de travail était `"C:/coursR"`, les chemins d'accès `"C:/coursR/data_ex.txt"`, `"../coursR/data_ex.txt"`, `"/data_ex.txt"` et `"data_ex.txt"` seraient tous équivalents.

1.2 Explorer le contenu du système de fichiers

R a donc accès au système de fichiers de l'ordinateur à partir duquel la session a été démarrée. Directement dans la console R, il est possible de voir le contenu d'un répertoire, de créer des fichiers et des répertoires, d'effacer des fichiers, etc. Pour de l'information à ce propos, la fiche d'aide nommée `files` (ouverte avec la commande `help(files)`) est un bon point de départ. Contentons-nous ici d'illustrer l'utilisation de la fonction `dir`, qui permet d'énumérer le contenu d'un répertoire (par défaut le répertoire de travail).

```
dir()
```

```
## [1] "data_ex.csv" "data_ex.json" "data_ex.txt" "data_ex.xlsx"
```

Note : Tous les fichiers de la liste ci-dessus sont disponibles dans le fichier `lecture_ecriture_r_2020.zip`. Il s'agit des fichiers utilisés à la section 2 dans les exemples de lecture de données.

Un bon truc pour trouver facilement le nom complet d'un fichier, qui respecte la syntaxe de R, est d'utiliser la commande suivante :

```
file.choose()
```

Cette commande ouvre une fenêtre pour naviguer dans le système de fichiers et sélectionner un fichier. Une fois la sélection du fichier complétée, la commande `file.choose()` retourne une chaîne de caractères contenant le chemin d'accès complet du fichier sélectionné.

1.3 Projets RStudio

Il est souvent judicieux de rassembler sous un même répertoire un programme R et tous les fichiers qu'il utilise. Les fichiers de données peuvent même être placés dans un sous-répertoire, surtout s'ils sont nombreux.

RStudio offre l'option de transformer un répertoire qui rassemble un ou des programmes R et leurs fichiers associés en « projet ». Des explications à propos de l'utilisation de projets RStudio se trouvent sur la page web suivante : <https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>.

Les projets RStudio facilitent la gestion du répertoire de travail. En ouvrant un projet RStudio, une nouvelle session R est démarrée ayant pour répertoire de travail le répertoire de base du projet.

2 Lecture de fichiers

Une analyse de données en R débute typiquement par l'importation de données. Les données à importer en R peuvent provenir de toutes sortes de sources. Elles peuvent se trouver dans des fichiers sous un certain format, dans des bases de données, ou provenir directement du web. En outre, il existe plusieurs formats de fichiers pour stocker des données : texte (dont CSV), JSON, EXCEL, formats propres à un logiciel statistique particulier, etc. Nous aborderons ici uniquement les formats texte, CSV, JSON, EXCEL, ainsi que les formats propres à R (vus plus loin). La lecture de table HTML sur le web sera aussi introduite. Pour en savoir plus sur les autres possibilités, des références sont fournies en fin de document.

Les données du data frame suivant seront utilisées dans les exemples. Il s'agit de données créées dans les notes sur les [structures de données en R](#), dans lesquelles deux valeurs ont été remplacées par des valeurs manquantes. Ces données se rapportent à une expérience fictive de lancers de dés.

```
de_1 <- c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4)
de_2 <- c(1, 4, 2, 3, 5, 4, 6, 2, 5, 3)
lanceur <- rep(c("Luc", "Kim"), each = 5)
data_ex <- data.frame(de1 = de_1, de2 = de_2, lanceur = lanceur)
# Introduction de valeurs manquantes
data_ex$de2[7] <- NA
data_ex$lanceur[3] <- NA
# Affichage du data frame
data_ex
```

```
##      de1 de2 lanceur
## 1      2  1      Luc
## 2      3  4      Luc
## 3      4  2    <NA>
## 4      1  3      Luc
## 5      2  5      Luc
## 6      3  4      Kim
## 7      5 NA      Kim
## 8      6  2      Kim
## 9      5  5      Kim
## 10     4  3      Kim
```

Avant de lire en R un fichier de données, il est utile de d'abord le visualiser afin de connaître la façon dont les données sont formatées. Cependant, il est important de ne pas modifier le fichier en le visualisant. Pour être absolument certain(e) de ne pas modifier le fichier par inadvertance, une bonne pratique est de faire une copie du fichier et de visualiser cette copie plutôt que le fichier lu en R.

2.1 Fichier texte

Le fichier `data_ex.txt` (partagé dans le fichier `lecture_ecriture_r_2021.zip` au haut de cette page afin que vous puissiez reproduire les exemples proposés) contient les données exemple sous un format texte. Le contenu du fichier a l'allure suivante.

```
de1 de2 lanceur
2   1   Luc
3   4   Luc
4   2   .
1   3   Luc
2   5   Luc
3   4   Kim
5   .   Kim
6   2   Kim
5   5   Kim
4   3   Kim
```

Remarquons que les valeurs manquantes dans ce fichier sont représentées par des points.

Les fonctions les plus simplistes en R pour lire dans des fichiers texte externes sont `readLines` et `scan`. Cependant, ces fonctions ne sont pas les meilleurs outils pour la lecture de tableaux de données.

Comme son nom l'indique, la fonction `readLines` lit un fichier texte ligne par ligne. Par exemple, voici ce que nous obtenons en lisant le contenu du fichier `data_ex.txt` avec la fonction `readLines`.

```
lecture_readLines <- readLines("C:/coursR/data_ex.txt")
lecture_readLines
```

```
## [1] "de1\tde2\tlanceur" "2\t1\tLuc"      "3\t4\tLuc"      "4\t2\t."
## [5] "1\t3\tLuc"          "2\t5\tLuc"      "3\t4\tKim"      "5\t.\tKim"
## [9] "6\t2\tKim"          "5\t5\tKim"      "4\t3\tKim"
```

Le résultat est un vecteur contenant un élément par ligne du fichier lu. Pour chaque ligne, tout son contenu a été laissé dans une seule chaîne de caractère. Nous pouvons constater que le caractère séparant les champs dans le fichier `data_ex.txt` est `\t`, ce qui représente une tabulation.

La fonction `scan` permet quant à elle de lire des données en les considérant toutes du même type et stocke les données lues dans un vecteur ou une liste. Voici le résultat de la lecture du fichier `data_ex.txt` avec `scan`.

```
lecture_scan <- scan("C:/coursR/data_ex.txt", what = "character")
lecture_scan
```

```
## [1] "de1"      "de2"      "lanceur" "2"        "1"        "Luc"      "3"        "4"
## [9] "Luc"      "4"        "2"        "."         "1"        "3"        "Luc"      "2"
## [17] "5"        "Luc"      "3"        "4"        "Kim"      "5"        "."         "Kim"
## [25] "6"        "2"        "Kim"      "5"        "5"        "Kim"      "4"        "3"
## [33] "Kim"
```

Par défaut, `scan` suppose que le caractère séparant les champs dans le fichier lu est un « blanc », soit un ou plusieurs espaces, tabulations, fins de ligne ou retours de chariot. Ce comportement par défaut a bien convenu au fichier `data_ex.txt`. Il resterait cependant encore du travail à faire pour mettre ces données sous un format facilitant leur utilisation en R, par exemple un data frame utilisant les 3 premières valeurs comme noms de variables.

Pour l'importation d'un tableau de données provenant d'un fichier texte, la meilleure fonction offerte dans le R de base est plutôt `read.table` du package `utils` (chargé par défaut à l'ouverture d'une session R). Par exemple, lisons les données contenues dans le fichier `data_ex.txt` avec `read.table`.

```
data_ex_txt <- read.table("C:/coursR/data_ex.txt")
```

Le seul argument obligatoire dans la fonction `read.table` est le chemin d'accès au fichier à lire. Ici (comme dans les exemples précédents), il n'aurait pas été nécessaire de spécifier le chemin d'accès complet du fichier puisque ce fichier se trouve dans le répertoire de travail de la session R. Dans tous les exemples suivants, seul le nom du fichier sera fourni.

Les autres arguments de la fonction `read.table` doivent être adaptés selon le formatage du fichier à lire (voir la [fiche d'aide de la fonction read.table](#)). Dans l'exemple ci-dessus, est-ce que toutes les valeurs par défaut des arguments convenaient vraiment ? Jetons un coup d'œil à l'objet obtenu.

```
data_ex_txt
```

```
##      V1 V2      V3
## 1 de1 de2 lanceur
## 2   2  1      Luc
## 3   3  4      Luc
## 4   4  2        .
## 5   1  3      Luc
## 6   2  5      Luc
## 7   3  4      Kim
## 8   5  .      Kim
## 9   6  2      Kim
## 10  5  5      Kim
## 11  4  3      Kim
```

```
str(data_ex_txt)
```

```
## 'data.frame':   11 obs. of  3 variables:
## $ V1: chr  "de1" "2" "3" "4" ...
## $ V2: chr  "de2" "1" "4" "2" ...
## $ V3: chr  "lanceur" "Luc" "Luc" "." ...
```

Il s'agit d'un data frame. La fonction `read.table` retourne toujours un objet de ce type.

Nous pouvons tout de suite remarquer un problème avec ce data frame. Le nom des variables a été interprété comme une observation. Il faudrait utiliser l'argument `header` pour indiquer que la première ligne du fichier contient les noms des variables.

```
data_ex_txt <- read.table("data_ex.txt", header = TRUE)
str(data_ex_txt)
```

```
## 'data.frame':   10 obs. of  3 variables:
## $ de1      : int  2 3 4 1 2 3 5 6 5 4
## $ de2      : chr  "1" "4" "2" "3" ...
## $ lanceur  : chr  "Luc" "Luc" "." "Luc" ...
```

La première variable, `de1`, a été correctement lu, mais il y a un problème avec la deuxième variable, `de2`. Elle contient des données de type caractères, alors qu'il s'agit d'une variable numérique (le résultat d'un lancer de dé). Afin de pouvoir faire des calculs numériques sur cette variable, elle doit être stockée dans un vecteur (= colonne du data frame) de valeurs numériques. R a cru que les valeurs dans cette colonne étaient de type caractère à cause de la valeur manquante représentée par un point. L'argument `na.strings` de `read.table` indique à R les chaînes de caractères à interpréter comme des valeurs manquantes. Par défaut, `na.strings` prend la valeur "NA". Il faut donc changer la valeur de cet argument comme suit.

```
data_ex_txt <- read.table("data_ex.txt", header = TRUE, na.strings = ".")
str(data_ex_txt)
```

```
## 'data.frame':   10 obs. of  3 variables:
## $ de1      : int  2 3 4 1 2 3 5 6 5 4
## $ de2      : int  1 4 2 3 5 4 NA 2 5 3
## $ lanceur  : chr  "Luc" "Luc" NA "Luc" ...
```

Nous avons réglé le problème de lecture de la deuxième colonne. Attardons-nous maintenant à la dernière colonne. Elle est enregistrée dans un vecteur de chaînes de caractères. Si nous avions préféré que `read.table` transforme les colonnes contenant des valeurs caractères en facteur, il aurait fallu assigner la valeur `TRUE` à l'argument `stringsAsFactors`. Avant sa version 4.0.0, R avait ce comportement par défaut. Le nouveau comportement par défaut correspond à ce que nous désirons ici.

Nous avons illustré deux arguments de la fonction `read.table`. Elle en possède plusieurs autres, à adapter selon le fichier à lire.

2.1.1 Arguments de la fonction `read.table`

Voici un résumé des arguments les plus utiles de la fonction `read.table`. Chaque argument est en lien avec une question que l'utilisateur doit se poser lors de la lecture d'un fichier de données.

- Est-ce que les noms des variables sont sur la première ligne ?
 - Si oui, utiliser `header = TRUE`, sinon `header = FALSE`.
 - Par défaut `header = FALSE`, sauf si la première ligne contient un élément de moins que les lignes suivantes. Dans ce cas, R considère que la première ligne contient les noms des variables et que les lignes suivantes contiennent les observations précédées d'un nom de ligne.
- Quel caractère sépare les champs (différentes valeurs) ?
 - Fournir ce caractère comme valeur à l'argument `sep`.

