Manipulation de données avec **R**

Sébastien Déjean et Thibault Laurent

06 octobre 2021

Contents

Ce document a été généré directement depuis **RStudio** en utilisant l'outil Markdown. La version .pdf se trouve ici.

1 Introduction

1.1 Principe du document

Ce document fait suite aux documents suivants :

- *Pour se donner un peu d'**R*** disponible ici
- *Introduction à **R*** disponible ici

Il contient des commandes à saisir, des commentaires pour attirer l'attention sur des points particuliers et quelques questions/réponses pour tester la compréhension des notions présentées. Pour certains points particuliers, nécessitant par exemple un environnement logiciel particulier, les faits ne sont que mentionnés et il n'y a pas toujours de mise en oeuvre pratique.

1.2 Pré-requis

Même si la plupart des points abordés dans ce document ne sont pas très compliqués, ils relèvent d'une utilisation avancée de **R** et ne s'adressent donc pas au débutant en **R**. Avant d'utiliser ce document, le lecteur doit notamment savoir :

- se servir de l'aide en ligne de **R**,
- manipuler les objets de base de **R** : vecteur, matrice, liste, **data.frame**,
- programmer une fonction élémentaire.

1.3 Packages à installer

```
install.packages(c(
  # import data:
  "foreign", "jsonlite", "readr", "readxl", "sas7bdat", "XML",
  # big data analysis
  "data.table", "ff", "ffbase",
  # matrices creuses
  "Marix",
  # character treatment
  "classInt", "glue", "stringr", "wordcloud",
  "gplots", # plotting data with ggplot2 style
  "tidyverse", "DSR", # Data Scientists toolkits
  "Amelia", "DMwR", "missForest", "naniar",  # missing values treatement
  "sp", "sf", # spatial data object
  "zoo")  # Time series analysis
)
```

2 Les chaînes de caractères

2.1 Fonctions de base

On considère la chaîne de caractère suivante :

```
phrase <- "Notes obtenues\nanglais: 16, Stat:14, Eco=18"
class(phrase)
```

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

Parmi les fonctions qui permettent de manipuler les chaînes de caractères, voici celles qui nous semblent importantes de connaître :

2.1.1 *nchar()*

Permet de compter le nombre de caractères de chaque élément d'un vecteur :

```
nchar(phrase)
```

```
## [1] 43
```

2.1.2 *substr()*

Permet d'extraire un sous-ensemble de caractères :

```
substr(phrase, start = 1, stop = 5)
```

```
## [1] "Notes"
```

2.1.3 *strsplit()*

Permet d'éclater une chaîne de caractères dès qu'on trouve une sous-chaîne particulière :

```
strsplit(phrase, split = " ")
```

```
## [[1]]
## [1] "Notes"              "obtenues\nanglais:" "16,"
## [4] "Stat:14,"           "Eco=18"
```

```
strsplit(phrase, split = "\n")
```

```
## [[1]]
## [1] "Notes obtenues"              "anglais: 16, Stat:14, Eco=18"
```

Remarque 1 : un espace est considéré comme une chaîne de caractère.

Remarque 2 : "\n" est un caractère spécial qui correspond à un saut de ligne (pour consulter la liste des caractères spéciaux, voir cette page web). Pour évaluer un caractère spécial dans une chaîne de caractère, on peut utiliser la fonction *cat()*:

```
cat(phrase)
```

```
## Notes obtenues
## anglais: 16, Stat:14, Eco=18
```

Remarque 3 : l'objet retourné est de type **list**. Pour convertir une **list** en un vecteur, sur lequel il est plus facile de faire de la manipulation, on utilise la fonction *unlist()*. Enfin, si on veut éclater une chaîne de caractère en fonction de plusieurs caractères, on utilise le symbole |. Dans l'exemple suivant, l'idée est de séparer tous les mots en présence d'un des caractères spéciaux :

```
(mots <- strsplit(phrase, split = ",|\n| |:|="))
```

```
## [[1]]
##  [1] "Notes"    "obtenues" "anglais"  ""         "16"       ""
##  [7] "Stat"     "14"       ""         "Eco"      "18"
```

```
(mots <- unlist(mots))
```

```
##  [1] "Notes"    "obtenues" "anglais"  ""         "16"       ""
##  [7] "Stat"     "14"       ""         "Eco"      "18"
```

Si on utilise le type **NULL** comme critère de recherche, cela a pour effet d'éclater tous les éléments de la chaîne de caractères en des caractères uniques :

```
(lettres <- strsplit(phrase, split = NULL))
```

```
## [[1]]
##  [1] "N"  "o"  "t"  "e"  "s"  " "  "o"  "b"  "t"  "e"  "n"  "u"  "e"  "s"  "\n"
## [16] "a"  "n"  "g"  "l"  "a"  "i"  "s"  ":"  " "  "1"  "6"  ","  " "  "S"  "t"
## [31] "a"  "t"  ":"  "1"  "4"  ","  " "  "E"  "c"  "o"  "="  "1"  "8"
```

```
length(lettres[[1]])
```

```
## [1] 43
```

Remarque : dans le contexte d'une étude statistique, on appliquera cette fonction à des vecteurs de caractère. Par exemple :

```
strsplit(mots, split = ":")
```

```
## [[1]]
## [1] "Notes"
##
## [[2]]
## [1] "obtenues"
##
## [[3]]
## [1] "anglais"
##
## [[4]]
## character(0)
##
## [[5]]
## [1] "16"
##
## [[6]]
## character(0)
##
## [[7]]
## [1] "Stat"
##
## [[8]]
## [1] "14"
##
## [[9]]
## character(0)
##
## [[10]]
```

```
## [1] "Eco"
##
## [[11]]
## [1] "18"
```

2.1.4 *toupper()*, *tolower()*

Pemettent de convertir toutes les lettres en majuscules et minuscules :

```
toupper(phrase)
```

```
## [1] "NOTES OBTENUES\nANGLAIS: 16, STAT:14, ECO=18"
```

```
tolower("AAA")
```

```
## [1] "aaa"
```

2.1.5 *grep()*

Permet de trouver dans un vecteur de caractères quels sont les indices des composantes du vecteur qui contiennent un ensemble de caractères. Par exemple quels sont les mots qui contiennent la lettre "e" :

```
(res <- grep(pattern = "e", x = mots))
```

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

```
mots[res]
```

```
## [1] "Notes"    "obtenues"
```

On peut chercher plusieurs lettres à la fois. Ici, on cherche les mots qui contiennent une des lettres "j", "J" et "t". Pour cela, on utilise une expression régulière grâce aux crochets (nous verrons dans la section suivante plus en détail le fonctionnement des expressions régulières) :

```
(res <- grep(pattern = "[jSE]", x = mots))
```

```
## [1]  7 10
```

```
mots[res]
```

```
## [1] "Stat" "Eco"
```

Remarque : un vecteur de taille nulle est retourné si le critère n'est pas satisfait.

2.1.6 *agrep()*

Pemet de trouver dans un vecteur de caractères quels sont les indices des composantes du vecteur qui contiennent "approximativement" une sous-chaîne, l'approximation pouvant être réglée avec les options de la fonction. Dans l'exemple suivant, la lettre s minuscule est différente de S majuscule, mais cette différence d'1 caractère sera tolérée.

```
agrep("stat", mots)
```

```
## [1] 7
```

Remarque : lorsqu'on traite des fichiers de données brutes, ce fichier peut contenir des erreurs de saisie, par exemple "Toulouze" au lieu de "Toulouse" et c'est pourquoi le fait de tolérer un nombre de différences peut s'avérer intéressant.

- *sub()* pour changer une sous-chaîne de caractères par une autre :

```
(mots <- sub(pattern = "=", replacement = ":", x = phrase))
```

```
## [1] "Notes obtenues\nanglais: 16, Stat:14, Eco:18"
```

2.1.7 *regexpr()*

Permet de dire à quelle position dans le mot se trouve une sous-chaîne de caractères. Dans l'exemple suivant, on cherche à savoir où se trouve le caractère ":" dans les mots. Si un caractère est présent, alors la fonction retourne les positions de la lettre ":" et si elle n'apparaît pas, la valeur -1 est retournée. Le résultat est encore donné sous forme de **list** car cela permet de traiter chaque mot du vecteur.

```
gregexpr(pattern = ":", text = mots, ignore.case = TRUE)
```

Compléments : en pratique, on peut vouloir faire des recherches plus complexes (plusieurs caractères, des caractères spéciaux, etc), ce qui nécessite une adaptation dans les critères de recherche. La gestion des chaînes de caractères est synthétisée dans l'aide suivante :

```
help(regexp)
```

2.1.8 *abbreviate()*

Pemet de faire des abbréviations de chaînes de caratères trop longue, tout en respectant l'unicité de chaque mot (autement dit, une même abbréviation ne peut pas être donnée à deux mots différents). Par exemple :

```
pays <- c("Bosnie-Herzégovine", "Burkina Faso",  "Côte d'Ivoire")
abbreviate(pays)
```

```
## Warning in abbreviate(pays): abbreviate utilisé avec des caractères non ASCII
```

```
## Bosnie-Herzégovine       Burkina Faso      Côte d'Ivoire
##             "Bs-H"             "BrkF"             "Cd'I"
```

Exercice 1.1

Q1 Compter le nombre de caractères de chaque élément du vecteur suivant. Que constatez-vous ?

```
x_with_missing <- c("oui", "peut-être", NA, "non", NA, "si")
```

Q2 Dans le vecteur suivant, faire l'extraction des deux entiers qui se trouvent entre le symbole _ et présenter le résultat sous forme de vecteur :

```
code_INSEE <- c("toulouse_31_HG", "lyon_69_Rhone",
                "marsei_13_PACA")
```

2.2 Les expressions régulières

Ce paragraphe s'inspire de cette note de cours écrite par Ricco Rakotomalala.

On va s'attarder sur l'utilisation d'expressions régulières qui est très populaire dans certaines disciplines et notamment le text mining qui englobe l'analyse de tweets.

2.2.1 Définition

Une expression régulière est une séquence de caractères qui définit un motif d'intérêt. Par exemple, on considère le motif d'intérêt "b.b.", une séquence de 4 caractères où le 1er et 3ème caractères sont le caractère "b" et le 2ème et 4ème peuvent être n'importe quel autre caractère tel que "bobi", "buba", "bib1", "bpbe", "byb=", etc.

On pourrait considérer un second motif d'intérêt "b.b.", une séquence de 4 caractères où le 1er et 3ème caractère sont le caractère "b" et le 2ème et 4ème peuvent être une voyelle uniquement. Dans ce cas, "bybe" serait un candidat, mais pas "bib1".

Une expression régulière correspond donc à la syntaxe informatique qui sera utilisée pour détecter un motif d'intérêt.

2.2.1.1 Exemples Par défaut, les fonctions *strsplit()*, *grep()*, *sub()* ou *regexpr()* permettent d'utiliser une expression régulière comme critère de recherche. Par exemple, pour identifier le 1er motif d'intérêt, l'expression régulière est la suivante "b.b." où le "." indique donc que n'importe quel caractère est accepté

```
textes <- c("bobi", "bibé", "tatane", "bAbA", "tbtc",
            "tut", "byb=", "baba", "bub1", "t5t3")
print(grep("b.b.", textes))
```

```
## [1] 1 2 4 7 8 9
```

Pour le second motif d'intérêt, on va utiliser l'expression régulière suivante :

```
print(grep("b[aeiouy]b[aeiouy]", textes))
```

```
## [1] 1 8
```

On met entre crochets les caractères qui sont autorisés.