- Par défaut `sep = ""`, ce qui signifie que le séparateur est un « blanc », soit un ou plusieurs espaces, tabulations, fins de ligne ou retours de chariot.
- Quel est le symbole décimal ?
 - Fournir ce symbole sous forme de chaîne de caractères à l'argument `dec`.
 - Par défaut `dec = "."`.
- Comment sont représentées les valeurs manquantes ?
 - Fournir cette représentation sous forme de chaîne de caractères à l'argument `na.strings`.
 - Par défaut `na.strings = "NA"`.
- Est-ce que certaines valeurs sont encadrées ? Si oui, par quel symbole ?
 - S'il y a des valeurs encadrées, fournir à l'argument `quote` une chaîne de caractère contenant une concaténation de tous les caractères utilisés pour encadrer ces valeurs.
 - Par défaut `quote = "\""'`, ce qui signifie qu'un guillemet double ou simple indique le début ou la fin d'une valeur. La barre oblique inversée (`\`) devant le deuxième `"` dans `"\"'` est nécessaire afin que ce `"` ne soit pas interprété comme la fin de la chaîne de caractères fournie en valeur d'entrée à `quote`.
- Est-ce qu'il y a des caractères spéciaux, par exemple des accents ? Si oui, quel encodage est utilisé ?
 - Un des encodages les plus courants est `UTF-8`. En cas de problèmes avec des accents mal lus, essayez l'argument `encoding = "UTF-8"` ou `fileEncoding = "UTF-8"`. Ça pourrait régler le problème.
 - Par défaut l'encodage est supposé inconnu.
- Est-ce que les données ou les noms des variables débutent à la ligne 1 ?
 - Si ce n'est pas le cas, fournir à l'argument `skip` le nombre de lignes à ne pas lire au début du fichier.
 - Par défaut `skip = 0`, donc aucune ligne n'est sautée.
- Est-ce que les données se terminent à la dernière ligne non vide ?
 - Si ce n'est pas le cas, fournir à l'argument `nrows` le nombre de lignes à lire.
 - Par défaut la lecture des lignes se termine à la fin du fichier.
- Est-ce que le fichier contient des commentaires ? Si oui, quel signe précède ces lignes ?
 - Fournir à `comment.char` un seul caractère, soit celui indiquant que le reste d'une ligne contient un commentaire. Pour empêcher la détection de commentaires, utiliser `comment.char = ""`.
 - Par défaut `comment.char = "#"`.
- Est-ce que les colonnes contenant des chaînes de caractères doivent être converties en facteurs ?
 - Si oui, utiliser `stringsAsFactors = TRUE`, sinon `stringsAsFactors = FALSE`.
 - Par défaut `stringsAsFactors = FALSE` (depuis la version 4.0.0 de R, pour des versions antérieures de R la valeur par défaut est `TRUE`).
- Est-ce que les données dans les différentes colonnes sont toutes du bon type ?
 - Si ce n'est pas le cas, les types désirés peuvent être spécifiés avec l'argument `colClasses`.
 - Par défaut R déduit des types de données en fonction du contenu des colonnes.

Tous les détails ainsi que la description des autres arguments de la fonction `read.table` sont dans la [fiche d'aide de la fonction read.table](#).

2.1.2 Format CSV

Le format CSV est un format texte de données assez usuel. CSV signifie *Comma Separated Values*. En fait, dans un fichier CSV, les valeurs sont séparées par des virgules ou des points-virgules. Deux séparateurs consécutifs signifient qu'une valeur est manquante. Le fichier `data_ex.csv` est un exemple de fichier sous ce format. Il a l'allure suivante.

```
"de1";"de2";"lanceur"  
2;1;"Luc"  
3;4;"Luc"  
4;2;  
1;3;"Luc"  
2;5;"Luc"  
3;4;"Kim"
```

```
5;;"Kim"
6;2;"Kim"
5;5;"Kim"
4;3;"Kim"
```

La fonction `read.csv2` est tout indiquée pour importer ce fichier puisque la valeur par défaut de son argument `sep` est `";"`. Cette fonction appelle en fait la fonction `read.table`, mais avec des valeurs par défaut différentes pour les arguments. Ce type de fonction est appelé fonction enveloppe (en anglais *wrapper*). La fonction `read.table` possède 4 fonctions enveloppe : `read.csv`, `read.csv2`, `read.delim` et `read.delim2`.

Les arguments de la fonction `read.csv2` ont-ils toutes les bonnes valeurs par défaut pour importer le fichier `data_ex.csv` ?

```
data_ex_csv <- read.csv2("data_ex.csv")
str(data_ex_csv)

## 'data.frame':  10 obs. of  3 variables:
## $ de1      : int  2 3 4 1 2 3 5 6 5 4
## $ de2      : int  1 4 2 3 5 4 NA 2 5 3
## $ lanceur: chr  "Luc" "Luc" "" "Luc" ...
```

Ici, la première ligne a correctement été interprétée comme le nom des variables, parce que l'argument `header` de `read.csv2` prend par défaut la valeur `TRUE`. Aussi, la valeur manquante a été bien interprétée pour la variable numérique, mais pas pour la variable catégorique. Utilisons donc l'argument `na.strings` comme suit pour faire correctement l'importation.

```
data_ex_csv <- read.csv2("data_ex.csv", na.strings = "")
str(data_ex_csv)

## 'data.frame':  10 obs. of  3 variables:
## $ de1      : int  2 3 4 1 2 3 5 6 5 4
## $ de2      : int  1 4 2 3 5 4 NA 2 5 3
## $ lanceur: chr  "Luc" "Luc" NA "Luc" ...
```

2.1.3 Fichier texte volumineux

La fonction `read.table` (et ses fonctions enveloppe) arrive en principe à lire n'importe quel fichier texte de données. Elle peut cependant être lente pour lire des fichiers volumineux. Dans ce cas, il est plus pratique de se tourner vers une alternative plus rapide.

2.1.3.1 Package `readr`

Le package `readr` du `tidyverse` offre différentes fonctions pour lire des fichiers textes, selon leur format. Par exemple, il offre la fonction `read_csv2`, similaire à la fonction `read.csv2` du R de base. Voici un exemple d'utilisation de cette fonction.

```
library(readr)
data_ex_csv_readr <- read_csv2("data_ex.csv")
str(data_ex_csv_readr)

## spec_tbl_df[,3] [10 x 3] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ de1      : num [1:10] 2 3 4 1 2 3 5 6 5 4
## $ de2      : num [1:10] 1 4 2 3 5 4 NA 2 5 3
## $ lanceur: chr [1:10] "Luc" "Luc" NA "Luc" ...
## - attr(*, "spec")=
## .. cols(
## ..   de1 = col_double(),
## ..   de2 = col_double(),
```

```
## .. lanceur = col_character()
## .. )
```

L'objet retourné par les fonctions de `readr` est un [tibble](#). Remarquons que la fonction `read_csv2` a, par défaut, correctement interprété les valeurs manquantes et elle n'a pas stocké les données de type caractère dans un facteur. Des attributs ont été ajoutés au tibble pour spécifier les types des colonnes.

2.1.3.2 Package `data.table`

Bien que les fonctions du package `readr` tendent à être plus rapides que la fonction `read.table`, une autre fonction est capable d'être encore plus rapide : la fonction `fread` du package `data.table`. Voici un exemple d'utilisation de cette fonction.

```
library(data.table)
data_ex_csv_fread <- fread("data_ex.csv", na.strings = "")
str(data_ex_csv_fread)

## Classes 'data.table' and 'data.frame':  10 obs. of  3 variables:
## $ de1    : int  2 3 4 1 2 3 5 6 5 4
## $ de2    : int  1 4 2 3 5 4 NA 2 5 3
## $ lanceur: chr  "Luc" "Luc" NA "Luc" ...
## - attr(*, ".internal.selfref")=externalptr>
```

L'objet retourné par la fonction `fread` est un [data table](#). Tout comme la fonction `read.table`, la fonction `fread` possède un argument nommé `na.strings`, que nous avons du spécifier afin que la valeur manquante pour la variable catégorique soit bien interprétée. La fonction `fread` a, comme `read.table` et `read_csv2`, stocké par défaut dans un vecteur les données de types caractères.

2.5 Vérification de l'importation

Une leçon importante à retenir de tous les exemples d'importation présentés dans ce document est qu'il faut toujours vérifier que l'objet R dans lequel nous avons importé des données contient les bonnes données et sous le bon format. Si ce n'est pas le cas, la commande d'importation de données doit être corrigée. Plusieurs tentatives sont parfois nécessaires afin d'arriver à effectuer correctement l'importation.

Voici une liste de vérifications qu'il est toujours bon d'effectuer après une importation en R, dans un data frame, de données sous forme de tableau avec des observations en lignes et des variables en colonnes.

- Est-ce que le nombre de variables (colonnes) dans le data frame est le même que le nombre de variables dans le fichier ?
- Est-ce que le nombre d'observations (lignes) dans le data frame est le même que le nombre d'observations dans le fichier ?
- Est-ce que, pour chacune des variables du jeu de données, le type de données dans le data frame correspond au type de données dans le fichier ?
- Si le jeu de données n'est pas de très petite taille, il est impossible de vérifier une à une toutes les observations. Cependant, pour une observation (ligne) sélectionnée au hasard (ou plus d'une si vous désirez être encore plus rigoureux), est-ce que les valeurs de cette observation dans le data frame sont identiques aux valeurs de cette observation dans le fichier ?
- Est-ce que les données manquantes sont bien représentées dans data frame par la constante `NA`, et ce, dans les colonnes de tout type ?

Si vous répondez oui à toutes ces questions, votre importation de données a de bonnes chances d'avoir été effectuée correctement.

3 Écriture dans des fichiers

En cours d'analyse, il n'est pas rare d'obtenir des résultats dont il est préférable de conserver une copie dans un fichier externe. Ce fichier peut facilement être partagé avec d'autres ou importé dans un autre logiciel pour la poursuite des analyses selon les besoins. Il faut alors demander à R de créer un fichier et d'écrire dans celui-ci.

Exporter les données stockées dans un objet R peut aussi être utile simplement pour aider à les visualiser. Il est vrai que R offre la possibilité de voir le contenu de matrices ou de data frames dans un chiffrier ressemblant à une feuille EXCEL. Ce chiffrier peut être ouvert en appelant la [fonction View](#) ou en cliquant sur un objet dans la sous-fenêtre *Environment* de RStudio. Cependant, pour certaines structures de données non représentables sous forme de tableau, exporter les données dans un fichier externe demeure la meilleure option pour les visualiser.

Pour les exemples à venir, reprenons le data frame `data_ex` et voyons comment exporter son contenu.

3.1 Fichier texte

Le format de fichier le plus universel pour exporter des données est le format texte. Un grand nombre de logiciels peuvent lire ce format. Les fonctions de base en R pour écrire dans un fichier texte sont [writeLines](#), [write](#) et [write.table](#). Les fonctions `writeLines` et `write` sont l'équivalent de `readLines` et `scan`, respectivement, pour l'écriture dans des fichiers au lieu de la lecture. Pour réaliser une tâche d'exportation de jeux de données, la fonction `write.table` est la mieux adaptée des trois. Par exemple, exportons le contenu du data frame `data_ex` dans un fichier texte nommé `data_ex_export.txt` et positionné dans le répertoire de travail.

```
write.table(x = data_ex, file = "data_ex_export.txt", fileEncoding = "UTF-8")
```

Il faut spécifier à la fonction l'objet, typiquement une matrice ou un data frame, dans lequel se trouvent les données à exporter (argument `x`), ainsi que le fichier dans lequel écrire les données (argument `file`). Le fichier peut être spécifié par un chemin d'accès complet ou relatif, comme pour les fonctions de lecture de fichier.

Tout comme `read.table`, la fonction `write.table` possède plusieurs arguments permettant de contrôler le formatage des données. Un de ces arguments, nommé `fileEncoding`, est utilisé dans l'exemple précédent. En fournissant la valeur "UTF-8" à l'argument `fileEncoding`, on crée un fichier d'encodage UTF-8, dont l'usage est très répandu. Sans cet argument, `write.table` pourrait par défaut créer un fichier dont l'encodage est spécifique au système d'exploitation utilisé et qui pourrait causer problèmes sous un autre système d'exploitation. Par exemple, si vous travaillez sous Windows, le fichier créé pourrait contenir des lettres accentuées non reconnues sous macOS en raison d'un encodage mal compris. Spécifier l'argument `fileEncoding = "UTF-8"` permet d'éviter ce genre de souci.

La fonction `write.table` possède d'autres arguments, tel que notamment `dec` pour spécifier le symbole de décimale, `na` pour spécifier le symbole représentant les valeurs manquantes, `sep` pour spécifier le séparateur entre les champs, `row.names` et `col.names` pour indiquer si les noms des lignes et des colonnes, respectivement, doivent être inclus dans le fichier créé. Tous les informations concernant ces arguments, ainsi que tous les autres arguments non mentionnés ici, se trouvent dans la [fiche d'aide de la fonction write.table](#).

Il existe aussi deux fonctions enveloppe à la fonction `write.table` pour les formats CSV, nommées [write.csv](#) et [write.csv2](#).

1.2 Types de données

Les éléments contenus dans les objets atomiques sont des données d'un des types suivants :

- réel,
- entier,
- caractère,
- logique,
- ainsi que quelques autres types qui ne seront pas utilisés dans le cours, tels que « complexe » et « brut ».

Voici un exemple de chacun des types.

1.2.1 Données réelles

Une donnée réelle est un nombre réel, par exemple

```
re <- 5.8  
re
```

```
## [1] 5.8
```

Le terme informatique en anglais pour une donnée réelle est « double ».

```
typeof(re)
```

```
## [1] "double"
```

Par contre, en R, la distinction entre les réels et les entiers est peu utilisée. R simplifie les choses et dit parfois d'une donnée réelle qu'il s'agit d'une donnée numérique.

```
str(re)
```

```
##  num 5.8
```

1.2.2 Données entières

Une donnée entière est un nombre entier (en anglais *integer*), donc sans partie décimale.

```
en <- 1L
```

```
en
```

```
## [1] 1
```

Dans cet exemple, le caractère L après le nombre indique à R que nous désirons qu'il le considère comme un entier.

```
str(en)
```

```
##  int 1
```

```
typeof(en)
```

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

```
is.numeric(en)
```

```
## [1] TRUE
```

```
is.integer(en)
```

```
## [1] TRUE
```

```
is.double(en)
```

```
## [1] FALSE
```

Sans ce caractère L, la majorité des fonctions en R traite par défaut les nombres comme des réels.

```
un <- 1
```

```
un
```

```
## [1] 1
```

```
str(un)
```

```
##  num 1
```

```
typeof(un)
```

```
## [1] "double"
```

```
is.numeric(un)
```

```
## [1] TRUE
```

```
is.integer(un)
```

```
## [1] FALSE
```

```
is.double(un)
```

```
## [1] TRUE
```

1.2.3 Données caractères

Une donnée caractère est une chaîne de caractères.

```
ca <- "Hello world!"  
ca
```

```
## [1] "Hello world!"
```

```
str(ca)
```

```
## chr "Hello world!"
```

```
typeof(ca)
```

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

Ce sont les guillemets (simples ou doubles) qui indiquent à R qu'il s'agit d'une donnée de type caractère.

```
str("1")
```

```
## chr "1"
```

1.2.4 Données logiques

Une donnée logique est simplement TRUE ou FALSE.

```
lo <- TRUE  
lo
```

```
## [1] TRUE
```

```
str(lo)
```

```
## logi TRUE
```

```
typeof(lo)
```

```
## [1] "logical"
```

Attention : "TRUE" est une chaîne de caractères et non une donnée logique, à cause des guillemets bien sûr.

```
str("TRUE")
```

```
## chr "TRUE"
```

1.2.5 Données manquantes

Peu importe le type de données, une donnée manquante est représentée en R par la constante NA (pour « Not Available »).

```
NA
```

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

Ce n'est pas la même chose que NaN qui signifie plutôt « Not a Number ».

```
0/0
```

```
## [1] NaN
```

L'existence d'une constante pour représenter les données manquantes est une autre particularité du langage R spécifique à l'analyse de données.

1.3 Obtention d'informations sur les objets

Pour avoir accès aux informations relatives à un objet, il existe une série de fonctions utiles, par exemple :

- type de l'objet : `is.(vector/matrix/array/list/data.frame/factor/...)`;
- attributs de l'objet : `attributes` (tous les attributs), `attr` (un seul attribut) et les fonctions raccourcies `names`, `dimnames`, `colnames`, `rownames`, `dim`, `nrow`, `ncol`, `class`, `levels`, etc.
- type des éléments : `typeof`, `mode`, `is.(numeric/character/logical/...)`;
- nombre d'éléments : `length`.

Ces fonctions seront illustrées plus loin.

Le tableau suivant résume ce que retournent les fonctions `typeof`, `mode` et `class` pour les différents types d'objets.

type d'objet	<code>typeof</code>	<code>mode</code>	<code>class</code>
vecteur	type des données	type simplifié des données	comme <code>typeof</code>
matrice	type des données	type simplifié des données	"matrix" "array"
array	type des données	type simplifié des données	"array"
liste	"list"	"list"	"list"
data frame	"list"	"list"	"data.frame"
facteur	"integer"	"numeric"	"factor"

Le type simplifié des données retourné par `mode` est explicité dans le tableau suivant, présentant ce que retournent les fonctions `typeof` et `mode` pour les différents types de données.

type de données	<code>typeof</code>	<code>mode</code>
réelles	"double"	"numeric"
entières	"integer"	"numeric"
caractères	"character"	"character"
logiques	"logical"	"logical"

Ainsi, `typeof` distingue les données réelles et entières, alors que `mode` les qualifie toutes de numériques. (Les tableaux précédents sont inspirés de [1].)

1.3.1 Attributs des objets

Les structures de données R peuvent posséder des attributs, qui servent à contenir des informations supplémentaires concernant les données stockées dans l'objet. Ces informations supplémentaires sont parfois appelées métadonnées.

Par exemple, une matrice ou un array possède toujours un attribut portant le nom `dim` contenant les tailles de toutes les dimensions de l'objet.

```
attributes(mat)
```

```
## $dim
## [1] 4 3
```

Le fonction `attributes` retourne une liste contenant tous les attributs d'un objet, nommés. La fonction `attr` permet d'accéder à la valeur d'un attribut particulier, identifié par son nom.

```
attr(mat, "dim")
```

```
## [1] 4 3
```

Pour certains attributs communs, tels que `dim`, il existe même des fonctions spécifiques pour obtenir leur valeur. Par exemple :

- `dim` retourne l'attribut `dim` d'un objet,

```
dim(mat)
```

```
## [1] 4 3
```

- `nrow` retourne la taille de la première dimension d'un objet,

```
nrow(mat)
```

```
## [1] 4
```

- `ncol` retourne la taille de la deuxième dimension d'un objet,

```
ncol(mat)
```

```
## [1] 3
```

Certains types d'objets, tels que le vecteur, ne possèdent pas d'attributs par défaut.

```
attributes(vec)
```

```
## NULL
```

Il est cependant toujours possible d'ajouter des attributs à un objet, portant le nom de notre choix. Les fonctions qui retournent les valeurs d'attributs peuvent aussi servir à initialiser ou remplacer les valeurs des attributs. Pour ce faire, il suffit d'accompagner la commande d'extraction de l'attribut d'un opérateur d'assignation (`<-`) suivi de la valeur souhaitée, comme dans l'exemple ci-dessus.

```
attr(vec, "description") <- "exemple de vecteur"
str(vec)
```

```
##  num [1:4] 3.5 7.8 9.9 5.7
```

```
## - attr(*, "description")= chr "exemple de vecteur"
```

Pour retirer un attribut, il faut lui assigner la valeur spéciale `NULL`.

```
attr(vec, "description") <- NULL
str(vec)
```

```
##  num [1:4] 3.5 7.8 9.9 5.7
```

Vue d'ensemble des attributs possédés par un objet selon son type

Type d'objet	names	dimnames	dim	class	levels
vecteur	(✓)	-	-	•	-
matrice ou array	-	(✓)	✓	•	-
liste	(✓)	-	-	•	-
data frame	(✓)	(✓, •)	•	✓	-
facteur	(✓)	-	-	✓	✓

Légende :

- « - » = attribut non existant pour ce type d'objet
- « ✓ » = attribut existe (non obligatoire si encadré de parenthèses)
- « • » = métadonnée existante, mais pas dans les attributs obtenus avec la fonction `attributes`

6.2 Obtention d'informations

Le data frame `dat` possède les attributs `names`, `row.names` et `class`.

```
attributes(dat)
```

```
## $names
## [1] "V1" "V2" "V3"
##
## $row.names
## [1] 1 2 3 4
##
## $class
## [1] "data.frame"
```

Tout data frame possède ces attributs par défaut. Il s'agit des noms des éléments selon la dimension, ainsi que d'une « classe » identifiant le type de l'objet. Les types d'objets vus précédemment n'ont pas un tel attribut.

En contrepartie, le vecteur des tailles des dimensions est un attribut pour les matrices et les arrays, alors qu'il n'en est pas un pour les data frames. Les fonctions `dim`, `nrow` et `ncol` fonctionnent tout de même avec des data frames.

```
dim(dat)
```

```
## [1] 4 3
```

```
nrow(dat)
```

```
## [1] 4
```

```
ncol(dat)
```

```
## [1] 3
```

Comme pour tout objet R, la longueur d'un data frame est le nombre d'éléments qu'il contient.

```
length(dat)
```

```
## [1] 3
```

Pourquoi la longueur de `dat` n'est-elle pas $4 \times 3 = 12$?

Parce qu'un data frame est un objet récursif dont les éléments sont les colonnes.

```
typeof(dat)
```

```
## [1] "list"
```

Alors, le nombre d'éléments dans un data frame correspond au nombre de colonnes. C'est aussi pour cette raison que les noms des éléments d'un data frame sont aussi considérés comme les noms des colonnes.

```
names(dat)
```

```
## [1] "V1" "V2" "V3"
```

```
colnames(dat)
```

```
## [1] "V1" "V2" "V3"
```

6.6 Extraction d'éléments

L'extraction d'éléments d'un data frame s'effectue comme pour une liste ou encore comme pour une matrice. Les deux options sont possibles.

6.6.1 En traitant le data frame comme une liste

Comme il a déjà été mentionné à quelques reprises maintenant, le data frame est un objet récursif et ses éléments sont ses colonnes. Alors, extraire des éléments d'un data frame signifie extraire des colonnes.

Tous les opérateurs d'indexage introduits plus tôt fonctionnent avec le data frame, comme le démontre ces exemples.

```
des_plus[c("de1", "de2")]
```

```
##      de1 de2
## 11     2  1
## 12     3  4
## 13     4  2
## 14     1  3
## 15     2  5
```

```
## 16    3    4
## 17    5    6
## 18    6    2
## 19    5    5
## 110   4    3

des_plus[[1]]

## [1] 2 3 4 1 2 3 5 6 5 4

des_plus$de1

## [1] 2 3 4 1 2 3 5 6 5 4
```

6.6.2 En traitant le data frame comme une matrice

Il est aussi possible d'exploiter les deux dimensions du data frame et d'extraire des éléments de ses colonnes. Pour ce faire, il faut simplement traiter le data frame comme une matrice et fournir deux arguments à l'opérateur d'indigage [ou [[: le premier pour identifier des lignes, le deuxième pour identifier des colonnes. Voici quelques exemples.

```
des_plus[, "de1"]

## [1] 2 3 4 1 2 3 5 6 5 4

des_plus[c(rep(FALSE, 6), rep(TRUE, 3), FALSE), 2:3]

##      de2 lanceur
## 17     6      Kim
## 18     2      Kim
## 19     5      Kim

des_plus[[1, 1]]

## [1] 2
```

6.6.3 Extraction par exclusion

Comme pour une liste, assigner la valeur NULL à un élément d'un data frame l'efface de celui-ci, comme dans cet exemple.

```
autre$jour <- NULL
autre

##      de1 de2 lanceur
## 11     2   1      Luc
## 12     3   4      Luc
## 13     4   2      Luc
## 14     1   3      Luc
## 15     2   5      Luc
## 16     3   4      Kim
## 17     5   6      Kim
## 18     6   2      Kim
## 19     5   5      Kim
## 110    4   3      Kim
```

8 Conversions de type de données

Rappelons-nous que tous les éléments d'un objet atomique doivent être du même type. Que se passe-t-il alors si nous tentons de modifier un élément d'un vecteur pour le remplacer par une donnée d'un type différent du type des éléments d'origine dans le vecteur ?

```
de <- c(2, 3, 4, 1, 2, 3, 5, 6, 5, 4)
str(de)
```

```
##  num [1:10] 2 3 4 1 2 3 5 6 5 4
```

```
de[1] <- "2"
str(de)
```

```
##  chr [1:10] "2" "3" "4" "1" "2" "3" "5" "6" "5" "4"
```

Dans l'exemple ci-dessus, tous les éléments, à l'origine réels, sont devenus des chaînes de caractères, soit le type de la nouvelle donnée.

Voici un autre exemple.

```
de[1] <- FALSE
str(de)
```

```
##  chr [1:10] "FALSE" "3" "4" "1" "2" "3" "5" "6" "5" "4"
```

Cette fois, c'est le type de la nouvelle donnée qui a été modifié pour être conforme aux éléments déjà présents dans le vecteur. La nouvelle donnée, à l'origine logique, a été transformée en chaîne de caractères.