La négation des caractères autorisés est obtenue avec le symbole "^". Par exemple, dans l'exemple suivant, on autorise tous les caractères sauf les voyelles :

```
print(grep("b[^aeiouy]b[^aeiouy]", textes))
```

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

Pour autoriser une suite de caractères, on utilise le symbole "-". Par exemple, si on autorise uniquement les lettres minuscules de l'alphabet, on fait:

```
print(grep("b[a-z]b[a-z]", textes))
```

```
## [1] 1 8
```

Si on autorise toutes les lettres de l'alphabet (minuscules et majuscules ainsi que les caractères spéciaux comme les accents é, à, è, ç, etc.), on utilise la syntaxe "[:alpha:]" entre crochets. Par exemple :

```
print(grep("b[[:alpha:]]b[[:alpha:]]", textes))
```

```
## [1] 1 2 4 8
```

Si on utilise toutes les lettres de l'alphabet ainsi que les chiffre numériques, on utilise la syntaxe "[:alnum:]" entre crochets. Par exemple :

```
print(grep("b[[:alnum:]]b[[:alnum:]]", textes))
```

```
## [1] 1 2 4 8 9
```

2.2.2 Les autres expressions régulières

Nous avons résumé différentes expressions régulières pouvant être utilisées :

- [...] : un des caractères indiqués entre les crochets. Par exemple:

```
print(grep("t[aeiouy]t[aeiouy]", textes))
```

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

- [^...] : tous les caractères sauf ceux indiqués après le ^. Par exemple:

```
print(grep("t[^aeiouy]t[^aeiouy]", textes))
```

```
## [1]  5 10
```

- [x-y] : les caractères compris entre x à y inclus. Par exemple :

```
print(grep("t[a-z]t[a-z]", textes))
```

```
## [1] 3 5
```

- [:alnum:] équivalent à a-zA-Z0-9 avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées comme les é, è, ù, ç, à.
- [:alpha:] équivalent à a-zA-Z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées
- [:digit:] équivalent à 0-9. Par exemple :

```
print(grep("t[[:digit:]]t[[:digit:]]", textes))
```

```
## [1] 10
```

- [:lower:] équivalent à a-z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées
- [:upper:] équivalent à A-Z avec en plus les caractères spéciaux que l'on retrouve suivant les langues utilisées comme les Â, Û, Ô, etc.
- [:xdigit:] équivalent à 0-9a-fA-F
- [:graph:] tout caractère graphique
- [:print:] tout caractère affichable
- [:punct:] tout caractère de ponctuation
- [:blank:] espace, tabulation
- [:space:] espace, tabulation, nouvelle ligne, retour chariot
- [:cntrl:] tout caractère de contrôle

Exercice 1.2:

On considère la chaîne de caractère suivante :

```
ww <- "we went to warwick 5 times"
```

Avec la fonction *gregexpr()*, trouver les expressions régulières qui permettent de donner l'emplacement de :

- la chaîne de caractères "i"
- un chiffre numérique
- la chaîne de caractères "we"
- une des deux chaînes de caractères "w" ou "e"
- un pattern commençant par un espace suivi de la chaîne "w".

2.3 Application aux tweets

On considère le vecteur suivant dont les éléments correspondent à des extraits de tweets

```
tweet <- c("TopStartupsUSA: RT @FernandoX: 7 C's of Marketing in the Digital Era.\n",
  "#Analytics #MachineLearning #DataScience #MalWare #IIoT",
  "YvesMulkers: RT @wil_bielert: RT @neptanum: Standard Model Physics from an Algebra?",
  "#BigData #Analytics #DataScience #AI #MachineLearning #IoT #IIoT #Python")
```

Exercice 1.3.

Expliquer la fonction de chacune des expressions régulières suivantes

```
correct <- gsub("(RT|via)((?:\\b\\W*@\\w+)+)", "", tweet)
correct <- gsub("@\\w+", "", correct)
correct <- gsub("[[:punct:]]", "", correct)
correct <- gsub("[[:digit:]]", "", correct)
correct <- gsub("http\\w+", "", correct)
correct <- gsub("[\t ]{2,}", " ", correct)
correct <- gsub("^\\s+|\\s+$", "", correct)
correct <- iconv(correct, "UTF-8", "ASCII", sub = "")
```

2.4 Nuage de mots

On présente ici la fonction *wordcloud()* du package **wordcloud** qui permet de représenter un nuage de mots en fonctions du nombre d'occurences des mots trouvés dans un corps de texte.

```
word <- unlist(strsplit(correct, " "))
tab_word <- table(word)
require("wordcloud")
```

```
## Le chargement a nécessité le package : wordcloud
```

```
## Le chargement a nécessité le package : RColorBrewer
```

```
wordcloud(names(tab_word), tab_word)
```

2.5 Ordonnancement

Les caractères peuvent s'ordonner comme on le fait avec des chiffres. Par exemple, le caratère "a" est plus petit que "b" qui n'est pas plus grand que "c". Pour le vérifier :

```
"a" < "b"
```

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

```
"b" > "c"
```

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

On s'intéresse ici plus particulièrement aux règles d'ordonnancement utilisées pour la langue française. Selon la langue utilisée par la machine, il existe donc des règles particulières pour ordonnancer les chaînes de caractères, qui diffèrent d'une langue à l'autre. Pour vérifier la langue utilisée par la machine, on peut utiliser la commande

```
Sys.setlocale(category = "LC_CTYPE", locale = "")
```

```
## [1] "fr_FR.UTF-8"
```

Considérons la citation suivante stockée dans l'objet **citation** :

```
citation <- "Il est important que les étudiants portent un regard neuf
et irrévérencieux  sur leurs études ; il ne doivent pas vénérer le savoir
mais le remettre en question (chapitre : 1 - paragraphe : 2 - ligne : 10
- page : 185. Jacob Chanowski)."
```

On peut récupérer chaque "mot" (entités séparées par des espaces") par la commande :

```
(mots <- unlist(strsplit(citation, split = " ")))
```

```
##  [1] "Il"            "est"           "important"     "que"
##  [5] "les"           "étudiants"     "portent"       "un"
##  [9] "regard"        "neuf"          "\net"          "irrévérencieux"
## [13] ""              "sur"           "leurs"         "études"
## [17] ";"             "il"            "ne"            "doivent"
## [21] "pas"           "vénérer"       "le"            "savoir"
## [25] "\nmais"        "le"            "remettre"      "en"
## [29] "question"      "(chapitre"     ":"             "1"
## [33] "-"             "paragraphe"    ":"             "2"
## [37] "-"             "ligne"         ":"             "10"
## [41] "\n-"           "page"          ":"             "185."
## [45] "Jacob"         "Chanowski)."
```

Le tri des éléments (uniques) du vecteur **mots** nous montre les règles d'ordonnancement appliquées par **R**.

```
sort(unique(mots))
```

```
##  [1] ""              "\n-"           "\net"          "\nmais"
##  [5] "-"             ";"             ":"             "(chapitre"
##  [9] "1"             "10"            "185."          "2"
## [13] "Chanowski)."   "doivent"       "en"            "est"
## [17] "études"        "étudiants"     "il"            "Il"
## [21] "important"     "irrévérencieux" "Jacob"        "le"
## [25] "les"           "leurs"         "ligne"         "ne"
## [29] "neuf"          "page"          "paragraphe"    "pas"
## [33] "portent"       "que"           "question"      "regard"
## [37] "remettre"      "savoir"        "sur"           "un"
## [41] "vénérer"
```

- les caractères spéciaux **-**, **:**, **;**, **(**, etc. sont prioritaires sur les chiffres et les lettres.
- les chiffres sont prioritaires sur les lettres.
- les mots sont ordonnancés comme dans un dictionnaire français.
- quand il y a des lettres avec des accents, on ordonnance comme s'il n'y avait pas d'accents.
- les lettres majuscules sont insérées dans l'ordre alphabétique et ne sont pas prioritaires par rapport aux lettres minuscules.
- la syntaxe "$\backslash n$" n'est pas comptabilisée.
- les chiffres ne sont pas regardés comme des chiffres mais comme une chaîne de caractères. Autrement dit "10" doit être vu comme un mot avec 2 caractères consécutifs : "1", puis "0". "2" doit être vu comme 1 mot avec un seul caractère. Pour comparer ces 2 mots, on compare les caractères entre eux les uns après les autres. Dans un premier temps, on regarde le 1er caractère de chaque mot : "1" est plus petit que "2". Aussi, quelque soit le nombre de caractère qu'on va ajouter après "1" ce mot sera plus petit que "2". Par exemple "15552525" sera plus petit que "2".

Exercice 1.4.

A partir du jeu de données **USArrests**, extraire les lignes dont le nom contient la chaîne de caractères "New". Vous pouvez vous inspirer des instructions suivantes. Dans le premier cas, tous les noms de lignes contenant la lettre **C** sont renvoyés ; dans le second cas, seuls ceux commencant par **C** sont renvoyés. Consulter la fiche d'aide sur les expressions régulières pour en savoir (beaucoup !) plus (**help(regexp)**).

```
USArrests[grep("C", rownames(USArrests)),]
```

```
##                Murder Assault UrbanPop Rape
## California        9.0     276       91 40.6
## Colorado          7.9     204       78 38.7
## Connecticut       3.3     110       77 11.1
## North Carolina   13.0     337       45 16.1
## South Carolina   14.4     279       48 22.5
```

```
USArrests[grep("^C", rownames(USArrests)),]
```

```
##             Murder Assault UrbanPop Rape
## California     9.0     276       91 40.6
## Colorado       7.9     204       78 38.7
## Connecticut    3.3     110       77 11.1
```

2.6 Package stringr

On a vu ci-dessus les fonctions de base pour manipuler des chaînes de caractères. Toutefois, si on cherche à faire des statistiques un plus poussées sur les chaînes de caractères, on va devoir avoir recours aux fonctions *sapply()* ou *lapply()* qui permettent de faire des opérations sur les listes. Par exemple, on cherche à calculer le nombre de fois qu'apparaît la lettre "a" dans le vecteur **mots** précédent. Pour cela, on va appliquer la fonction *sapply()* (dont nous reparlerons plus tard) sur le résultat donné par la fonction *gregexpr()*. On rappelle que le résultat de cette fonction est -1 si le caractère n'a pas été trouvé et sinon, il retourne le vecteur des positions. Pour répondre à notre problème, on exécute donc le code suivant :

```
res1 <- gregexpr(pattern = "a", text = mots, ignore.case = T)
sapply(res1, function(x) ifelse(x[1] > 0, length(x), 0))
```

```
##  [1] 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 3 0 0 0 0
## [39] 0 0 0 1 0 0 1 1
```

Le package **stringr** a vu le jour dans le but d'effacer certaines lacunes des fonctions de base et également simplifier la syntaxe des fonctions de base. Adopter les fonctions de ces packages revient un peu à oublier la

syntaxe des fonctions que nous avons vues. Par exemple, la fonction *str_c()* est plus ou moins équivalente à la fonction *paste()*, la fonction *str_length()* est équivalent à la fonction *nchar()*. La fonction *str_count()* retourne le même résultat précédent en 1 seul ligne de commande :

```
library("stringr")
str_count(mots, "a")
```

```
##  [1] 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 3 0 0 0 0
## [39] 0 0 0 1 0 0 1 1
```

Parmi les autres fonctions intéressantes de ce package, on notera la fonction *str_pad()* qui permet de faire en sorte qu'une chaîne de caractère (1er argument de la fonction) possède au minimum un nombre de caractère (2ème argument) et la complète si nécessaire avec un caractère (3ème argument). Par exemple, si on souhaite que chaque élément du vecteur suivant possède au moins 3 caractères et qu'on souhaite compléter les chaînes manquantes par le caractère "-" en début de chaîne, on procède ainsi :

```
vec_to_change <- c("1", "10", "105", "9999", "0008")
str_pad(vec_to_change, 4, pad = "0")
```

```
## [1] "0001" "0010" "0105" "9999" "0008"
```

On notera que le package **stringr** a été développé par Hadley Wickam (**RStudio**) qui est l'auteur d'un grand nombre d'autres packages avec cet esprit de rendre les choses plus simples pour l'utilisateur. Nous vous présenterons ces outils au fur et à mesure, mais à notre sens, il est important d'avoir conscience que derrière ces fonctions, se cachent des programmes qui utilisent les fonctions de base de **R**.