Les exemples précédents illustrent la conversion implicite de types. Ces conversions sont qualifiées d'« implicites », car l'utilisateur n'a pas indiqué clairement à R ce qu'il doit faire. Il n'est pas possible de stocker dans un vecteur des éléments de types différents. Ce qui était demandé à R dans ces exemples était donc en théorie incorrect. Plusieurs langages de programmation génèrent une erreur suite à une telle tentative d'opération.

Cependant, R étant conçu pour des gens qui s’y connaissent parfois peu en informatique, il ne génère pas d’erreur. Il arrange les choses pour l’utilisateur. Il fait une conversion de type des données sans même imprimer un message d’avertissement. Lorsqu’il doit choisir entre deux types de données, il opte toujours pour le type le moins contraignant des deux. Pour les types que nous utiliserons dans ce cours, leur classement du moins contraignant au plus contraignant est le suivant :

caractère > réel > entier > logique.

8.1 Conversions explicites

Afin de s’assurer de garder le contrôle de nos objets, il est aussi possible de réaliser des conversions explicites avec les fonctions `as.(character/numeric/double/integer/logical/...)`. Une conversion est dite « explicite » lorsque l’utilisateur indique à R vers quel type faire la conversion.

Exemple : Comment créer un vecteur d’entiers ?

- l’opérateur : crée par défaut des vecteurs d’entiers lorsque la valeur de départ est un entier

```
vect_1 <- 1:3
str(vect_1)
```

```
## int [1:3] 1 2 3
```

- la fonction `as.integer` effectue la conversion vers le type entier

```
vect_2 <- c(1, 2, 3)
str(vect_2)
```

```
## num [1:3] 1 2 3
```

```
vect_3 <- as.integer(vect_2)
str(vect_3)
```

```
## int [1:3] 1 2 3
```

- le caractère L placé tout de suite après un nombre entier (sans espace entre le nombre et L) est un indicateur du type entier

```
vect_4 <- c(1L, 2L, 3L)
str(vect_4)
```

```
## int [1:3] 1 2 3
```

Les conversions implicites de type permettent d’éviter des erreurs. Voici une assignation qui retourne le résultat escompté en R grâce aux conversions implicites de type, mais qui ne retournerait pas le résultat escompté dans certains autres langages.

```
x <- c(1L, 4L) # x est d'abord créé avec le type entier
typeof(x)
```

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

```
x[1] <- 1/2
x
```

```
## [1] 0.5 4.0
```

```
typeof(x)
```

```
## [1] "double"
```

L'objet `x` a changé le type de ses éléments afin de pouvoir garder en mémoire correctement la nouvelle donnée assignée à son premier élément. Dans d'autres langages, par exemple en C, la donnée `0.5` aurait été tronquée à `0`.

Conversions explicites de chaînes de caractères vers des nombres

Il est même possible de convertir certains caractères en nombres. Lorsque la chaîne de caractères contient en fait des caractères numériques utilisant le point comme signe décimal, la conversion est directe avec `as.numeric` (ou `as.double` ou `as.integer`).

```
vect_5 <- c("3.5", "4.6", "7")
as.numeric(vect_5)
```

```
## [1] 3.5 4.6 7.0
```

Pour les cas particuliers, il faut plutôt utiliser la fonction `type.convert`.

```
vect_6 <- c("3,5", "4,6", "-")
as.numeric(vect_6)
```

```
## Warning: NAs introduced by coercion
```

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

```
type.convert(vect_6, na.strings = "-", dec = ",")
```

```
## [1] 3.5 4.6 NA
```

1.1 Modification ou ajout de variables

1.1.1 Opérateurs d'indexage et fonction `transform`

Pour remplacer une variable dans un data frame par une transformation de celle-ci, il suffit d'assigner le nouveau vecteur ou facteur contenant les observations de la variable transformée à la colonne contenant la variable à modifier.

Par exemple, dans le jeu de données `cars`, la distance est exprimée en pieds. Supposons que nous voulons transformer l'échelle de mesure de cette variable pour des mètres. Cette transformation est effectuée par une

simple opération mathématique : la multiplication par un facteur de conversion. Ce facteur de conversion est ici 0.3048, car un pied est l'équivalent de 0.3048 mètre.

```
cars$dist * 0.3048
```

Pour modifier la variable sans en créer une nouvelle, il suffit d'assigner le vecteur contenant les données pour la variable transformée à la colonne qui contenait la variable d'origine.

```
cars$dist <- cars$dist * 0.3048
```

Nous savons que l'instruction `cars$dist`, servant à identifier la colonne à modifier, aurait pu être écrite de bien d'autres façons, notamment :

- `cars[["dist"]]`
- `cars[, "dist"]`
- `cars[, 2]`
- etc.

Je considère cependant que l'opérateur `$` procure la syntaxe la plus simple pour un remplacement de variable dans un data frame. Le jeu de données `cars` a été modifié comme suit.

```
str(cars)
```

```
## 'data.frame':   50 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  0.61 3.05 1.22 6.71 4.88 ...
```

La [fonction `transform`](#) offre une autre façon d'écrire ce genre d'assignation. La commande suivante produit le même résultat que la dernière assignation.

```
cars <- transform(cars, dist = dist * 0.3048)
```

Il faut donner comme premier argument à la fonction `transform` le data frame dans lequel nous souhaitons effectuer une transformation de variable. Les arguments suivants sont des assignations de nouvelles valeurs. Dans ces assignations, il est possible de référer aux colonnes du data frame fourni comme premier argument par leur nom sans les précéder de `nom_du_data_frame$` et sans les encadrer de guillemets, comme avec la [fonction `subset`](#), vue dans le [cours sur les structures de données en R](#).

Remarque concernant la présence de deux objets nommés `cars` dans le chemin de recherche

Nous avons maintenant deux objets nommés `cars` dans notre chemin de recherche, parce que nous venons de modifier un data frame provenant d'un package chargé. Le data frame modifié, est situé dans notre environnement de travail. Alors que le data frame `cars` original est toujours dans l'environnement du package `datasets`. Celui-là n'a pas été modifié.

Le chemin de recherche de notre session R, pour une commande soumise dans la console, est le suivant.

```
search()
```

```
## [1] ".GlobalEnv"          "package:data.table" "package:tidyr"      "package:dplyr"
## [5] "tools:rstudio"       "package:stats"      "package:graphics"   "package:grDevices"
## [9] "package:utils"       "package:datasets"   "package:methods"    "Autoloads"
## [13] "package:base"
```

Lorsque nous demandons à R d'accéder à l'objet nommé `cars`, il cherche ce nom dans les environnements de son chemin de recherche, un environnement à la fois, en respectant l'ordre des environnements dans la liste précédente. L'environnement `".GlobalEnv"`, soit notre environnement de travail, est le premier de la liste. Lorsque R trouve le nom, il cesse sa recherche. Ainsi, toute commande dans la console contenant le nom `cars`, accèdera maintenant à l'objet nommé `cars` de notre environnement de travail.

```
str(cars)

## 'data.frame':    50 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  0.186 0.929 0.372 2.044 1.486 ...
```

Il est possible de forcer R à aller chercher l'objet `cars` dans le package `datasets` avec l'opérateur `::`.

```
str(datasets::cars)

## 'data.frame':    50 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

Nous reviendrons sur le fonctionnement des évaluations en R dans un autre cours.

Ajout plutôt que modification de variables Au lieu d'écraser une colonne d'un data frame en la remplaçant par un autre objet, comme nous venons de le faire, il sera souvent préférable d'ajouter les nouvelles variables créées tout en conservant les anciennes. Dans ce cas, il suffit de concaténer les nouvelles variables au data frame avec la fonction `cbind` ou `data.frame` (vues plus loin), ou encore d'assigner les nouvelles variables à des colonnes portant des noms qui ne sont pas initialement présents dans le data frame. Par exemple, pour ajouter une variable contenant la distance exprimée en mètres au data frame `cars`, nous pourrions procéder comme suit.

```
cars <- datasets::cars # pour repartir avec la version originale du jeu de données
cars$dist_m <- cars$dist * 0.3048
head(cars)
```

```
##   speed dist dist_m
## 1     4    2 0.6096
## 2     4   10 3.0480
## 3     7    4 1.2192
## 4     7   22 6.7056
## 5     8   16 4.8768
## 6     9   10 3.0480
```

Étant donné que le jeu de données `cars` ne contenait pas à l'origine de colonne nommée `dist_m`, une nouvelle colonne portant ce nom a été créée et ajoutée après les colonnes déjà présentes dans le data frame.

La fonction `transform` permet aussi d'ajouter de nouvelles variables par une assignation à un nouveau nom de colonne.

Remarque : Il y a souvent plusieurs façons de réaliser une même tâche en R. Personnellement, je tends à adopter la façon qui présente le meilleur compromis entre un code le plus facile à comprendre possible et un code le plus court possible. Ici, l'assignation à un nouveau nom de colonne avec l'opérateur `$` me semble la meilleure option, mais à vous de choisir la méthode que vous préférez.

1.1.2 Fonction `mutate` du package `dplyr`

Pour les adeptes du `tidyverse`, le package `dplyr` propose, entre autres, la fonction `mutate` pour modifier ou créer de nouvelles variables. L'opération précédente aurait pu être effectuée comme suit avec cette fonction, dont l'utilisation ressemble à celle de la fonction `transform`.

```
cars <- mutate(cars, dist_m = dist * 0.3048)
```

1.1.3 Opérateur `:=` du package `data.table`

Si le data frame contenant les données à manipuler est d'abord transformé en data table, déjà mentionné dans les notes sur les [structures de données](#), l'opérateur d'assignation `:=` offert par le package `data.table`

peut ensuite être utilisé pour ajouter, modifier ou même effacer des variables dans les données. Cet opérateur s'utilise en conjonction avec l'opérateur, `[` comme dans l'exemple suivant.

```
cars_dt <- as.data.table(datasets::cars)
cars_dt[, dist_m := dist * 0.3048]
str(cars_dt)
```

```
## Classes 'data.table' and 'data.frame':  50 obs. of  3 variables:
## $ speed : num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist  : num  2 10 4 22 16 10 18 26 34 17 ...
## $ dist_m: num  0.61 3.05 1.22 6.71 4.88 ...
## - attr(*, ".internal.selfref")=<externalptr>
```

Avec `:=`, aucune copie de l'objet n'est effectuée pour ensuite être réassignée. L'objet est modifié par référence (donc « sur place »), ce qui n'est pas le cas pour toutes les autres façons de faire présentées précédemment. La modification d'objets par référence est une des techniques utilisées par le package `data.table` pour optimiser ses temps de traitement.

1.2 Variable conditionnelle à une autre avec la fonction `ifelse`

La fonction `ifelse` est souvent utile pour transformer des variables. Cette fonction prend comme premier argument un vecteur de valeurs logiques. Cet argument est nommé `test`. Elle retourne en sortie un vecteur de même longueur que `test`. Ce vecteur est formé des valeurs dans le deuxième argument (`yes`) aux positions pour lesquelles `test` comporte une valeur `TRUE`, et des valeurs dans le troisième argument (`no`) aux autres positions. Voici un petit exemple pour comprendre le fonctionnement de cette fonction.

```
ifelse(
  test = c(TRUE, FALSE, FALSE, TRUE),
  yes = c(1, 2, 3, 4),
  no = c(-1, -2, -3, -4)
)
```

```
## [1]  1 -2 -3  4
```

Par exemple, supposons que nous souhaitons créer une nouvelle variable dans le jeu de données `cars` contenant la vitesse en km/heure plutôt qu'en miles/heure, mais dans laquelle les valeurs de distance en miles à l'heure inférieures à 10 sont ramenées à des valeurs manquantes. Nous pourrions créer cette variable comme suit (1.60934 est le facteur de conversion entre un mile à l'heure et un kilomètre à l'heure).

```
cars$speed_km <- ifelse(test = cars$speed < 10, yes = NA, no = cars$speed * 1.60934)
head(cars, n = 10)
```

```
##   speed dist  dist_m speed_km
## 1     4    2  0.6096      NA
## 2     4   10  3.0480      NA
## 3     7    4  1.2192      NA
## 4     7   22  6.7056      NA
## 5     8   16  4.8768      NA
## 6     9   10  3.0480      NA
## 7    10   18  5.4864 16.09340
## 8    10   26  7.9248 16.09340
## 9    10   34 10.3632 16.09340
## 10   11   17  5.1816 17.70274
```

Dans cet exemple, la valeur fournie au deuxième argument (`yes`) n'est pas un vecteur de même longueur que les valeurs fournies aux arguments `test` et `no`. R applique donc la règle de recyclage pour transformer `NA`, qui est en fait un vecteur de longueur 1, en un vecteur contenant `length(cars$speed)= 50` valeurs `NA`.

Il est même possible d'imbriquer plusieurs appels à la fonction `ifelse`, comme dans cet exemple.

```
cars$speed_cat <- ifelse(
  test = cars$speed < 10,
  yes = "moins de 10",
  no = ifelse(
    test = cars$speed >= 20,
    yes = "20 ou plus",
    no = "entre 10 et 20"
  )
)
str(cars)
```

```
## 'data.frame': 50 obs. of 5 variables:
## $ speed : num 4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num 2 10 4 22 16 10 18 26 34 17 ...
## $ dist_m : num 0.61 3.05 1.22 6.71 4.88 ...
## $ speed_km : num NA NA NA NA NA ...
## $ speed_cat: chr "moins de 10" "moins de 10" "moins de 10" "moins de 10" ...
```

Voici un extrait du jeu de données pour mieux comprendre de quoi a l'air la nouvelle variable créée.

```
cars[c(5:7,38:40), c("speed", "speed_cat")]
```

```
##      speed      speed_cat
## 5         8      moins de 10
## 6         9      moins de 10
## 7        10  entre 10 et 20
## 38       19  entre 10 et 20
## 39       20      20 ou plus
## 40       20      20 ou plus
```

Nous avons catégorisé la variable `speed` en 3 catégories. Il existe cependant de meilleures façons de faire ce genre de tâche, présentées dans la sous-section suivante.

Notons, avant de clore cette sous-section, que le package `dplyr` offre une fonction de remplacement à `ifelse`, nommée `if_else`, tout comme le package `data.table` avec la fonction `fifelse`. Ces fonctions ont le potentiel d'être plus rapides que `ifelse`.

1.5 Manipulation de chaînes de caractères

D'un langage de programmation à l'autre, les chaînes de caractères ne sont pas manipulées de la même façon. Par exemple, si un objet R contient une chaîne de caractères telle la suivante,

```
chaine <- "Bonjour"
```

il n'est pas possible d'extraire disons la deuxième lettre de la chaîne avec la commande suivante.

```
chaine[2]
```

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

Cette commande aurait fonctionné en langage Python. Elle ne fonctionne pas en R, parce que l'objet `chaine` est un vecteur de longueur 1. L'opérateur d'indexage `[` réfère à des éléments d'un objet. L'objet `chaine` ne contient qu'un seul élément, qui est une chaîne de caractères.

```
str(chaine)
```

```
## chr "Bonjour"
```

```
length(chaine)
```

```
## [1] 1
```

```
chaine[1]
```

```
## [1] "Bonjour"
```

Cette section présente quelques fonctions R de base pour manipuler des vecteurs dont les éléments sont des chaînes de caractères. Toutes ces fonctions travaillent de façon vectorielle. Elles effectuent d'un coup une opération sur tous les éléments d'un vecteur.

Ces fonctions seront illustrées en utilisant un data frame nommé `boxoffice` créé le 25 janvier 2019 en important le contenu de la page web http://pro.boxoffice.com/numbers/all_time. Cette page web n'existe plus aujourd'hui, car le site web Boxoffice Pro a subi une refonte. Utilisons tout de même ces données, même si elles ne sont plus à jour, puisque leur format est parfait pour illustrer la manipulation de chaînes de caractères et de dates en R.

Une copie des données se trouve dans le haut de [la page web des notes que vous êtes en train de lire](#). Le fichier contenant les données est en format `.rds`, il se nomme `boxoffice_20190125.rds`. Il se trouve dans le fichier compressé `pretraitement_donnees_r_2021.zip`. Après avoir téléchargé ce fichier sur votre ordinateur, et après l'avoir décompressé pour en extraire `boxoffice_20190125.rds`, vous pouvez charger les données en R avec le code suivant (en spécifiant le bon chemin d'accès) :

```
boxoffice <- readRDS("C:/coursR/boxoffice_20190125.rds")
```

Le data frame `boxoffice` utilisé ici ressemble à celui créé dans les [notes sur la lecture et l'écriture dans des fichiers externes à partir de R](#), mais il contient des données moins récentes et présentées sous un format un peu différent.

```
head(boxoffice, n = 3)
```

```
##      X1                                X2          X3          X4
## 1  1 Star Wars: The Force Awakens (Disney) Dec 18, 2015 $936,662,225
## 2  2                                Avatar (Fox) Dec 18, 2009 $760,507,625
## 3  3                                Black Panther (Disney) Feb 16, 2018 $700,059,566
```

Les noms des colonnes du data frame ne sont pas informatifs. Renommons les colonnes du data frame en reprenant les noms de colonnes qui apparaissaient dans la page web d'où ses données proviennent (en remplaçant les espaces par des tirets bas et en omettant les parenthèses).

```
colnames(boxoffice) <- c("Ranking", "Movie_Distributor", "Release_Date", "Gross")
```

Il s'agit de données concernant les recettes (**Gross**) des films ayant généré le plus de revenus dans les cinémas américains.

1.5.1 Fonction paste

La [fonction paste](#) sert à concaténer des éléments de vecteurs. Elle retourne toujours un résultat sous le format caractère. Elle n'est pas illustrée ici, car elle a été présentée dans les [notes sur les structures de données en R](#).

Rappelons que la fonction `paste0` est en quelque sorte une version de la fonction `paste` avec l'argument `sep = ""` (aucun caractère comme séparation entre les chaînes de caractères à concaténer).

1.5.2 Fonction nchar

La [fonction nchar](#) permet de compter le nombre de caractères dans des chaînes de caractères. Elle a aussi été illustrée dans les [notes sur les structures de données en R](#).

1.5.3 Fonctions toupper, tolower et casefold

Les [fonctions toupper et tolower](#) transforment tous les caractères alphabétiques en majuscules et en minuscules respectivement.

```
extrait <- head(boxoffice$Movie_Distributor, n = 2)
extrait
```

```
## [1] "Star Wars: The Force Awakens (Disney)" "Avatar (Fox)"
```

```
toupper(extrait)
```

```
## [1] "STAR WARS: THE FORCE AWAKENS (DISNEY)" "AVATAR (FOX)"
```

```
tolower(extrait)
```

```
## [1] "star wars: the force awakens (disney)" "avatar (fox)"
```

La [fonction casefold](#), utilisée comme exemple de fonction dans les [notes sur les concepts de base en R](#), permet de faire les deux transformations de casse, soit transformer tous les caractères alphabétiques en majuscules (avec l'argument `upper = TRUE`) ou transformer tous les caractères alphabétiques en minuscules (avec l'argument `upper = FALSE`).

```
casefold(extrait, upper = TRUE)
```

```
## [1] "STAR WARS: THE FORCE AWAKENS (DISNEY)" "AVATAR (FOX)"
```

```
casefold(extrait, upper = FALSE)
```

```
## [1] "star wars: the force awakens (disney)" "avatar (fox)"
```

1.5.4 Fonction `strsplit`

La fonction `strsplit` sépare des chaînes de caractères en sous-chaînes de caractères en coupant lors de la rencontre d'une certaine série de caractères. Tentons d'utiliser cette fonction pour séparer le nom du film du nom du distributeur, qui se retrouvent tous deux dans la colonne `Movie_Distributor` du data frame `boxoffice`. Demandons à `strsplit` de briser le contenu de cette colonne en deux parties : ce qui se trouve avant la parenthèse ouvrante et ce qui se trouve après. Étant donné que le nom du distributeur est toujours placé à la fin de la chaîne de caractères, entre parenthèses, nous réussirons ainsi à isoler les deux éléments.

```
res_split <- strsplit(x = boxoffice$Movie_Distributor, split = "(", fixed = TRUE)
head(res_split, n = 2)
```

```
## [[1]]
## [1] "Star Wars: The Force Awakens " "Disney)"
##
## [[2]]
## [1] "Avatar " "Fox)"
```

Par défaut, la fonction considère que la valeur donnée à l'argument `split` est une [expression régulière](#). Les expressions régulières ne seront pas vues ici (mais des références pour en apprendre davantage à ce sujet sont fournies dans les références). Elles constituent un moyen puissant pour effectuer des recherches dans des chaînes de caractères. L'argument `fixed = TRUE` demande à `strsplit` de ne pas considérer l'argument `split` comme une expression régulière, mais bien de traiter la chaîne de caractères telle qu'elle a été fournie.

Le résultat de la fonction `strsplit` est toujours une liste de la même longueur que le vecteur assigné à l'argument `x`. Chaque élément de la liste est un vecteur comprenant les sous-chaînes de caractères obtenues de la séparation. Ici, ces vecteurs ont le plus souvent une longueur de 2. Par exemple pour le premier film du jeu de données, *Star Wars : The Force Awakens* (Disney), on obtient le vecteur `["Star Wars: The Force Awakens ", "Disney)"]`. Par contre, certains titres de film comprennent un élément entre parenthèses de plus, par exemple *"The Lion King (1994) (Disney)"*. Pour eux, la longueur du vecteur est 3.

La commande suivante permet d'extraire le dernier élément de chacun des vecteurs se trouvant dans `res_split`. Elle utilise la fonction `sapply` vue dans le cours sur les [calculs par sections d'un objet R](#).

```
distributeur <- sapply(res_split, tail, n = 1)
head(distributeur, n = 2)
```

```
## [1] "Disney)" "Fox)"
```

1.5.5 Fonction `substr`

Afin d'avoir vraiment isolé le nom du distributeur, il ne reste plus qu'à retirer la parenthèse fermante à la fin des chaînes de caractères dans `distributeur`. La fonction `substr` permet d'extraire une partie de chaînes de caractères en spécifiant les positions, dans les chaînes, des caractères à conserver.

```
distributeur <- substr(x = distributeur, start = 1, stop = nchar(distributeur) - 1)
head(distributeur, n = 2)
```

```
## [1] "Disney" "Fox"
```

Ici nous voulons extraire les caractères de la première position jusqu'à l'avant-dernière position. Celle-ci varie d'une observation à l'autre, c'est pourquoi nous utilisons l'expression `nchar(distributeur) - 1` pour la calculer.

Notons que nous aurions aussi pu effectuer ce traitement avec la fonction `strsplit`, en utilisant `)` comme point de coupure, ou encore avec la fonction `sub`, comme dans l'exemple suivant.

1.5.6 Fonctions `sub` et `gsub`

Voyons maintenant comment il est possible de remplacer une sous-chaîne de caractères par une autre. Dans les données `boxoffice`, les recettes des films comprennent un signe de dollar au début, ainsi que des virgules pour séparer les milliers. À cause de ces caractères, cette colonne ne peut pas être transformée sous un format numérique directement. Nous allons donc retirer ces caractères en les remplaçant par des chaînes de caractères vides. Ensuite, nous pourrions convertir le vecteur au format numérique plutôt que caractère.

```
recettes <- boxoffice$Gross
str(recettes)

## chr [1:1000] "$936,662,225" "$760,507,625" "$700,059,566" "$678,815,482" ...

recettes <- sub(pattern = "$", replacement = "", x = recettes, fixed = TRUE)
str(recettes)

## chr [1:1000] "936,662,225" "760,507,625" "700,059,566" "678,815,482" "658,672,302" ...

recettes <- gsub(pattern = ",", replacement = "", x = recettes, fixed = TRUE)
str(recettes)

## chr [1:1000] "936662225" "760507625" "700059566" "678815482" "658672302" "652270625" ...

recettes <- as.numeric(recettes)
str(recettes)

## num [1:1000] 9.37e+08 7.61e+08 7.00e+08 6.79e+08 6.59e+08 ...
```

Il est maintenant possible de faire des calculs numériques sur les recettes des films, puisqu'elles sont stockées sous un format numérique.

Les fonctions `sub` et `gsub` servent donc à chercher les occurrences d'un « motif » (argument `pattern`) dans des chaînes de caractères (argument `x`), puis à remplacer ces occurrences par un autre motif (argument `replacement`). La fonction `sub` remplace seulement la première occurrence de `pattern`, alors que la fonction `gsub` remplace toutes les occurrences. Tout comme la fonction `strsplit`, les fonctions `sub` et `gsub` travaillent par défaut avec des motifs exprimés sous forme d'expressions régulières. L'argument `fixed = TRUE` empêche ce comportement et implique le traitement des motifs comme des chaînes de caractères.

1.5.7 Fonctions `grep` et `grepl`

Supposons que nous désirions avoir une variable logique prenant la valeur `TRUE` si la compagnie Disney a distribué le film, `FALSE` sinon. Nous pourrions utiliser la fonction `grepl` comme suit pour obtenir cette variable.