On citera également le package **stringi** qui propose un grand nombre de fonctions plus orientées vers l'analyse statistique de chaînes de caractères.

Bibliographie On pourra consulter ce document très intéressant et complet sur la gestion des chaînes de caractères avec **R** (la version gratuite est très bien) : https://leanpub.com/r4strings

2.7 Package glue

Ce package contient la fonction *glue()* qui permet d'insérer dans du texte des objets qui ont été créés dans l'environnement courant. En reprenant l'exemple de l'auteur du package (https://cran.r-project.org/web/packages/glue/readme/README.html) :

```
require("glue")
```

```
## Le chargement a nécessité le package : glue
```

```
name <- "Fred"
anniversary <- as.Date("1991-10-12")
age <- as.numeric(floor((Sys.Date() - anniversary)/365))
new_object <- glue('My name is {name},',
  ' my age next year is {age + 1},',
  ' my anniversary is {format(anniversary, "%A, %d %B, %Y")}.')
```

Remarque: l'objet créé est à la fois un objet de type **glue** et **character** en même temps. La particularité de ce type d'objets est qu'il hérite de toutes les fonctions qui peuvent s'appliquer à ces deux types:

```
class(new_object)
```

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

```
new_object
```

```
## My name is Fred, my age next year is 31, my anniversary is samedi, 12 octobre, 1991.
```

Il est important de noter qu'une fois l'objet **glue** créé, sa valeur est fixée. Autrement dit, même si on modifie les objets qui ont été utilisés pour le créer, cela ne le modifiera (à moins bien sûr de re-exécuter la commande). Exemple :

```
name <- "Jojo"
new_object
```

```
## My name is Fred, my age next year is 31, my anniversary is samedi, 12 octobre, 1991.
```

```
new_object <- glue('My name is {name},',
  ' my age next year is {age + 1},',
  ' my anniversary is {format(anniversary, "%A, %d %B, %Y")}.')
new_object
```

```
## My name is Jojo, my age next year is 31, my anniversary is samedi, 12 octobre, 1991.
```

3 Les facteurs

Même s'ils peuvent a priori ressembler à des chaînes de caractères, les **factor** ont un comportement différent. Cette classe d'objet a été créé pour correspondre à une variable qualitative.

On commence par créer un vecteur de chaîne de caractères :

```
genre <- sample(c("Ctrl", "Trait"), size = 10000, replace = TRUE)
```

Puis, on le transforme en **factor** en utilisant la fonction *factor()* ou *as.factor()* :

```
genre_fact <- factor(genre)
```

Les facteurs ont des modalités pré-définies qui sont retournées avec la fonction *levels()*. L'affectation d'une valeur différente de ces modalités pré-définies provoque un message d'avertissement et une valeur manquante dans le vecteur :

```
levels(genre_fact)
```

```
## [1] "Ctrl"  "Trait"
```

```
genre_fact[1] <- "Autre"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "Autre"): niveau de facteur
## incorrect, NAs générés
```

```
genre_fact[1]
```

```
## [1] <NA>
## Levels: Ctrl Trait
```

ce qui n'est bien entendu pas le cas pour les vecteurs de caractères :

```
genre[1] <- "Autre"
genre[1]
```

```
## [1] "Autre"
```

Dans le cas de vecteurs de taille importantes, le stockage d'un **factor** est un peu moins volumineux qu'un objet **character**. Ici, on utilise la fonction *object.size()* qui indique la taille allouée à un objet en mémoire vive :

```
object.size(genre)
```

```
## 80216 bytes
```

```
object.size(genre_fact)
```

```
## 40560 bytes
```

L'exemple suivant permet de mieux comprendre qu'un **factor** peut être considéré comme un vecteur de valeurs entières où chaque entier pourrait être remplacé par un **label**. Dans le cas où l'on souhaite associer un **label**, on procède de la façon suivante.

```
vec <- sample(1:4, size = 20, rep = T)
(f_vec <- factor(vec, levels = 1:3,
                      labels = c("Rien", "Peu", "Beaucoup")))
```

```
##  [1] <NA>     Beaucoup Beaucoup Peu      Peu      Beaucoup <NA>     Rien
##  [9] Beaucoup Peu      Beaucoup Peu      Beaucoup <NA>     Peu      <NA>
## [17] Beaucoup Beaucoup Peu      Rien
## Levels: Rien Peu Beaucoup
```

Remarque : si on oublie d'associer un **level** à un **label**, cela a pour conséquence de créer une valeur manquante.

La fonction *cut()* qui permet le recodage d'une variable quantitative en classes, génère un objet de type **factor**. On précise les amplitudes des classes avec l'option **breaks** :

```
set.seed(123)
mesures <- rnorm(100)
codage <- cut(mesures, breaks = -4:4)
table(codage)
```

```
## codage
## (-4,-3] (-3,-2] (-2,-1]  (-1,0]   (0,1]   (1,2]   (2,3]   (3,4]
##       0       1      13      34      35      14       3       0
```

Pour aider à trouver un découpage d'une variable quantitative, la fonction *classIntervals()* du package **classInt** propose différentes méthodes de discrétisation d'une variable quantitative. Parmi ces méthodes, on compte la méthode basée sur des classes d'amplitudes égales, d'effectifs égaux, ou calculées à partir de l'algorithme des k-means :

```
require("classInt")
codage2 <- cut(mesures,
           classIntervals(mesures, n = 5, style = "kmeans")$brks)
table(codage2)
```

```
## codage2
##   (-2.31,-0.874] (-0.874,-0.0726]  (-0.0726,0.614]     (0.614,1.44]
##               13               31               28               19
##      (1.44,2.19]
##                8
```

Remarque : les fonctions classiques d'importation des jeux de données codent les variables qualitatives sous forme de **factor**. Cet aspect a été critiqué par certains programmeurs dont Hadley Wickam (nous en verrons les raisons un peu plus tard), qui préconise un codage des variables qualitatives sous forme de **character**. Toutefois, un des avantages de la classe **factor** est que cela permet de représenter les modalités d'une variable qualitative de façon ordonnée, ce qui se révelera très pratique pour représenter des diagramme en barres par exemple..

Exercice 1.5.

- **Q1. Facteur ordonné:**

  - Créer un vecteur de chaîne de caractères **doses** de taille 25, comprenant les valeurs “faible”, “moyenne” ou “forte” (on pourra créer ce vecteur de façon aléatoire).
  - Convertir cet élément en **f_dose** de type **factor** ordonné (voir si besoin *?factor*).
  - Vérifier que les niveaux du facteur sont effectivement ordonnés. Pour cela, il suffit de comparer d'utiliser les opérateurs de comparaison $<$ ou $>$ sur deux composantes du vecteur.
- **Q2. Codage et comptage :** donner un équivalent de l'enchaînement des fonctions *cut()* et *table()* utilisées précédemment.

4 Les dates

4.1 Dates et unités de temps

Il existe plusieurs packages pour manipuler des données temporelles ; voir la Task View Time Series Analysis. Nous nous contentons ici de présenter quelques manipulations élémentaires. Une référence sur le sujet :

- G. Grothendieck and T. Petzoldt (2004), **R** Help Desk: Date and Time Classes in **R**, **R** News 4(1), 29-32 (https://www.r-project.org/doc/Rnews/Rnews_2004-1.pdf).

Dans **R**, une façon naturelle de manipuler des dates est d'utiliser la classe d'objet **Date**. Voici l'allure d'un objet de classe **Date** :

```
(format.Date <- Sys.Date())
```

```
## [1] "2021-10-06"
```

```
class(format.Date)
```

```
## [1] "Date"
```

En gros, il s'agit d'une chaîne de caractère de la forme **“YYYY-MM-DD”**, parfois **“YYYY/MM/DD”** ou encore **“MM/DD/YY”**. Pour passer d'une chaîne de caractère à un objet de classe **Date**, il faut donc préciser où sont placés les années, mois et jours dans la chaîne de caractère, préciser si les mois sont des valeurs numériques ou écrit en lettre, etc. Par exemple :

```
dates <- c("01/01/17", "02/03/17", "03/05/17")
as.Date(dates, "%d/%m/%y")
```

```
## [1] "2017-01-01" "2017-03-02" "2017-05-03"
```

```
dates <- c("1 janvier 2017", "2 mars 2017", "3 mai 2017")
as.Date(dates, "%d %B %Y")
```

```
## [1] "2017-01-01" "2017-03-02" "2017-05-03"
```

Les informations sur la façon dont convertir les chaînes de caractères en objet **Date** sont données dans l'aide suivante :

```
?format.Date
```

Une autre classe d'objet utile est la classe **POSIXct/POSIXt**. Le format **POSIXct/POSIXlt** est plus précis que le format **Date** car il mesure à la fois la date et l'heure. Ce format est notamment utilisé dans les séries temporelles d'indices boursiers.

```
(format.POSIXlt <- Sys.time())
```

```
## [1] "2021-10-06 19:08:44 CEST"
```

```
class(format.POSIXlt)
```

```
## [1] "POSIXct" "POSIXt"
```

Un certain nombre de fonctions de **R** reconnaissent ce type de format. On en cite ici quelques-unes :

```
weekdays(format.POSIXlt)
```

```
## [1] "mercredi"
```

```
months(format.POSIXlt)
```

```
## [1] "octobre"
```

```
quarters(format.POSIXlt)
```

```
## [1] "Q4"
```

De même que pour les dates, il est possible de convertir des chaînes de caractères en **POSIXct/POSIXlt** ou de faire l'inverse. Pour cela, il faut respecter la nomenclature des dates (voir l'aide en ligne *?strptime*). Pour convertir une chaîne de caractères en **POSIXct/POSIXlt** :

```
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
times <- c("23:03:20", "22:29:56", "01:03:30", "18:21:03", "16:56:26")
x <- paste(dates, times)
strptime(x, "%m/%d/%y %H:%M:%S")
```

```
## [1] "1992-02-27 23:03:20 CET" "1992-02-27 22:29:56 CET"
## [3] "1992-01-14 01:03:30 CET" "1992-02-28 18:21:03 CET"
## [5] "1992-02-01 16:56:26 CET"
```

Pour faire l'inverse (transformer un **POSIXct/POSIXlt** en **character**) :

```
(z <- Sys.time())
```

```
## [1] "2021-10-06 19:08:44 CEST"
```

```
format(z,"%a %d %b %Y %X %Z")
```

```
## [1] "mer. 06 oct. 2021 19:08:44 CEST"
```

La fonction *system.time()* renvoie plusieurs informations concernant le temps de calcul d'une commande :

- *Utilisateur* : il s'agit du temps mis par l'ordinateur pour exécuter directement le code donné par l'utilisateur,
- *Système* : il s'agit du temps utilisé pas directement par le calcul, mais par le système (ex: gestion des entrées/sorties, écriture sur le disque, etc.) lié au code qui doit être exécuté.
- *écoulé* : il s'agit du temps Utilisateur + Système C'est en général ce dernier qui est utilisé pour comparer des temps de calcul.

```
system.time(for(i in 1:100) var(runif(100000)))
```

```
## utilisateur     système      écoulé
##       0.221       0.028       0.249
```

Remarque : il est parfois compliqué de convertir des chaînes de caractères en **Date** ou **POSIXct/POSIXlt**, mais une fois que cela est fait, il est alors possible d'utiliser un nombre conséquent de packages existants, qui permettent en autre la manipulation de séries temporelles.

Exercice 1.6.

Q1 Quel jour (lundi, mardi ... ?) sera le 1er janvier de l'année 2022 ?

Q2 Combien de jours nous séparent du 31 décembre de l'année en cours

4.2 Séries temporelles

Nous faisons ici un aparte pour évoquer la manipulation de séries temporelles.

On commence par simuler deux séries temporelles issues respectivement d'un processus AR(2) et MA(2). Nous ne rentrerons pas dans le détail de ces modèles mais le lecteur pourra se référer à l'ouvrage d'Yves Aragon "Séries temporelles avec **R**" pour une introduction. On utilise la fonction *set.seed()* avant chaque simulation, ce qui va nous permettre de générer la même séquence dès lors qu'on utilisera les mêmes entiers comme argument de la fonction (493 et 494 ici).

```
set.seed(493)
x1 <- arima.sim(model = list(ar = c(.9, -.2)), n = 100)
set.seed(494)
x2 <- arima.sim(model = list(ma = c(-.7, .1)), n = 100)
```

Le principe d'une série temporelle est que les observations sont associées à une date et dans certains cas une date et un temps (indices boursiers observés en temps réel). Pour faire simple, on va associer la série à des dates journalières commencant le 1er octobre 2017 et de taille 100. Une façon de faire est d'utiliser la fonction *seq.Date()* :

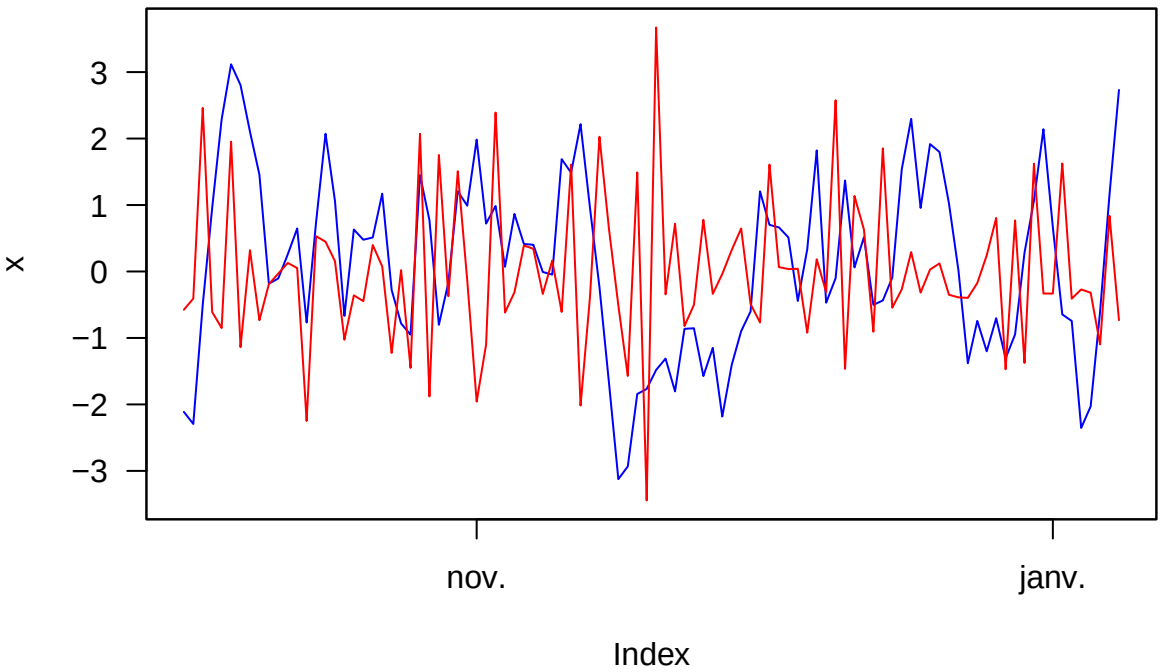
```
date_x <- seq.Date(as.Date("2017-10-01"), by = "day", len = 100)
```

Ensuite, on utilise la fonction *zoo()* du package qui porte le même nom pour associer les séries aux dates précédemment créées :

```
require("zoo")
x <- zoo(cbind(x1, x2), date_x)
```

L'objet précédemment créé peut ensuite être appliqué aux fonctions du package **zoo**. Ainsi, en appliquant la fonction *plot()* à un tel objet, cela a pour avantage d'afficher en abscisses les dates, d'afficher plusieurs courbes s'il s'agit de séries temporelles multidimensionnelles, etc. On pourra consulter les fonctions du package **zoo** pour plus d'informations (*help(package="zoo")*)

```
par(las = 1)
plot(x, col = c("blue", "red"), screens = 1)
```



Remarque : l'option **las = 1** dans la fonction *par()* permet d'afficher la légende de l'axe des ordonnées

horizontalement. L'options **screens = 1** permet de représenter les deux séries dans la meme figure.

5 Opérations ensemblistes

On définit deux vecteurs A et B d'entiers.

```
(A <- 1:10)
```

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

```
(B <- c(3:6, 12, 15, 18))
```

```
## [1]  3  4  5  6 12 15 18
```

Les opérations ensemblistes classiques sont :

5.1 L'union

```
union(A, B)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 12 15 18
```

équivalent à la syntaxe suivante écrite avec des fonctions de base:

```
unique(c(A, B))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 12 15 18
```

5.2 L'intersection

- fonction *intersect()*

```
intersect(A, B)
```

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

équivalent à la syntaxe suivante écrite avec des fonctions de base:

```
A[A %in% B]
```

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

- fonction *match()*

Ici, il nous semble important de parler de la fonction *match()* qui a été utilisée pour coder la fonction *intersect()*. Dans le code ci-dessous, elle retourne pour chaque élément de **A** s'il se trouve dans **B** et si oui à quelle position dans **B**. Ci-dessous, le 3ème élement de **A** est bien dans **B** et il se situe à la 1ère position de **B**.

```
match(A, B)
```

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

Il est important de rappeler que l'opérateur **%in%** fait appel à la fonction *match()*. Pour afficher le code de la fonction **%in%**, on peut utiliser la syntaxe suivante:

```
`%in%`
```

```
## function (x, table)
## match(x, table, nomatch = 0L) > 0L
## <bytecode: 0x564032254818>
## <environment: namespace:base>
```

5.3 La différence ($A - B$, différent de $B - A$)

```
setdiff(A, B)
```

```
## [1]  1  2  7  8  9 10
```

```
setdiff(B, A)
```

```
## [1] 12 15 18
```

Pour savoir si un ou plusieurs éléments sont contenus dans un ensemble :

```
is.element(2, A)
```

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

```
is.element(2, B)
```

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

```
is.element(A, B)
```

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

```
is.element(B, A)
```

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

Attention : on utilise ces fonctions sur des objets de même classe. Si on fait l'union d'un vecteur d'entiers avec un vecteur de caractères, le vecteur d'entiers sera transformé en caractères. Par exemple :

```
let <- letters[1:10]
union(A, let)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "a"  "b"  "c"  "d"  "e"
## [16] "f"  "g"  "h"  "i"  "j"
```

Il est donc possible de faire des opérations ensemblistes sur les chaînes de caractères.

Exercice 1.7.

Q1 : donner une notation équivalente à *is.element(2, A)*.

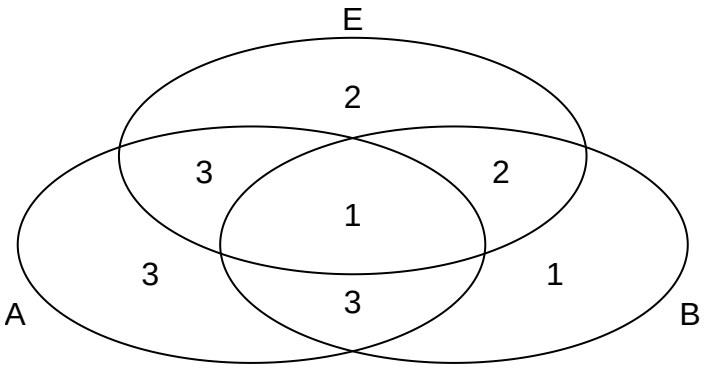
Q2 : l'objet **letters** est un vecteur de longueur 26 qui contient les lettres de l'alphabet. Tester l'appartenance des lettres **k** et **m** à l'alphabet ; que renvoie la syntaxe "réciproque" (appartenance de l'alphabet à l'ensemble *c("k", "m")*) ? Comment obtenir le rang des lettres **k** et **m** dans l'alphabet ?

Q3 : en plus des 2 vecteurs **A** et **B** définis précédemment, considérons le vecteur **E** (nous évitons la lettre **C** qui existe déjà dans **R**, voir *help(C)*) suivant :

```
E <- c(18, 1, 9, 14, 12, 6, 2, 19)
```

Donner l'enchaînement des commandes qui permettent de retrouver les valeurs disposées sur le diagramme de Venn ci-dessous (obtenu avec la fonction *venn()* du package **gplots**).

```
require("gplots")
venn(list(A = A, B = B, E = E))
```



6 Manipulation de bases de données

6.1 Jointure et agrégation

On considère une table **patient** qui contient des informations sur les patients et une table **visite** qui contient des informations sur les visites. La clé de référence est le nom du patient dont la variable ne porte pas le même nom dans les deux tables.

```
patient <- data.frame(
   nom.famille = c("René", "Jean", "Ginette", "Joseph"),
   ddn = c("02/02/1925", "03/03/1952", "01/10/1992", "02/02/1920"),
   sexe = c("m", "m", "f", "m"))
```

```
visite <- data.frame(
   nom = c("René", "Jean", "René", "Simone", "Ginette"),
   ddv = c("01/01/2020", "10/12/2020", "05/01/2020", "04/12/2020", "05/10/2020"))
```

Problématique : on souhaite avoir une table contenant l'âge du patient au moment de sa visite. Pour cela, on voit bien qu'il faut connaître la date de naissance du patient au moment de sa visite. L'idée est donc d'ajouter une colonne "ddn" à la table visite pour pouvoir ensuite calculer l'âge en faisant la différence entre la date de visite et la date naissance.

6.1.1 Jointure sans utiliser la fonction *merge()*

Dans un premier temps, nous allons essayer de répondre au problème sans utiliser la fonction *merge()*. Pour cela, nous allons utiliser la fonction *match()* vue précédemment. Elle va nous permettre de savoir où se trouvent dans la table **patient**, les patients qui ont eu des visites. Une fois les indices connus, on a plus qu'à sélectionner les dates de naissance des patients.

```
visite$ddn <- patient$ddn[match(visite$nom, patient$nom.famille)]
```

Ensuite, on calcule l'âge du patient :

```
visite$age <- round(as.numeric(as.Date(visite$ddv, format = "%d/%m/%Y") -
                               as.Date(visite$ddn, format = "%d/%m/%Y"))/365,
                    0)
```

Enfin, on efface la date de naissance qui ne nous intéresse pas ici :

```
visite$ddn <- NULL
head(visite)
```

```
##       nom        ddv age
## 1    René 01/01/2020  95
## 2    Jean 10/12/2020  69
```

```
## 3    René 05/01/2020  95
## 4  Simone 04/12/2020  NA
## 5 Ginette 05/10/2020  28
```

Pour la suite, on remet à jour la table **visite** :

```
visite$age <- NULL
```

6.1.2 Jointure en utilisant la fonction *merge()*

A présent, nous allons utiliser la fonction *merge()* qui permet de réaliser la jointure entre deux tables. Il est essentiel de préciser la clé de référence des deux tables avec l'option **by=** si les deux tables ont un nom de clé identique ou alors **by.x=** et **by.y=** si la clé de référence porte un nom différent selon la table.