```
Disney <- grepl(pattern = "Disney", x = distributeur, fixed = TRUE)
sum(Disney)
```

```
## [1] 161
```

Ainsi, 161 des 1000 films du jeu de données `boxoffice` ont été distribués par Disney.

Les fonctions `grep` et `grepl` identifient les éléments d'un vecteur de chaînes de caractères (argument `x`) contenant un certain « motif » (argument `pattern`). Pour ne pas que ce motif soit traité comme une expression régulière, il faut fournir la valeur `TRUE` à l'argument `fixed`. La fonction `grep` retourne un vecteur numérique contenant les positions dans `x` des éléments possédant le motif. Comme nous l'avons constaté dans l'exemple précédent, la fonction `grepl` retourne pour sa part un vecteur logique aussi long que `x`, contenant la valeur `TRUE` pour les éléments possédant le motif, `FALSE` sinon.

Les fonctions `sub`, `gsub`, `grep` et `grepl` sont toutes documentées dans la même fiche d'aide : [celle de la fonction `grep`](#). Quelques autres fonctions de manipulation de chaînes de caractères sont documentées dans cette fiche, mais ne sont pas présentées ici.

2.1 Omission d'observations

2.1.1 Retrait des observations dupliquées

Il est parfois utile de gérer les observations dupliquées dans un jeu de données. En R, les fonctions suivantes sont utiles avec des observations dupliquées :

- pour tester la présence d'observations dupliquées : `duplicated`,
- pour retirer les observations dupliquées : `unique`.

Une observation est ici définie par l'ensemble des valeurs observées de toutes les variables pour un individu (ou une unité) de la population statistique à l'étude. Donc une observation dupliquée est une ligne répétée (donc deux lignes ou plus complètement identiques) dans une matrice ou un data frame. Dans le cas d'une seule variable, stockée dans un vecteur, une observation dupliquée est une valeur présente plus d'une fois dans le vecteur.

Dans le jeu de données `data_ex`, les lignes 1 et 5 de sont identiques. La fonction `duplicated` identifie la 5e observation comme une duplication d'une autre observation.

```
duplicated(data_ex)
```

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

L'observation peut être retirée avec la fonction `unique` comme suit.

```
unique(data_ex)
```

```
##   de1 de2 lanceur
## 1   2   1     Luc
## 2   3   4     Luc
## 3   4   2  <NA>
## 4   1   3     Luc
## 6   3   4     Kim
## 7   5  NA     Kim
## 8   6   2     Kim
## 9   5   5     Kim
## 10  4   3     Kim
```

Si elles reçoivent un vecteur en entrée, les fonctions `duplicated` et `unique` réagissent comme suit.

```
duplicated(c(1, 3, 2, 1, 2, 1))
```

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

```
unique(c(1, 3, 2, 1, 2, 1))
```

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

2.1.2 Retrait des observations avec données manquantes

Comme pour les observations dupliquées, il existe des outils en R pour identifier ou retirer les observations contenant des données manquantes :

- pour identifier les observations sans données manquantes : `complete.cases`,
- pour retirer les observations contenant au moins une donnée manquante : `na.omit`.

Dans le jeu de données `data_ex`, les lignes 3 et 7 contiennent chacune une donnée manquante. La fonction `complete.cases` identifie ces lignes comme n'étant pas des cas complets (`FALSE` retourné pour les lignes contenant des NA).

```
complete.cases(data_ex)
```

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

Ces deux observations peuvent facilement être retirées avec la fonction `na.omit` comme suit.

```
na.omit(data_ex)
```

```
##   de1 de2 lanceur
## 1   2   1     Luc
## 2   3   4     Luc
## 4   1   3     Luc
## 5   2   1     Luc
## 6   3   4     Kim
## 8   6   2     Kim
## 9   5   5     Kim
## 10  4   3     Kim
```

La fonction `na.omit` a déjà été vue dans les [notes sur les calculs statistiques en R](#)). Cette fonction ajoute des attributs à l'objet retourné pour identifier les observations omises.

```
data_ex_no_NA <- na.omit(data_ex)
str(data_ex_no_NA)
```

```
## 'data.frame':   8 obs. of  3 variables:
## $ de1      : num  2 3 1 2 3 6 5 4
## $ de2      : num  1 4 3 1 4 2 5 3
## $ lanceur: chr  "Luc" "Luc" "Luc" "Luc" ...
## - attr(*, "na.action")= 'omit' Named int [1:2] 3 7
## ..- attr(*, "names")= chr [1:2] "3" "7"
```

Les fonctions `complete.cases` et `na.omit` fonctionnent aussi avec des vecteurs, comme le montre ces exemples.

```
complete.cases(c(1, 3, 2, NA, 2, 1)) # équivaut à !is.na(c(1, 3, 2, NA, 2, 1))
```

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

```
na.omit(c(1, 3, 2, NA, 2, 1))
```

```
## [1] 1 3 2 2 1
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"
```

La fonction `is.na` (vue dans les [notes sur les calculs mathématiques en R](#)), utilisée avec l'opérateur `[`, permet aussi de retirer les données manquantes d'un vecteur.

```
x <- c(1, 3, 2, NA, 2, 1)
x[!is.na(x)]
```

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

Remarquons que cette solution de rechange à `na.omit` n'ajoute pas d'attributs au résultat.

2.2 Sélection de sous-ensembles de données

Sélectionner un sous-ensemble de données signifie **extraire** d'un jeu de données une partie des observations (lignes) et/ou des variables (colonnes) qu'il contient. La sélection d'observations répondant à certains critères est aussi parfois appelée **filtrage** de données. La matière couverte dans les notes sur les calculs de base en R concernant les [conditions logiques](#) nous sera très utile pour tester des critères.

2.2.1 Opérateur `[` et fonction `subset`

Les principaux outils permettant d'extraire des sous-ensembles de données stockées dans un data frame ont déjà été présentés dans les notes sur les structures de données en R. Il s'agit de [l'opérateur d'indexage `\[` et de la fonction d'extraction `subset`](#). La fonction `subset` a d'ailleurs été utilisée dans les derniers exemples pour sélectionner seulement certaines variables dans l'affichage des résultats.

Pour illustrer l'utilisation de ces outils dans un contexte de sélection de sous-ensembles de données, voyons comment conserver du jeu de données `data_ex` seulement les résultats de lancers de dés effectués par Kim et ne comprenant aucune donnée manquante. Dans ce sous-ensemble, la variable `lanceur` prendra toujours la même valeur. Elle ne sera donc plus vraiment pertinente et sera retirée.

Tout d'abord, voyons comment effectuer cette tâche avec l'opérateur `[`, étape par étape. Pour limiter les observations à celles pour lesquelles la variable `lanceur` prend la valeur `Kim`, nous écrivons la condition logique représentant ce critère et la fournissons au premier argument de l'opérateur `[` à plusieurs dimensions.

```
data_ex[data_ex$lanceur == "Kim", ]
```

```
##   de1 de2 lanceur
## NA  NA  NA    <NA>
## 6   3   4     Kim
## 7   5  NA     Kim
## 8   6   2     Kim
## 9   5   5     Kim
## 10  4   3     Kim
```

Remarquons que l'observation pour laquelle la valeur de la variable `lanceur` est manquante a été conservée. Ensuite, la façon la plus simple de retirer les lignes comprenant au moins une donnée manquante est d'utiliser la fonction `na.omit` mentionnée précédemment.

```
na.omit(data_ex[data_ex$lanceur == "Kim", ])
```

```
##   de1 de2 lanceur
## 6   3   4     Kim
## 8   6   2     Kim
## 9   5   5     Kim
## 10  4   3     Kim
```

Finalement, la variable `lanceur` peut être retirée en fournissant les noms des autres variables, celles à conserver, au deuxième argument de l'opérateur `[`. Dans un cas de manipulation de data frame, l'identification des variables par leur nom plutôt que par leur position permet d'éviter des erreurs.

```
na.omit(data_ex[data_ex$lanceur == "Kim", c("de1", "de2")])
```

```
##    de1 de2
## 6    3  4
## 8    6  2
## 9    5  5
## 10   4  3
```

Effectuons maintenant le même travail avec la fonction `subset`, encore une fois étape par étape, pour bien comprendre les traitements réalisés.

```
subset(data_ex, subset = lanceur == "Kim")
```

```
##    de1 de2 lanceur
## 6    3  4      Kim
## 7    5 NA      Kim
## 8    6  2      Kim
## 9    5  5      Kim
## 10   4  3      Kim
```

Tout d'abord, remarquons que le comportement de la fonction `subset` diffère un peu de celui de l'opérateur `[`. L'observation pour laquelle la valeur de la variable `lanceur` est manquante n'a pas été conservée. Nous pourrions retirer les observations contenant des valeurs manquantes pour les autres variables en combinant des conditions logiques comme suit.

```
subset(data_ex, subset = !is.na(de1) & !is.na(de2) & lanceur == "Kim")
```

```
##    de1 de2 lanceur
## 6    3  4      Kim
## 8    6  2      Kim
## 9    5  5      Kim
## 10   4  3      Kim
```

Cependant, il est plus simple d'utiliser encore une fois la fonction `na.omit`. Pour retirer la variable `lanceur`, l'argument `select` de la fonction `subset` peut être spécifié comme dans la commande suivante, qui accomplit la tâche souhaitée.

```
na.omit(subset(data_ex, subset = lanceur == "Kim", select = -lanceur))
```

```
##    de1 de2
## 6    3  4
## 8    6  2
## 9    5  5
## 10   4  3
```

La possibilité de pouvoir retirer des variables en utilisant l'opérateur `-` devant leur nom permet de simplifier des commandes. Pour conserver ou retirer plus d'une variable, il faut fournir leurs noms dans un vecteur, comme nous l'avons fait précédemment dans la commande suivante (non exécutée ici).

```
head(subset(boxoffice, select = -c(Ranking, Gross)))
```

2.2.2 Fonctions `filter` et `select` du package `dplyr`

Le package `dplyr` attribue les tâches de sélection d'observations et de variables à deux fonctions distinctes :

- `filter` pour la sélection d'observations (lignes),
- `select` pour la sélection de variables (colonnes).

Par exemple, nous pouvons sélectionner les observations de `data_ex` pour lesquelles la valeur de la variable `lancer` est "Kim" avec la fonction `filter` comme suit.

```
filter(data_ex, lanceur == "Kim")
```

```
##   de1 de2 lanceur
## 1   3   4      Kim
## 2   5  NA      Kim
## 3   6   2      Kim
## 4   5   5      Kim
## 5   4   3      Kim
```

Tout comme le fait la fonction `subset`, les lignes pour lesquelles la condition vaut `NA` n'ont pas été conservées. Pour retirer les observations contenant des données manquantes, nous pouvons utiliser `na.omit`, ou encore `drop_na`, une fonction similaire contenue dans le package `tidyr`.

```
drop_na(filter(data_ex, lanceur == "Kim"))
```

```
##   de1 de2 lanceur
## 1   3   4      Kim
## 2   6   2      Kim
## 3   5   5      Kim
## 4   4   3      Kim
```

Notons que la fonction `drop_na` permet de limiter la recherche de valeurs manquantes à un sous-ensemble de variables, ce qui n'est pas possible avec `na.omit`.

Ensuite, la variable `lanceur` peut être retirée à l'aide de la fonction `select` comme suit.

```
select(drop_na(filter(data_ex, lanceur == "Kim")), -lanceur)
```

```
##   de1 de2
## 1   3   4
## 2   6   2
## 3   5   5
## 4   4   3
```

Cette commande est un peu difficile à lire en raison du grand nombre d'appels de fonctions imbriqués qu'elle contient. Elle serait peut-être plus claire en ajoutant des retours à la ligne et des indentations comme suit.

```
select(
  drop_na(
    filter(
      data_ex,
      lanceur == "Kim"
    )
  ),
  -lanceur
)
```

```
##   de1 de2
## 1   3   4
## 2   6   2
## 3   5   5
## 4   4   3
```

La pratique courante avec les packages du `tidyverse` est cependant d'utiliser l'opérateur « pipe » `%>%` du package `magrittr` pour rendre les commandes plus lisibles. En utilisant cet opérateur, la commande aurait l'allure suivante.