Dans la première commande, le merge se fait sur les variables qui sont présentes dans les deux tables. Autrement dit, l'individu "Simone" n'est pas présente dans la nouvelle table compte tenu qu'elle n'est pas identifiée parmi les patients.

```
visite$age <- NULL
merge(visite, patient, by.x = "nom", by.y = "nom.famille")
```

```
##       nom        ddv        ddn sexe
## 1 Ginette 05/10/2020 01/10/1992    f
## 2    Jean 10/12/2020 03/03/1952    m
## 3    René 01/01/2020 02/02/1925    m
## 4    René 05/01/2020 02/02/1925    m
```

Si on souhaite garder toute l'information contenue dans le fichier visite, on ajoute l'option **all.x=TRUE**. Autrement dit, on affiche ici toutes les visites, y compris celles des personnes qui ne sont bas dans la table **patient**. On remarquera que Simone apparaît en queue de table :

```
merge(visite, patient, by.x = "nom", by.y = "nom.famille", all.x = TRUE)
```

```
##       nom        ddv        ddn sexe
## 1 Ginette 05/10/2020 01/10/1992    f
## 2    Jean 10/12/2020 03/03/1952    m
## 3    René 01/01/2020 02/02/1925    m
## 4    René 05/01/2020 02/02/1925    m
## 5  Simone 04/12/2020       <NA> <NA>
```

Enfin, si on souhaite conserver l'information dans les deux tables, on utilise **all.x=TRUE** et **all.y=TRUE** :

```
(visite.patient <- merge(visite, patient, by.x = "nom",
    by.y = "nom.famille", all.x = TRUE, all.y = TRUE))
```

```
##       nom        ddv        ddn sexe
## 1 Ginette 05/10/2020 01/10/1992    f
## 2    Jean 10/12/2020 03/03/1952    m
## 3  Joseph       <NA> 02/02/1920    m
## 4    René 01/01/2020 02/02/1925    m
## 5    René 05/01/2020 02/02/1925    m
## 6  Simone 04/12/2020       <NA> <NA>
```

Pour calculer l'âge du patient au moment de la visite, on utilise la commande vue précédemment :

```
visite.patient$age <- round(as.numeric(as.Date(visite.patient$ddv,
                                               format = "%d/%m/%Y") -
                               as.Date(visite.patient$ddn,
                                               format = "%d/%m/%Y"))/365,
                               0)
```

Maintenant, on souhaite connaître l'âge des patients lors de leurs visites en fonction du sexe. On va donc faire une aggrégation.

6.1.3 Aggrégation

Pour faire l'agrégation, on utilise la fonction *tapply()* pour aggréger une variable ou *aggregate()* pour plusieurs variables. Le principe de ces finctions est le suivant :

- 1. utiliser *split()* sur l'échantillon total pour découper en sous-échantillons selon la variable utilisée pour faire l'aggrégation:

```
my_split <- split(x = visite.patient$age, f = visite.patient$sexe)
```

- 2. utiliser *lapply()* ou *sapply()* pour calculer des statistiques sur chaque sous-échantillon

```
sapply(my_split, FUN = mean, na.rm = T)
```

```
##        f        m
## 28.00000 86.33333
```

En utilisant la fonction *tapply()*, les deux étapes précédentes sont réalisées à la suite :

```
tapply(X = visite.patient$age, INDEX = list(sexe = visite.patient$sexe),
       FUN = mean, na.rm = T)
```

```
## sexe
##        f        m
## 28.00000 86.33333
```

Si on utilise la fonction *aggregate()*, la variable qui permet d'aggréger doit être mise sous forme d'une liste dans l'option **by=**. L'option **FUN=** renseigne s'il s'agit de faire une somme, une moyenne, etc. Dans notre cas :

```
aggregate(visite.patient$age, by = list(S = visite.patient$sexe),
 FUN = mean, na.rm = TRUE)
```

```
##   S        x
## 1 f 28.00000
## 2 m 86.33333
```

Exercice 1.8.

Q1 à partir du jeu de données **iris** (disponible par défaut dans l'environnement courant), construire l'objet **iris1** qui contient en fonction des espèces (variable **Species**), la taille moyenne de la variable **Petal.Length**.

Q2 Construire l'objet **iris2** qui contient en fonction des espèces, la taille totale de la variable **Petal.Width**.

Q3 Faire le merge des jeux de données **iris1** et **iris2**.

6.2 Package dplyr

Ce document est inspiré d'une présentation de Sophie Lamarre aux rencontres des Ingénieurs statisticiens de Toulouse, disponible sur ce lien. On recommande aussi la lecture de la vignette du package, disponible sur le site du CRAN : https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html

Ce package est de plus en plus utilisé dans la manipulation de jeu de données, notamment parmi les "Data Scientists". Son principe est de simplifier la syntaxe des fonctions de base de **R** et d'utiliser du code **C++** derrière certaines de ses fonctions dans le but d'améliorer sensiblement les temps de calcul. Il fait partie du projet Tidyverse qui contient un ensemble de packages, développés en grande partie par **RStudio**. Avec la commande suivante, vous chargez un ensemble de packages dont certains très populaires (**ggplot2**, **dplyr**, etc.)

```
require("tidyverse")
```

```
## Le chargement a nécessité le package : tidyverse
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.2     v dplyr   1.0.6
## v tidyr   1.1.3     v forcats 0.5.1
## v readr   1.4.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::collapse() masks glue::collapse()
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
```

Description du jeu de données utilisées : il s'agit du jeu de données **diamonds** du package **ggplot2**. On observe sur un peu plus de 50000 diamants, un certain nombre de variables dont : le prix de vente, le poids, la qualité, la couleur, etc.

```
head(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

Voici le résumé statistique des variables :

```
summary(diamonds)
```

```
##      carat               cut        color        clarity          depth
##  Min.   :0.2000   Fair      : 1610   D: 6775   SI1    :13065   Min.   :43.00
##  1st Qu.:0.4000   Good      : 4906   E: 9797   VS2    :12258   1st Qu.:61.00
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194   Median :61.80
##  Mean   :0.7979   Premium   :13791   G:11292   VS1    : 8171   Mean   :61.75
##  3rd Qu.:1.0400   Ideal     :21551   H: 8304   VVS2   : 5066   3rd Qu.:62.50
##  Max.   :5.0100                     I: 5422   VVS1   : 3655   Max.   :79.00
##                                     J: 2808   (Other): 2531
##      table           price             x                y
##  Min.   :43.00   Min.   :  326   Min.   : 0.000   Min.   : 0.000
##  1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710   1st Qu.: 4.720
##  Median :57.00   Median : 2401   Median : 5.700   Median : 5.710
##  Mean   :57.46   Mean   : 3933   Mean   : 5.731   Mean   : 5.735
##  3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
```

```
##  Max.   :31.800
##
```

Remarque : le jeu de données **diamonds** est dans un format de données nouveau : **tibble**. Il s'agit encore une fois d'une nouveauté proposée par **RStudio**. L'objet de classe **data.frame** peut présenter certaines contraintes pour ses utilisateurs. Par exemple, lorsque'on souhaite afficher un **data.frame** dans la console, **R** affiche autant de lignes qu'il le peut. D'autre part, avec un **data.frame**, certains noms de variables ne sont pas autorisés. Par exemple, dans un **data.frame** un nom de variable ne peut pas commencer par un chiffre ce qui n'est pas le cas avec le **tibble** :

```
data.frame(`1a` = 1:10)
tibble(`1a` = 1:10)
```

C'est pourquoi des développeurs ont pensé à créer un nouveau type d'objets, le **tibble** qui ne présenterait pas ce genre de contraintes. Ce n'est pas une grande révolution, mais ce package faisant partie du projet **tidyverse**, c'est essentiellement ce type de données qui est utilisé dans cet univers. Les fonctions que nous allons présenter dans cette section s'appliquent aussi bien sur des **data.frame** que sur des **tibble**, ce dernier type d'objet ayant hérité des propriétés des **data.frame**. Pour en savoir plus sur les **tibbles**, vous pouvez consulter ce cours en ligne : http://r4ds.had.co.nz/tibbles.html.

6.2.1 Sélection de lignes avec la fonction *filter()*

La fonction "phare" du package **dplyr** est la fonction *filter()* qui permet de sélectionner un sous-échantillon du jeu de données à partir de critères qu'on lui donne. Par exemple, pour sélectionner uniquement les diamants d'une valeur supérieure à 15000 dollars et ayant une couleur **E** ou **F**, on écrit :

```
filtrage1 <- dplyr::filter(diamonds, price > 15000 & (color == "E" | color == "F"))
```

Ou de façon équivalente (les virgules remplaçant la condition **et**) :

```
filtrage1 <- filter(diamonds, price > 15000, (color == "E" | color == "F"))
head(filtrage1, 3)
```

```
## # A tibble: 3 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1.54 Premium   E     VS2      62.3    58 15002  7.31  7.39  4.58
## 2  1.19 Ideal     F     VVS1     61.5    55 15005  6.82  6.84  4.2
## 3  2.05 Very Good F     SI2      61.9    56 15017  8.13  8.18  5.05
```

Une nouveauté est également l'utilisation de l'opérteur %>% dit 'pipe' en anglais. Il se trouve après un **data.frame** et indique qu'on va utiliser une fonction dont le premier argument est un objet de type **data.frame**. Dans ce cas, ce n'est plus la peine d'indiquer le premier argument de la fonction *filter()*. Nous verrons par la suite son intérêt lorsqu'on souhaite appliquer successivement des commandes.

```
filtrage1 <- diamonds %>%
  filter(price > 15000, (color == "E" | color == "F"))
head(filtrage1, 3)
```

```
## # A tibble: 3 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1.54 Premium   E     VS2      62.3    58 15002  7.31  7.39  4.58
## 2  1.19 Ideal     F     VVS1     61.5    55 15005  6.82  6.84  4.2
## 3  2.05 Very Good F     SI2      61.9    56 15017  8.13  8.18  5.05
```

Remarque: sans utiliser le package **dplyr**, on aurait du faire quelque chose comme ça :

```
filtrage1 <- with(diamonds, diamonds[price > 15000 &
                                       (color == "E" | color == "F"), ])
```

On aurait également pu utiliser la fonction de base *subset()*dont le package **dplyr** s'est fortement inspiré.