```
data_ex %>% filter(lanceur == "Kim") %>% drop_na() %>% select(-lanceur)
```

```
##   de1 de2
## 1   3   4
## 2   6   2
## 3   5   5
## 4   4   3
```

Je ne fais que mentionner cet opérateur ici. Il est recommandé, mais pas obligatoire, de l'employer lors de l'utilisation de packages du `tidyverse`. L'utilisation de cet opérateur est expliqué dans les [notes sur les bonnes pratiques de programmation en R](#).

2.2.3 Opérateur [et fonction subset du package data.table

Pour effectuer de la sélection de sous-ensembles de données avec la package `data.table`, nous pouvons exploiter son [opérateur \[et ses arguments i et j](#) ou encore sa [méthode pour la fonction subset](#) (qui est en fait une fonction générique) spécifique aux data tables. Illustrons ici l'utilisation de l'opérateur `[`. La fonction `subset` s'utilise à peu près de la même façon avec un data table qu'avec un data frame.

Après avoir transformé le data frame `data_ex` en data table,

```
data_ex_dt <- as.data.table(data_ex)
```

nous pouvons l'indiquer à l'aide de l'opérateur `[` de `data.table` en fournissant une condition logique à l'argument `i` pour sélectionner des lignes.

```
data_ex_dt[i = lanceur == "Kim"]
```

```
##   de1 de2 lanceur
## 1:   3   4     Kim
## 2:   5  NA     Kim
## 3:   6   2     Kim
## 4:   5   5     Kim
## 5:   4   3     Kim
```

Dans les commandes suivantes, je vais omettre l'assignation explicite à l'argument `i` puisque ce n'est vraiment pas une pratique courante d'inclure cette assignation dans un appel à `[`. Selon l'ordre des arguments de l'opérateur dans sa définition (voir la section Usage de sa [fiche d'aide](#)), la première valeur fournie après `[` est assignée à l'argument `i` et la deuxième à l'argument `j` si elles ne sont pas explicitement assignées à un nom d'argument (et si aucune valeur fournie en troisième position ou plus loin n'est assignée à `i` ou `j`, selon les [règles d'assignation implicite de valeurs aux arguments](#) présentée dans les notes sur les concepts de base en R).

Le package `data.table` offre une [méthode pour la fonction na.omit](#) spécifique aux data tables.

```
na.omit(data_ex_dt[lanceur == "Kim"])
```

```
##   de1 de2 lanceur
## 1:   3   4     Kim
## 2:   6   2     Kim
## 3:   5   5     Kim
## 4:   4   3     Kim
```

La sélection de variables dans un data table s'effectue en fournissant une valeur à l'argument `j`. Par exemple, nous pouvons sélectionner les variables `de1` et `de2` de `data_ex_dt` comme suit (commande non évaluée),

```
na.omit(data_ex_dt[lanceur == "Kim", .(de1, de2)])
```

ou encore omettre la variable `lanceur`, comme suit.

```
na.omit(data_ex_dt[lanceur == "Kim", !c("lanceur")])
```

```
##   de1 de2
```

```
## 1: 3 4
## 2: 6 2
## 3: 5 5
## 4: 4 3
```

Remarquons que l'omission de variables s'indique avec l'opérateur `!` avec `data.table`, plutôt qu'avec `-`.

2.1.2 Opérateurs de comparaison

Les **opérateurs de comparaison** permettent de comparer des valeurs. Ils retournent **TRUE** ou **FALSE**. Il s'agit des opérateurs suivants :

- **==** : égalité,
- **!=** : non-égalité,
- **>** : plus grand,
- **>=** : plus grand ou égal,
- **<** : plus petit,
- **<=** : plus petit ou égal.

Supposons **x** et **y** les deux vecteurs numériques suivants.

```
x <- c(2, 5, 7, 3)
y <- c(3, 5, 6, 4)
```

Comparons ces vecteurs à l'aide d'un opérateur de comparaison. Est-ce que les valeurs contenues dans `x` sont supérieures aux valeurs contenues dans `y` ?

```
x > y
```

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

L'opérateur fonctionne de façon vectorielle, donc une comparaison est effectuée pour toutes les paires d'éléments à la même position dans les vecteurs `x` et `y`. Les valeurs dans le résultat retourné sont de type logique.

Les valeurs dans un vecteur peuvent aussi être comparées à une seule valeur, auquel cas la règle de recyclage s'applique.

```
x != 5
```

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

2.1.2.1 Comparaison de valeurs non numériques

Les opérateurs de comparaison ne fonctionnent pas seulement avec des valeurs numériques. Ils peuvent aussi être utilisés pour comparer des valeurs logiques ou caractères. Dans ce cas, il faut savoir que R considère que `FALSE` est inférieure à `TRUE`.

```
FALSE < TRUE
```

```
## [1] TRUE
```

Quant aux caractères, les opérateurs de comparaison utilisent l'ordre de classement des caractères pour déterminer, entre deux valeurs, celle qui est inférieure. Cet ordre dépend des paramètres régionaux de la session R. D'une langue à l'autre, cet ordre peut varier.

Pour connaître l'ordre utilisé dans une session R, les instructions suivantes sont utiles :

```
caracteres_speciaux <-
  c("!", "\", "#", "$", "%", "&", "'", "(", ")", "*", "+", ",", "-", ".", "/", ":", ";",
    "<", "=", ">", "?", "@", "[", "\\", "]", "^", "_", "{", "|", "}", "~")
lettres_accentuees <- c("â", "ã", "ä", "å", "æ", "ç", "è", "é", "ê", "ë", "î", "ï", "ô", "ù", "ü", "û", "ÿ")
catacteres_ordonnes <- sort(c(caracteres_speciaux, 0:9, letters, LETTERS,
                              lettres_accentuees, toupper(lettres_accentuees)))
paste(catacteres_ordonnes, collapse = "")
```

J'ai obtenu le résultat suivant, qui sera peut-être différent sur votre ordinateur si vous n'avez pas les mêmes paramètres régionaux que moi.

```
"' - ! \"#$%&'()* , . / : ; ? @ [ \ ] ^ _ { | } ~ + < = > 0 1 2 3 4 5 6 7 8 9 a â ã ä å æ ç è é ê ë î ï ô ù ü û ÿ
n No O ô Ô p P q Q r R s S t Tu U ù Û ü Ü v V w W x X y Y z Z "
```

Ainsi, dans ma session R :

- les caractères spéciaux sont inférieurs aux chiffres et aux lettres,
- les chiffres sont inférieurs aux lettres,
- les lettres sont classées en ordre alphabétique et
 - les lettres minuscules sont inférieures aux lettres majuscules,
 - les lettres non accentuées sont inférieures aux lettres accentuées.

Pour des chaînes à plus d'un caractère, la comparaison s'effectue caractère par caractère (premiers caractères comparés entre eux, puis deuxièmes en cas d'égalité, puis troisièmes en cas d'égalité aux deux premières positions, etc.).

```
"arborescence" < "arbre"
```

```
## [1] TRUE
```

Aussi, l'absence de caractères vaut moins que la présence.

```
"a" < "aa"
```

```
## [1] TRUE
```

Remarque : Afin de correctement ordonner des nombres, il faut s'assurer de les stocker sous un format numérique. S'ils sont stockés sous forme de chaînes de caractères, les résultats obtenus ne seront pas toujours ceux attendus, comme dans cet exemple pour lequel 2 est dit non inférieur à 10 lorsque les nombres sont fournis à l'opérateur de comparaison sous forme de chaînes de caractères.

```
2 < 10
```

```
## [1] TRUE
```

```
"2" < "10"
```

```
## [1] FALSE
```

2.1.3 Opérateurs et fonction logiques vectoriels

Un [opérateur ou une fonction logique](#) vectoriel prend en entrée un ou deux vecteurs de logiques et retourne un autre vecteur de valeurs logiques. Le R de base comporte les opérateurs et la fonction logiques vectoriels suivants :

- `!` : négation,
- `&` : et,
- `|` : ou,
- `xor` : ou exclusif.

2.1.3.1 Opérateur de négation !

L'opérateur `!` n'a qu'un seul argument, alors que les autres opérateurs logiques en ont deux. Il effectue une négation, donc transforme les `TRUE` en `FALSE` et les `FALSE` en `TRUE`.

```
!c(TRUE, FALSE)
```

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

2.1.3.2 Opérateurs `&` et `|`, fonction `xor`

Les opérateurs `&` et `|`, ainsi que la fonction `xor`, appliquent de façon vectorielle les [tables de vérité](#) des fonctions mathématiques logiques « et », « ou » et « ou exclusif » respectivement.

Rappel : table de vérité de « et », « ou » et « ou exclusif »

```
p <- rep(c(FALSE, TRUE), each = 2)
q <- rep(c(FALSE, TRUE), times = 2)
cbind(p, q, "p et q" = p & q, "p ou q" = p | q, "p xor q" = xor(p, q))
```

```
##           p      q p et q p ou q p xor q
## [1,] FALSE FALSE FALSE FALSE FALSE
## [2,] FALSE TRUE  FALSE TRUE  TRUE
## [3,] TRUE  FALSE FALSE TRUE  TRUE
## [4,] TRUE  TRUE  TRUE  TRUE  FALSE
```

Ainsi,

- l'expression `p & q` retournera un vecteur contenant des `TRUE` aux positions pour lesquelles la valeur en `p` et la valeur en `q` sont toutes les deux `TRUE` et contenant des `FALSE` partout ailleurs ;
- l'expression `p | q` retournera un vecteur contenant des `FALSE` aux positions pour lesquelles la valeur en `p` et la valeur en `q` sont toutes les deux `FALSE` et contenant des `TRUE` partout ailleurs ;
- l'expression `xor(p, q)` retournera un vecteur contenant des `TRUE` aux positions pour lesquelles la valeur en `p` ou la valeur en `q` est `TRUE`, mais pas les deux, et contenant des `FALSE` partout ailleurs.

3.1.3 Fonctions de comparaison pour constantes spéciales

Notons que tester si un ou des éléments sont égaux à `NA`, `NaN`, `Inf` ou `-Inf`, ne se fait pas directement avec l'opérateur `==` comme suit.

```
c(1, 2, NA, 4, 5) == NA
```

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

Il faut plutôt utiliser la fonction `is.na`, `is.nan` ou `is.infinite`.

```
is.na(c(1, 2, NA, 4, 5))
```

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

La fonction `is.finite` retourne pour sa part `TRUE` pour les éléments non égaux à `NA`, `NaN`, `Inf` ou `-Inf`.

```
is.finite(c(NA, NaN, Inf, -Inf, 4.5, 3))
```

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

3.2.2 Fonctions `all` et `any`

Les fonctions `all` et `any` font partie des fonctions R retournant toujours une seule valeur logique. La fonction `all` indique si tous les éléments d'un vecteur logique sont `TRUE`. Par exemple, pour tester si toutes les valeurs dans le vecteur `de` sont entières au sens mathématique, nous pourrions utiliser la commande suivante.

```
all(de %% 1 == 0)
```

```
## [1] TRUE
```

La fonction `any` indique pour sa part si au moins un élément d'un vecteur logique est `TRUE`. Nous pourrions par exemple vérifier si le vecteur `de` comporte des valeurs négatives comme suit.

```
any(de < 0)
```

```
## [1] FALSE
```

1.1 Fonctions retournant une seule statistique

Certaines fonctions de calcul de statistiques descriptives retournent en sortie une seule valeur. C'est le cas des fonctions suivantes :

- mesures de position : `min`, `max` ;
- mesures de tendance centrale : `mean`, `median` ;
- mesure de dispersion : `sd` (écart-type), `IQR` (écart interquartile), `mad` (écart absolu médian).

Utilisons le jeu de données `cars` du package `datasets` pour présenter quelques exemples. Ce jeu de données contient 50 observations de 2 variables numériques.

```
str(cars)

## 'data.frame':   50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
# Moyenne des observations de la variable dist
mean(cars$dist)

## [1] 42.98
```

Si l'objet en entrée a plus d'une dimension, la sortie est tout de même de longueur 1. Donc tous les éléments contenus dans l'objet sont mis en commun pour faire le calcul.

```
max(cars)

## [1] 120
```

Fonctions `which.max` et `which.min`

Les fonctions `min` et `max` retournent respectivement la valeur la plus petite et la valeur la plus grande parmi les éléments d'un objet. Les fonctions `which.max` et `which.min` retournent pour leur part la position dans l'objet du **premier** maximum ou minimum.

```
which.min(cars$speed)

## [1] 1
```

Dans l'exemple précédent, il y a en fait deux observations qui prennent la valeur minimum de `min(cars$speed)`. La commande suivante permet de trouver la position de toutes les observations prenant la valeur minimale.

```
which(cars$speed == min(cars$speed))
```

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

1.1.1 Traitement des données manquantes et argument `na.rm`

Les fonctions `min`, `max`, `mean`, `median` et `sd`, ainsi que quelques autres fonctions vues dans ces notes, ont un argument en commun nommé `na.rm`. Cet argument sert à indiquer à la fonction comment agir en présence de données manquantes (NA). Par défaut, `na.rm` prend la valeur `FALSE` pour ces fonctions. Cette valeur signifie que les données manquantes ne doivent pas être retirées avant d'effectuer le calcul. Cependant, en présence de données manquantes, ces fonctions ne sont pas en mesure de calculer des statistiques. Par exemple, supposons que nous voulions calculer la médiane des données dans le vecteur suivant.