```
filtrage1 <- subset(diamonds, price > 15000 & (color == "E" | color == "F"))
```

6.2.2 Trier le jeu de données avec la fonction *arrange()*

On souhaite trier le jeu de données en fonction de la coupe, de la couleur et du prix (en valeur décroissante). Pour cela, on fait :

```
tri1 <- arrange(diamonds, cut, color, desc(price))
head(tri1,3)
```

```
## # A tibble: 3 x 10
##   carat cut   color clarity depth table price     x     y     z
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  2.02 Fair  D     SI1      65      55 16386  7.94  7.84  5.13
## 2  2.01 Fair  D     SI2      66.9    57 16086  7.87  7.76  5.23
## 3  3.4  Fair  D     I1       66.8    52 15964  9.42  9.34  6.27
```

Sans le package **dplyr**, on aurait fait :

```
tri1 <- with(diamonds, diamonds[order(cut, color, -price),])
```

6.2.3 Sélectionner certaines variables avec la fonction *select()*

On ne souhaite garder que les variables **carat**, **price**, **color**, **z** et les variables dont le label contient le caractères **i** :

```
selection1 <- select(diamonds, carat, price, color, z,
                     dplyr::contains("i"))
head(selection1,3)
```

```
## # A tibble: 3 x 5
##   carat price color     z clarity
##   <dbl> <int> <ord> <dbl> <ord>
## 1  0.23   326 E      2.43 SI2
## 2  0.21   326 E      2.31 SI1
## 3  0.23   327 E      2.31 VS1
```

Sans le package **dplyr**, on aurait fait :

```
selection1 <- diamonds[, unique(c("carat", "price", "color", "z",
                                  names(diamonds)[grep("i", names(diamonds))]))]
```

Remarque : dans la deuxième syntaxe, pour que la variable **price** n'apparaisse qu'une seule fois dans le jeu de données, on a du utiliser la fonction *unique()*.

6.2.4 Changer le nom de variables avec la fonction *rename()*

On souhaite changer le nom de la variable **z** par le label **width** et **color** par **code_color** :

```
renom1 <- rename(diamonds, width = z, code_color = color)
head(renom1, 3)
```

```
## # A tibble: 3 x 10
##   carat cut     code_color clarity depth table price     x     y width
```

```
##   <dbl> <ord>   <ord>      <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E          SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E          SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good    E          VS1      56.9    65   327  4.05  4.07  2.31
```

Sans le package **dplyr**, on aurait fait :

```
names(diamonds)[c(match("z", names(diamonds)),
                  match("color", names(diamonds)))] <-
  c("width", "code_color")
```

6.2.5 Ajouter une nouvelle colonne avec la fonction *mutate()*

On calcule le prix au kilo pour chaque diamant et on convertit en euros :

```
calcul1 <- mutate(diamonds, prix.kilo = price / carat,
                  prix.kilo.euro = prix.kilo * 0.9035)
head(calcul1, 3)
```

```
## # A tibble: 3 x 12
##   carat cut     code_color clarity depth table price     x     y width prix.kilo
##   <dbl> <ord>   <ord>      <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>     <dbl>
## 1  0.23 Ideal   E          SI2      61.5    55   326  3.95  3.98  2.43     1417.
## 2  0.21 Premium E          SI1      59.8    61   326  3.89  3.84  2.31     1552.
## 3  0.23 Good    E          VS1      56.9    65   327  4.05  4.07  2.31     1422.
## # ... with 1 more variable: prix.kilo.euro <dbl>
```

Sans le package **dplyr**, on aurait fait :

```
diamonds$prix.kilo <- diamonds$price/diamonds$carat
diamonds$prix.kilo.euro <- diamonds$prix.kilo * 0.9035
```

6.2.6 Faire des calculs statistiques avec la fonction *summarise()*

On calcule le prix moyen en fonction de la couleur et de la coupe du diamant :

```
calcul2 <- summarise(group_by(diamonds, cut, code_color), prix.moy = mean(price))
```

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

```
head(calcul2, 3)
```

```
## # A tibble: 3 x 3
## # Groups:   cut [1]
##   cut   code_color prix.moy
##   <ord> <ord>         <dbl>
## 1 Fair  D             4291.
## 2 Fair  E             3682.
## 3 Fair  F             3827.
```

```
class(calcul2)
```

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

Sans le package **dplyr**, on aurait fait :

```
calcul2 <- aggregate(data.frame(price = diamonds[, "price"]),
                     list(color = diamonds$code_color,
                          cut = diamonds$cut), mean)
```

6.2.7 Faire des calculs statistiques en utilisant une nouvelle syntaxe

On souhaite calculer le prix maximum d'un diamant en fonction de la couleur et de la coupe. On ne veut garder que les prix supérieurs à 5000 dollars. Pour cela, on peut utiliser la syntaxe suivante, dite "pipelining' ', qui provient du package **magrittr** (et importé par défaut via la package **dplyr**) :

```
diamonds %>%
  group_by(code_color, cut) %>%
  summarise(prix_groupe = mean(price)) %>%
  filter(prix_groupe > 5000)
```

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

```
## # A tibble: 7 x 3
## # Groups:   code_color [3]
##   code_color cut       prix_groupe
##   <ord>      <ord>           <dbl>
## 1 H          Fair            5136.
## 2 H          Premium         5217.
## 3 I          Good            5079.
## 4 I          Very Good       5256.
## 5 I          Premium         5946.
## 6 J          Very Good       5104.
## 7 J          Premium         6295.
```

L'idée de cette syntaxe est de pouvoir comprendre rapidement les différentes opérations qui sont effectuées les unes à la suite des autres. Dans l'exemple précédent :

- on prend le jeu de données **diamonds**, le fait d'ajouter un "pipe' ' à la suite implique qu'on va appliquer une première opération dessus,
- *group_by* : on regroupe des observations en fonction des variables **color** et **cut**,
- *summarise* : on calcule le prix moyen sur les groupes précédemment créés,
- *filter* : on ne garde que les observations supérieures à 5000 dollars.

Remarque : depuis la version **4.1.0**, **R** a lancé une version native de l'opérateur '`pipe''. Il s'agit de la commande suivante|>`'. Voici un exemple d'utilisation :

```
my_vec <- rnorm(10)
my_vec |>
  mean()
```

```
## [1] -0.4518699
```

Dans la plupart des cas, il peut remplacer le pipe de **magrittr**.

6.2.8 Tirer un échantillon d'une population

On souhaite tirer de façon aléatoire 100 observations :

```
tirage1 <- sample_n(diamonds, 100)
```

On souhaite tirer de façon aléatoire 5% de l'échantillon :

```
tirage1 <- sample_frac(diamonds, 0.05)
```

7 Tidy data

Pour illustrer ce paragraphe, on considére les données suivantes : on observe pour 3 pays (Afghanistan, Brazil et Chine) sur 2 années consécutives (1999 et 2000), la taille de la population ainsi que le nombre de cas de

tuberculoses observés. On va voir qu'on peut utiliser différentes tables pour présenter ces données et parmi ces différentes possibilités, on aura intérêt à en utiliser une plutôt que les autres. Pour charger ces données, on va installer le package **DSR** accessible depuis **github**. En effet, il est possible de récupérer sur **github** des packages qui sont en cours de développement et qui n'ont pas encore passés le processus de validation pour être un package officiel du CRAN :

```
devtools::install_github("garrettgman/DSR")
```

On pourrait traduire *tidy data* par données rangées en opposition à *messy data*, données désordonnées. Ce courant de *tidy data* vient encore de Hadley Wickham (voir article https://www.jstatsoft.org/article/view/v059i10 ou encore https://r4ds.had.co.nz/tidy-data.html) qui part du principe suivant qui peut paraître évident, mais selon les situations, ce n'est pas toujours le cas :

- Une variable doit être rangée dans une colonne,
- Un individu est rangé dans une ligne,
- On place les valeurs observées dans les bonnes cases.

On présente ici une façon de présenter les données *tidy* :

```
DSR::table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <fct>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Au contraire, dans les données *messy*, on trouve en général une de ces situations :

- le nom des colonnes sont des valeurs et pas des noms,
- Plusieurs variables sont stockées dans une même colonne, c'est le cas du jeu de données suivant.

```
DSR::table3
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <fct>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

- Les variables sont à la fois présentes dans les lignes et les colonnes, c'est le cas du jeu de données suivant où constate que la colonne **key** contient le nom des deux variables **population** et **cases**, qu'il semblerait judiciable de présenter plutôt en colonnes.

```
DSR::table2
```

```
## # A tibble: 12 x 4
##    country      year key             value
##    <fct>       <int> <fct>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
```

```
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

- Les valeurs sont stockées dans plusieurs tables. C'est le cas de ces deux tables (une contenant les informations sur la population, l'autre sur le nombre de cas de tuberculoses) qui auraient pu être regroupées en une seule table

```
DSR::table4
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <fct>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

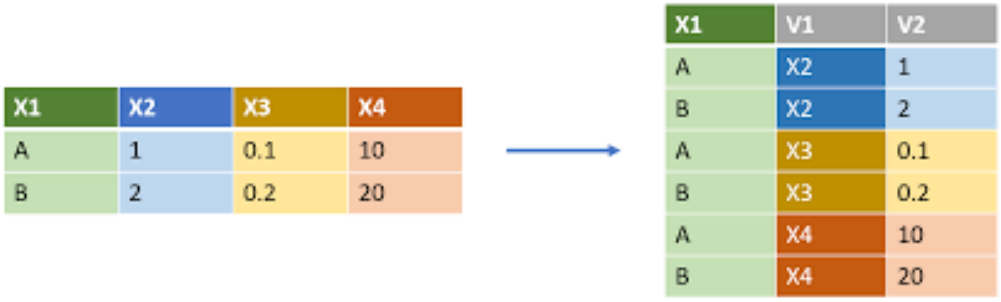
```
DSR::table5
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
##   <fct>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

On va présenter ici les fonctions principales du package **tidyr** (ce package fait partie de la bibliothèque **tidyverse**) qui permettent de passer d'un format *messy* à un format *tidy* ou inversement.

7.1 La fonction *pivot_longer()* : transformer des colonnes en lignes

Cette fonction permet de représenter en 1 seul colonne (et d'ajouter une colonne correspondant à une variable qualitative où les modalités sont le nom des variables initiales), une variable contenue à l'origine dans plusieurs colonnes.



L'argument **cols** indique les variables à transformer en ligne, l'argument **names_to** correspond au nom donné à la variable contenant les nouvelles modalités (le nom des variables) et l'argument **values_to** correspond au nom de la colonne contenant les valeurs qui ont été transformées de colonnes en lignes.

```
pivot_longer(DSR::table4,
             cols = c("1999", "2000"),
             names_to = "years",
             values_to = "cases")
```

```
## # A tibble: 6 x 3
##   country     years  cases
##   <fct>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

Avant *pivot_longer()*, il était possible d'utiliser la fonction *gather()*:

```
gather(DSR::table4, "year", "cases", 2:3)
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <fct>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000    2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

Avant toutes ces fonctions, il aurait fallu exécuter ce genre de commandes où l'opération mathématique consiste à transformer une matrice en un vecteur.

```
data.frame(
  country = rep(DSR::table4$country, times = 2),
  year = rep(c("1999", "2000"), each = nrow(DSR::table4)),
  cases = as.vector(as.matrix(DSR::table4[ , c("1999", "2000")]))
)
```

```
##       country year  cases
## 1 Afghanistan 1999    745
## 2      Brazil 1999  37737
## 3       China 1999 212258
## 4 Afghanistan 2000   2666
## 5      Brazil 2000  80488
## 6       China 2000 213766
```

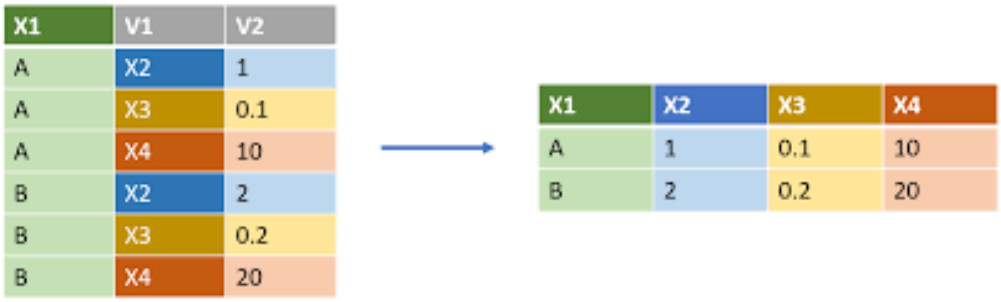
7.2 La fonction *pivot_wider()* : transformer des lignes en colonnes

Cette fonction permet de re-distribuer les valeurs d'une colonne qui contenait l'information de plusieurs variables, en plusieurs colonnes où chaque colonne correspond à une variable.

L'argument **names_from** correspond au nom de la colonne qui contient le nom des variables, et l'argument **values_from** correspond au nom de la colonne qui contient les valeurs à re-distribuer :

```
pivot_wider(DSR::table2, names_from = key,
            values_from = value)
```

```
## # A tibble: 6 x 4
```




```
##   country      year  cases population
##   <fct>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Avant *pivot_wider()*, il était possible d'utiliser la fonction *spread()*

```
spread(DSR::table2, key, value)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <fct>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Avant la création de ces fonctions, il aurait fallu utiliser la fonction *split()* et *merge()*:

```
sp_DSR <- split(DSR::table2[, c(1, 2, 4)],  DSR::table2[, 3])
names(sp_DSR$cases)[3] <- "cases"
names(sp_DSR$population)[3] <- "population"
merge(sp_DSR$cases, sp_DSR$population,
      by.x = c("country", "year"),
      by.y = c("country", "year"),)
```

```
##       country year  cases population
## 1 Afghanistan 1999    745   19987071
## 2 Afghanistan 2000   2666   20595360
## 3      Brazil 1999  37737  172006362
## 4      Brazil 2000  80488  174504898
## 5       China 1999 212258 1272915272
## 6       China 2000 213766 1280428583
```

7.2.1 La fonction *separate()*

Cette fonction permet de séparer une colonne en deux variables dès qu'elle détecte un caractère de séparation. Par exemple, pour spliter la colonne **rate** de la **table3**.

```
separate(table3, rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

7.2.2 La fonction *unite()*

Cette fonction permet de faire l'opération inverse de *separate()*. Elle permet de concaténer deux variables. Le résultat est proche de celui de la fonction *paste()*.

```
unite(DSR::table1, col = "rate",
      cases, population, sep = "/")
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <fct>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

7.2.3 La fonction *extract()*

Cette fonction permet de spliter une colonne en deux en précisant quelle chaîne de caractère est utilisée pour le split. Dans l'exemple ci-dessous, on splitte une chaîne de caractère lorsqu'il y a un espace dans une chaîne de caractère. Pour cela, on utilise l'argument **regex=** qui indique une expression régulière "REGEX".

```
df_to_split <- data.frame(
  player = c("Lionel Messi", "Christiano Ronaldo", "Antoine Griezman"),
  prix = c(170, 100, 120))
extract(df_to_split,
        col = player,
        into = c("prenom", "nom"),
        regex = "^(.).* (.).*$")
```

```
##   prenom nom prix
## 1      L   M  170
## 2      C   R  100
## 3      A   G  120
```

Ceci aurait pu se faire en utilisant la fonction *strsplit()*.

7.2.4 La fonction *complete()*

Cette fonction permet d'ajouter des lignes dans le cas où une valeur serait manquante. Par exemple, si on considère le jeu de données suivant, on constate que la firme B n'a pas de valeurs de CA pour l'année 2009.

```
firms <- data.frame(
  firms = c("A", "A", "B"),
  years = c("2008", "2009", "2008"),
  CA = c(20000, 25000, 40000)
)
```

Pour changer cela, on utilise la fonction *complete()* de la façon suivante:

```
complete(firms, firms, years)
```

```
## # A tibble: 4 x 3
##   firms years    CA
##   <chr> <chr> <dbl>
## 1 A     2008  20000
## 2 A     2009  25000
## 3 B     2008  40000
## 4 B     2009     NA
```

Pour plus de documentation sur l'univers **tidyr**, on recommande la lecture de cette page écrite par Julien Barnier.

Exercice 1.9.

Q1 Découper en 3 variables (**ville**, **num**, **dep**), le vecteur suivant :

```
code_INSEE <- c("toulouse_31_HG", "lyon_69_Rhone", "marsei_13_PACA")
```

8 Gestion de données volumineuses

Ici, nous allons essentiellement parler de données "moyennement volumineuses", à savoir des données qui prennent entre 1 et 2 Go de RAM, ce qui correspond très approximativement à des jeux de données contenant quelques millions de lignes et quelques dizaines de variables. Nous parlerons des packages qui permettent de traiter des bases de données plus volumineuses, sans rentrer dans les détails.

8.1 Optimiser la fonction *read.table()*

8.1.1 Astuce 1

Une première astuce concernant la gestion de données volumineuses consiste à mieux gérer l'importation des données. Regardons ce que cela donne avec un fichier contenant 500000 lignes et 3 colonnes.

- Commençons par générer des données :

```
n <- 500000 # à modifier selon la mémoire vive de votre machine
donnees_a_importer <- data.frame(chiffre = 1:n,
 lettre = paste0("caract", 1:n),
 date = sample(seq.Date(as.Date("2017-10-01"), by = "day", len = 100), n,
               replace = T))
object.size(donnees_a_importer)
```

```
## 40001136 bytes
```

```
str(donnees_a_importer)
```

```
## 'data.frame':    500000 obs. of  3 variables:
##  $ chiffre: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ lettre : chr  "caract1" "caract2" "caract3" "caract4" ...
##  $ date   : Date, format: "2017-12-04" "2017-10-01" ...
```

Ce jeu de données utilise environ 38Mo de mémoire vive ou RAM (pour un rappel sur les différents types de mémoire, voir ce tutoriel intéressant). Il contient 3 variables : la 1ère au format **integer** (un **integer** occupe moins de mémoires qu'un **double**), la seconde au format **factor**, le troisième au format **Date**.

- Exportons ces données dans un fichier avec la fonction *write.table()* :

```
write.table(donnees_a_importer, "fichier.txt", row.names = F)
```

On peut connaître la taille du fichier en mémoire disque :

```
file.info("fichier.txt")
```

On dispose ainsi d'un fichier appelé *fichier.txt* dans l'espace de travail, qui pèse environ 16Mo.

- Importons ces données avec la fonction *read.table()* en mesurant le temps pris pour accomplir cette action. On remarque que les 2ème et 3ème variables ont été stockées en tant que **factor** (parce que on lui a demandé en utilisant l'option **stringsAsFactors = TRUE**).

```
system.time(import1 <- read.table("fichier.txt", header = T,
                                  stringsAsFactors = TRUE))
```

```
## utilisateur     système      écoulé
##       2.029       0.001       2.030
```

```
str(import1)
```

```
## 'data.frame':    500000 obs. of  3 variables:
##  $ chiffre: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ lettre : Factor w/ 500000 levels "caract1","caract10",..: 1 111112 222223 333334 444445 455557 46
##  $ date   : Factor w/ 100 levels "2017-10-01","2017-10-02",..: 65 1 56 95 64 100 21 62 15 34 ...
```

- Exécutons la même opération en précisant que les chaînes de caractères seront stockées sous forme de **character**.

```
system.time(import2 <- read.table("fichier.txt", header = T,
                                  stringsAsFactors = FALSE))
```

```
## utilisateur     système      écoulé
##       0.326       0.000       0.326
```

```
str(import2)
```

```
## 'data.frame':    500000 obs. of  3 variables:
##  $ chiffre: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ lettre : chr  "caract1" "caract2" "caract3" "caract4" ...
##  $ date   : chr  "2017-12-04" "2017-10-01" "2017-11-25" "2018-01-03" ...
```

Le gain de temps dans le second cas vient du fait qu'on demande à ce que les variables qualitatives ne soient pas codées en **factor**. En effet, pour créer un **factor**, **R** a besoin de parcourir l'ensemble du fichier pour identifier tous les **levels** possibles, pour pouvoir attribuer un numéro à chaque **levels**. Cette procédure n'est évidemment pas optimale et il s'agissait d'une critique majeure concernant les objets **data.frame**, à savoir que les variables qualitatives sont codées par défaut en **factor** jusqu'à 2019. Depuis la version 4.0.0., l'argument *stringsAsFactors** vaut :

```
default.stringsAsFactors()
```

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

En effet, depuis la conférence useR!2019 qui s'est tenue à Toulouse, les chaînes de caractères sont à présent sauvegardées au format **character**. L'idée avancée était qu'un **factor** ordonne les caractères selon des règles lexicographiques qui ne sont pas les mêmes d'un pays à un autre. Autrement dit, compte tenu que ces règles

sont définies par la machine utilisée, on peut obtenir des résultats différents d'une machine à une autre et la reproductibilité d'un même code n'était donc plus assurée. Pour plus de détails , voir ce message.

8.1.2 Astuce 2

Pour importer un gros fichier, nous conseillons d'utiliser dans un premier temps la fonction *read.table()* (ou *read.csv()*, etc) sur les quelques premières lignes du fichier (entre 20 et 100 lignes selon la taille du fichier), puis d'analyser la structure du jeu de données, plus particulièrement le type de chaque colonne :

```
bigfile_sample <- read.table("fichier.txt", stringsAsFactors = FALSE,
                             header = T, nrows = 20)
(bigfile_colclass <- sapply(bigfile_sample, class))
```

```
##     chiffre      lettre        date
##   "integer" "character" "character"
```

Dans un second temps, on importe la table en précisant le type de chaque colonne (par défaut l'option **stringsAsFactors = FALSE** donc ce n'est pas la peine de l'ajouter), ce qui aura pour effet de diminuer encore légèrement le temps de traitement.

```
system.time(bigfile_raw <- read.table("fichier.txt", header = T,
                    colClasses = bigfile_colclass))
```

```
## utilisateur     système      écoulé
##       0.258       0.000       0.258
```

Remarque 1 : si le type d'une colonne a été mal spécifié, cela pourra créer un message d'erreur.

Remarque 2 : si vous êtes sûr que le fichier ne contient pas de lignes de commentaires, vous pouvez encore améliorer le temps de lecture en précisant l'option **comment.char=" "**, ce qui permettra d'éviter de faire une recherche systématique de caractères de commentaires.

8.2 Le package readr

Nous présentons ici le package **readr**, faisant également partie du projet **tidyverse** et dont l'objectif est encore et toujours de rendre les codes plus simples et calculs plus rapides. Pour cela, les fonctions de ce package utilisent du code **C++** ce qui vous le verrez permet de faire des améliorations considérables en temps de calcul. Parmi les fonctions de ce package, *read_csv()* ou *read_table()* dont le but est bien entendu l'importation de fichiers **csv** ou **txt**. Elles ont été programmées de telle sorte qu'il suffit en général de les appeler en précisant uniquement le chemin d'accès du fichier à importer.

```
system.time(
 tibble.don <- read_table2("fichier.txt")
)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   `"chiffre"` = col_double(),
##   `"lettre"` = col_character(),
##   `"date"` = col_date(format = "")
## )
```

```
## utilisateur     système      écoulé
##       0.225       0.000       0.326
```

L'objet importé n'est pas un **data.frame** mais un **tibble** que nous avons déjà vu précédemment :

```
class(tibble.don)
```

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

```
object.size(tibble.don)
```

```
## 44004504 bytes
```

Ce type de format ne s'est pas encore généralisé à toutes les fonctions de **R**. Il se peut donc que vous rencontriez des difficultés pour utiliser certaines fonctions sur ce type d'objets. Pour passer du format **tibble** au **data.frame**, cela se fait très facilement au moyen de la fonction **as.data.frame**.

```
don <- as.data.frame(tibble.don)
```

Autres formats de données : dans la même lignée que **readr**, nous citons également le package **readxl** pour la lecture de fichiers *xls/xlsx*, ainsi que le package **haven** pour la lecture de données issue des logiciels **SPSS**, **Stata** ou **SAS**.