```
x <- c(3, 6, NA, 8, 11, 15, 23)
```

Si nous ne retirons pas la donnée manquante, nous obtenons le résultat suivant.

```
median(x)
```

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

Ce résultat s'explique par le fait que la valeur de la médiane dépend de toutes les observations, incluant l'observation manquante, qui est inconnue. La valeur de la médiane est donc elle aussi inconnue. Pour calculer plutôt la médiane des observations non manquantes, il faut donner la valeur `TRUE` à l'argument `na.rm` comme suit.

```
median(x, na.rm = TRUE)
```

```
## [1] 9.5
```

Notons que la fonction `na.omit` permet de retirer les observations manquantes d'un objet R. Si l'objet est un vecteur, les éléments contenant NA sont retirés.

```
na.omit(x)
```

```
## [1] 3 6 8 11 15 23
## attr(,"na.action")
## [1] 3
## attr(,"class")
## [1] "omit"
```

La fonction `na.omit` ajoute deux attributs à l'objet, dont un pour identifier les observations retirées.

Remarquons que les deux commandes suivantes retournent le même résultat.

```
median(x, na.rm = TRUE)
```

```
## [1] 9.5
```

```
median(na.omit(x))
```

```
## [1] 9.5
```

Si la fonction `na.omit` reçoit en entrée une matrice ou un data frame, elle retire toutes les lignes contenant au moins un NA, comme dans cet exemple :

```
ex_jeu <- data.frame(x, y = c(2, NA, 8, 9, 6, NA, 2));
ex_jeu
```

```
##      x  y
```

Résumés et valeurs manquantes

```
## 1  3  2
## 2  6 NA
## 3 NA  8
## 4  8  9
## 5 11  6
## 6 15 NA
## 7 23  2
```

```
na.omit(ex_jeu)
```

```
##      x y
## 1    3 2
## 4    8 9
## 5   11 6
## 7   23 2
```

1.3 Fonctions `tapply`, `by` et `aggregate`

Ces fonctions appliquent elles aussi la même fonction à plusieurs sous-objets. Ce qui les distingue des autres fonctions de la famille des `apply` est la formation des sous-objets, qui se réalise cette fois selon les niveaux de facteurs.

Nous allons reprendre le jeu de données `Puromycin` pour illustrer l'utilisation de ces fonctions.

```
str(Puromycin)

## 'data.frame': 23 obs. of 3 variables:
## $ conc : num 0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 ...
## $ rate : num 76 47 97 107 123 139 159 152 191 201 ...
## $ state: Factor w/ 2 levels "treated","untreated": 1 1 1 1 1 1 1 1 1 1 ...
## - attr(*, "reference")= chr "A1.3, p. 269"
```

Fonction `tapply`

Nous pourrions par exemple calculer la moyenne de la variable `rate` selon les niveaux du facteur `state` comme suit.

```
tapply(Puromycin$rate, INDEX = Puromycin$state, FUN = mean)

## treated untreated
## 141.5833 110.7273
```

L'argument `INDEX` pourrait être une liste de plusieurs facteurs.

```
tapply(Puromycin$rate, INDEX = Puromycin[, c("conc", "state")], FUN = mean)

## state
## conc treated untreated
## 0.02 61.5 59.0
## 0.06 102.0 85.0
## 0.11 131.0 106.5
## 0.22 155.5 127.5
## 0.56 196.0 151.0
## 1.1 203.5 160.0
```

Dans l'exemple ci-dessous, nous avons fourni à `INDEX` un data frame, mais rappelons-nous que les data frames sont des cas particuliers de listes. De plus, un élément de ce data frame n'est pas un facteur. Il s'agit de la variable `conc`. Cela n'a pas posé problème parce que `tapply` est arrivé à transformer l'élément en facteur.

Fonction `by`

La fonction `by` prend comme objet en entrée un data frame et permet d'effectuer un calcul sur des sous-objets qui sont aussi des data frames. Par exemple, nous pourrions calculer la matrice de corrélations entre les observations des variables `conc` et `rate` selon les niveaux du facteur `state` comme suit.

```
by(Puromycin[, c("conc", "rate")], INDICES = Puromycin$state, FUN = cor)

## Puromycin$state: treated
## conc rate
## conc 1.0000000 0.8310362
## rate 0.8310362 1.0000000
## -----
## Puromycin$state: untreated
```

```
##           conc           rate
## conc 1.0000000 0.8207311
## rate 0.8207311 1.0000000
```

Fonction aggregate

Finalement, la fonction `aggregate` prend aussi en entrée un data frame, mais elle applique la fonction séparément pour chaque colonne du data frame.

```
aggregate(Puromycin[, c("conc", "rate")], by = list(state = Puromycin$state), FUN = mean)
```

```
##           state           conc           rate
## 1   treated 0.3450000 141.5833
## 2 untreated 0.2763636 110.7273
```

L'argument `by` doit obligatoirement être une liste. Nommer les éléments de la liste aide à clarifier la sortie.

La fonction `aggregate` accepte aussi des formules en entrée, comme dans les exemples ci-dessous.

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
aggregate(cbind(conc, rate) ~ state, data = Puromycin, FUN = mean)
```

```
##           state           conc           rate
## 1   treated 0.3450000 141.5833
## 2 untreated 0.2763636 110.7273
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
aggregate(rate ~ conc + state, data = Puromycin, FUN = median)
```

```
##    conc    state  rate
## 1  0.02  treated  61.5
## 2  0.06  treated 102.0
## 3  0.11  treated 131.0
## 4  0.22  treated 155.5
## 5  0.56  treated 196.0
## 6  1.10  treated 203.5
## 7  0.02 untreated  59.0
## 8  0.06 untreated  85.0
## 9  0.11 untreated 106.5
## 10 0.22 untreated 127.5
## 11 0.56 untreated 151.0
## 12 1.10 untreated 160.0
```

2 Solutions de rechange pour calculs par niveaux de facteurs

Quelques packages R offrent d'autres fonctions permettant de réaliser des calculs par niveaux de facteurs. L'utilisation de deux de ces packages, souvent mentionnés par la communauté R, est illustrée brièvement ici en reproduisant les deux exemples précédents.

2.1 Package dplyr

L'utilisation conjointe des fonctions `group_by` et `summarize` du package `dplyr` du `tidyverse` permet d'agréger, en utilisant des statistiques de notre choix, les observations de variables selon les niveaux de facteurs.

Reproduisons avec `dplyr` les deux derniers exemples présentés pour la fonction `aggregate`.

```
library(dplyr)
```

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
summarize(group_by(Puromycin, state), conc = mean(conc), rate = mean(rate))
```

```
## # A tibble: 2 x 3
##   state      conc rate
## * <fct>    <dbl> <dbl>
## 1 treated    0.345  142.
## 2 untreated  0.276  111.
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
summarize(group_by(Puromycin, conc, state), rate = median(rate))
```

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

```
## # A tibble: 12 x 3
## # Groups:   conc [6]
##   conc state      rate
##   <dbl> <fct>    <dbl>
## 1  0.02 treated    61.5
## 2  0.02 untreated   59
## 3  0.06 treated   102
## 4  0.06 untreated   85
## 5  0.11 treated   131
```

```
## 6 0.11 untreated 106.
## 7 0.22 treated 156.
## 8 0.22 untreated 128.
## 9 0.56 treated 196
## 10 0.56 untreated 151
## 11 1.1 treated 204.
## 12 1.1 untreated 160
```

Les statistiques obtenues n'ont pas changé, mais elles sont retournées dans un [tibble](#) au lieu d'un data frame.

Il faut fournir comme premier argument à la fonction `summarize` le jeu de données sur lequel effectuer les calculs. De plus, pour réaliser des calculs selon les niveaux d'un facteur, le jeu de données doit être préalablement brisé en sous-groupes avec la fonction `group_by`. Par exemple, l'expression `group_by(Puromycin, conc, state)` permet d'indiquer que les observations du jeu de données `Puromycin` (argument 1 de `group_by` = jeu de données) doivent être groupées selon les niveaux des variables `conc` et `state` (arguments suivants de `group_by` = noms des colonnes du jeu de données contenant les facteurs selon lesquels créer les groupes). L'appel à la fonction `group_by` est donc assigné au premier argument de `summarize`. Les arguments suivants de `summarize` servent à définir les statistiques à calculer. Autant de statistiques que désiré peuvent être demandées. Les statistiques sont définies par des appels à des fonctions R de calcul statistique, en fournissant comme premier argument à la fonction le nom d'une variable dans le jeu de données (p. ex. `mean(conc)`). Si cette expression est précédée d'un opérateur `=` et d'un nom, celui-ci apparaîtra dans la sortie produite.

Il aurait été possible de passer d'autres arguments dans les appels aux fonctions de calcul statistique. Par exemple, reprenons le premier exemple ci-dessus et demandons le calcul de moyennes tronquées.

```
summarize(
  group_by(Puromycin, state),
  conc = mean(conc, trim = 0.1),
  rate = mean(rate, trim = 0.1),
  n_obs = n()
)
```

```
## # A tibble: 2 x 4
##   state      conc rate n_obs
## * <fct>    <dbl> <dbl> <int>
## 1 treated  0.302  144.    12
## 2 untreated 0.213  112.    11
```

Ici, le calcul du nombre d'observations par sous-groupe a aussi été demandé avec la fonction `n` du package `dplyr`. Cette fonction n'a besoin d'aucun argument.

Notons que les fonctions de `dplyr` sont souvent utilisées avec l'opérateur `%>%`, nommé « *forward-pipe operator* ». En utilisant cet opérateur, la dernière instruction aurait l'allure suivante.

```
Puromycin %>%
  group_by(state) %>%
  summarize(
    conc = mean(conc, trim = 0.1),
    rate = mean(rate, trim = 0.1),
    n_obs = n()
  )
```

```
## # A tibble: 2 x 4
##   state      conc rate n_obs
## * <fct>    <dbl> <dbl> <int>
## 1 treated  0.302  144.    12
## 2 untreated 0.213  112.    11
```

L'utilisation de l'opérateur %>% sera expliquée plus tard, dans les [notes sur les bonnes pratiques](#). Bien que son utilisation avec les fonctions du **tidyverse** soit très populaire, elle n'est pas obligatoire.

Pour obtenir plus d'informations sur les calculs par niveaux de facteurs avec le package **dplyr** :

- <https://dplyr.tidyverse.org/articles/grouping.html#summarise->
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2016/agreger_donnees_dplyr/

2.2 Package **data.table**

Il est aussi possible de réaliser ces agrégations grâce à l'argument **by** de l'opérateur **[** du package **data.table**. Ce package offre l'avantage de pouvoir effectuer ces opérations rapidement sur de grands jeux de données.

Reproduisons encore une fois les deux derniers exemples présentés pour la fonction **aggregate**.

```
library(data.table)
Puromycin_dt <- data.table(Puromycin)
```

Tout d'abord, le jeu de données doit être transformé en data table.

```
# Exemple avec deux variables réponses et une variable explicative (de groupement)
Puromycin_dt[, j = .(conc = mean(conc), rate = mean(rate)), by = state]
```

```
##      state      conc      rate
## 1:  treated 0.3450000 141.5833
## 2: untreated 0.2763636 110.7273
```

```
# Exemple avec une variable réponse et deux variables explicatives (de groupement)
Puromycin_dt[, j = .(rate = median(rate)), by = .(conc, state)]
```

```
##      conc      state      rate
## 1: 0.02   treated    61.5
## 2: 0.06   treated   102.0
## 3: 0.11   treated   131.0
## 4: 0.22   treated   155.5
## 5: 0.56   treated   196.0
## 6: 1.10   treated   203.5
## 7: 0.02   untreated    59.0
## 8: 0.06   untreated    85.0
## 9: 0.11   untreated   106.5
## 10: 0.22   untreated   127.5
## 11: 0.56   untreated   151.0
## 12: 1.10   untreated   160.0
```

Ensuite, la clé est de spécifier adéquatement les arguments **j** et **by** de l'opérateur **[** pour le data table. Cet opérateur avait été présenté dans les [notes sur les structures de données en R](#).

Pour obtenir plus d'informations sur les calculs par niveaux de facteurs avec le package **data.table** :

- <https://rdatatable.gitlab.io/data.table/articles/datatable-intro.html#aggregations>
- https://stt4230.rbind.io/tutoriels_etudiants/hiver_2017/data.table/