Bibliographie: pour l'usage du package **readr**, le lecteur pourra consulter ce document http://readr.tidyverse.org/.

8.3 Autres packages

Voici 3 packages, parmi d'autres, susceptibles d'aider l'utilisateur à manipuler des fichiers de données volumineux.

8.3.1 Package data.table

Ce package est un package concurent au projet **tidyverse**, l'objectif de ce package étant également de rendre les lignes de codes plus élégantes, les calculs plus rapides, notamment en présence de données volumineuses. Le package **data.table** est plus ancien que le projet **tidyverse**, mais ce dernier bénéficiant du support financier de **RStudio**, il semble à ce jour plus intéressant de choisir l'univers **tidyverse** qui devrait bénéficier de plus de développements par la suite. Nous présentons toutefois ici quelques exemples d'utilisation de **data.table**.

Pour importer un jeu de données :

```
require("data.table")
system.time(
objet.data.table <- fread("fichier.txt")
)
```

```
## utilisateur     système      écoulé
##       0.272       0.000       0.040
```

L'objet créé appartient à la classe d'objet **data.table** :

```
class(objet.data.table)
```

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

```
object.size(objet.data.table)
```

```
## 40001760 bytes
```

Pour manipuler ce format de données, il faut se familiariser avec une nouvelle syntaxe propre à ce genre de données. Le lecteur pourra consulter la note suivante s'il souhaite utiliser ce package : https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

8.3.2 Package ff

Pour des données trop volumineuses par rapport à la mémoire vive disponible de la machine, la package **ff** va faire en sorte de ne charger qu'une partie des données en mémoire vive que lorsqu'il en aura besoin.

Ici, on créé un fichier *"big_file.csv"* qui va contenir 50,000,000 observations et 3 colonnes. Pour faire cela, on va concaténer 100 fois **Donnees.a.importer** au fichier de sortie *big_file.csv* en utilisant la fonction *write_csv()*. L'opération peut prendre quelques minutes et c'est pourquoi pour informer l'utilisateur du temps restant, on propose d'utiliser la fonction *progress_estimated()*, qui fait avancer une barre au début de chaque nouvelle boucle (ici la boucle est répétée 100 fois).

```
write_csv(donnees_a_importer, "big_file.csv")

p <- progress_estimated(100)
for(k in 1:100){
  p$pause(0.1)$tick()$print()
  write_csv(donnees_a_importer, "big_file.csv", append = T)
}
```

La taille du fichier créé, d'environ 1.5Go est encore théoriquement manipulable sur **R** (essayer de le faire via la fonction *read.csv()*), mais on va supposer ici qu'on ne souhaite pas charger ce jeu de données intégralement en mémoire vive. Pour cela, on va utiliser la fonction *read.csv.ffdf()* du package **ff**. On spécifie comme options que l'on souhaite lire les données par groupe de 5,000,000 d'observations, excepté la première fois où l'on va lire 500,000 observations

```
require("ff")
bigDF <- read.csv.ffdf(file="big_file.csv", header = TRUE,
                       first.rows = 500000, next.rows = 5000000)
```

L'objet créé ne prend qu'une partie de la mémoire vive.

```
bigDF
object.size(bigDF)
```

L'idée de ce package est de stocker les variables non pas en mémoire vive, mais quelque part sur le disque dur et d'y accéder en utilisant des pointeurs. On peut effectuer un certain nombre d'opérations sur ces objets. Par exemple, pour calculer la moyenne de la variable **chiffre** :

```
library("ffbase")
mean.ff(bigDF$chiffre)
```

Remarque: on citera également le package **bigmemory** dont le principe est similaire. Plus récemment, le package **disk.frame** semble également promis à un bel avenir.

8.4 Algorithme de type Map/Reducce

On peut utiliser un algorithme de type *Map/Reduce* pour éviter d'importer un jeu de données trop volumineux sur la RAM. A la base, ce type d'algorithme s'applique sur des données qui sont organisées selon l'approche conceptuelle d'Hadoop, où les fichiers de données sont fractionnés en gros bloc et distribués à travers les noeuds d'un cluster.

Dans cet exemple, nous n'avons qu'un gros fichier de données, l'idée étant d'importer des échantillons de ce jeu de données sur lesquels on va appliquer la première étape *Map* de l'algorithme.

Par exemple, dans l'exemple ci-après, on a choisi d'importer les 5,000,000 premières observations, puis les 5,000,000 suivantes, etc. jusqu'à ce qu'on est parcouru tout le fichier. Sur chacun de ces échantillons, on va calculer d'une part la somme et d'autre part le maximum d'une variable quantitative.

Une fois réalisée cette étape sur les sous-échantillons, on va assembler les résultats obtenus à l'étape précédente. Pour calculer la moyenne, on va faire la somme sur les sommes obtenues et diviser ensuite par le nombre total d'observations et pour le max, on va calculer le maximum sur les maximum.

Remarque : on ne peut pas appliquer toutes les méthodes statistiques sur ce type d'algorithme (par exemple calculer un quantile nécessite de travailler sur l'échantillon complet). En revanche, ce type d'algorithme

peut fonctionner pour calculer l'estimateur des moindres carrés d'un modèle linéaire. Par ailleurs, le calcul parallèle (que nous verrons dans un autre chapitre) pourra être envisagé sur ce type d'algorithme.

```
n_split <- 11
ind <- 1
n_max <- 5000001
my_max <- numeric(n_split)
my_mean <- numeric(n_split)
my_n <- numeric(n_split)
for (k in 1:n_split) {
  split_don <- read_csv("big_file.csv", skip = ind, n_max = n_max,
                        col_names = c("chiffre", "lettre", "date"),
                        col_types = "ncc")
  my_max[k] <- max(split_don$chiffre)
  my_mean[k] <- sum(split_don$chiffre)
  my_n[k] <- nrow(split_don)
  ind <- ind + n_max
}
sum(my_mean) / sum(my_n)
max(my_max)
```

8.4.1 Package Matrix

Il est possible d'avoir suffisament de mémoires vives pour importer des données, mais pour certaines opérations algébriques et notamment le calcul matriciel, il y a des cas où on a besoin d'avoir plus de mémoires vives que ce dont nous disposons (typiquement une inversion de matrice). Dans ce cas-là, il est important de voir si les données sur lesquelles on travaille sont creuses ou non, c'est-à-dire contenant beaucoup de 0. Si c'est le cas, le package **Matrix** permet d'obtenir de bonnes performances en temps calcul et d'utiliser moins de RAM, grâce aux propriétés des matrices creuses.

Ce package s'utilise très simplement et la plupart des fonctions existantes pour le calcul matriciel (*crossprod()*, *solve()*, %>%) s'appliquent directement sur ce type d'objet. Pour plus d'informations sur ce package, consulter la vignette.

```
library("Matrix")
mat <- matrix(rbinom(10000, 1, 0.05), 100, 100)
object.size(mat)
Mat <- as(mat, "Matrix")
object.size(Mat)
```

8.5 Interaction avec des systèmes de gestion de bases de données

Lorsqu'une entreprise travaille sur de gros volumes de données, cela sous-entend qu'elle dispose de systèmes de gestion de base de données.

R dispose de plusieurs packages permettant d'intéragir avec des systèmes de gestion de base de données : **RODBC**, **RMySQL**, **RPostgresSQL**, **RSQLite** ainsi qu'avec des bases de données orientées document comme *MongoDB* et *couchDB* via les packages **mongolite** et **couchDB**.

Le gros avantage de ces packages est qu'ils permettent de laisser les bases de données trop grandes sur l'espace disque et d'aller récupérer uniquement l'information dont on a besoin en envoyant depuis **R** des requêtes qui seront exécutées sur les systèmes de gestion de base de données.

Bibliographie pour le traitement de données volumineuses, nous recommendons la lecture de ce document : https://rpubs.com/msundar/large_data_analysis

8.6 utiliser la syntaxe SQL

Pour celles et ceux qui sont familiers avec le langage SQL, le package **sqldf** permet d'utiliser la syntaxe SQL pour faire des requêtes depuis **R**.

Pour plus d'informations, voir : https://github.com/ggrothendieck/sqldf

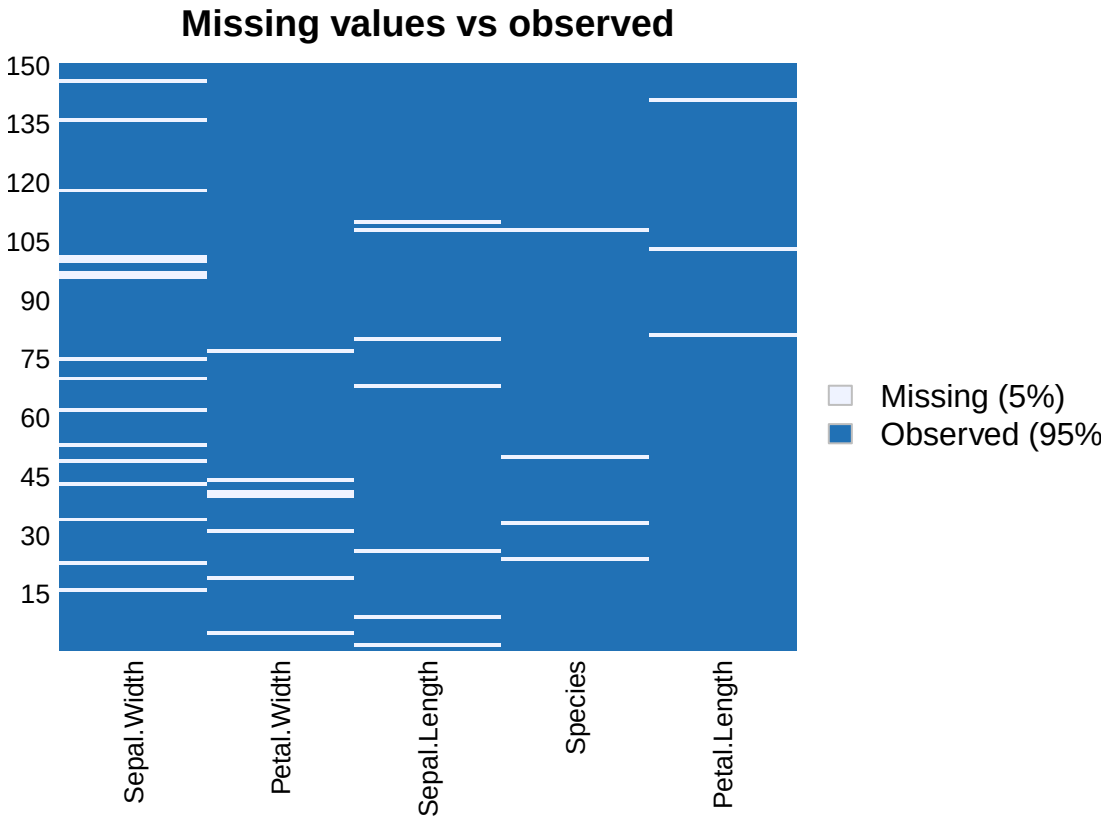
9 Visualiser et traiter les données manquantes

On présente dans ce paragraphe quelques packages qui permettent de visualiser et traiter les données manquantes. Pour commencer, nous allons générer des valeurs manquantes de façon aléatoire dans le jeu de données **iris** :

```
require("missForest")
iris.mis <- prodNA(iris, 0.05)
```

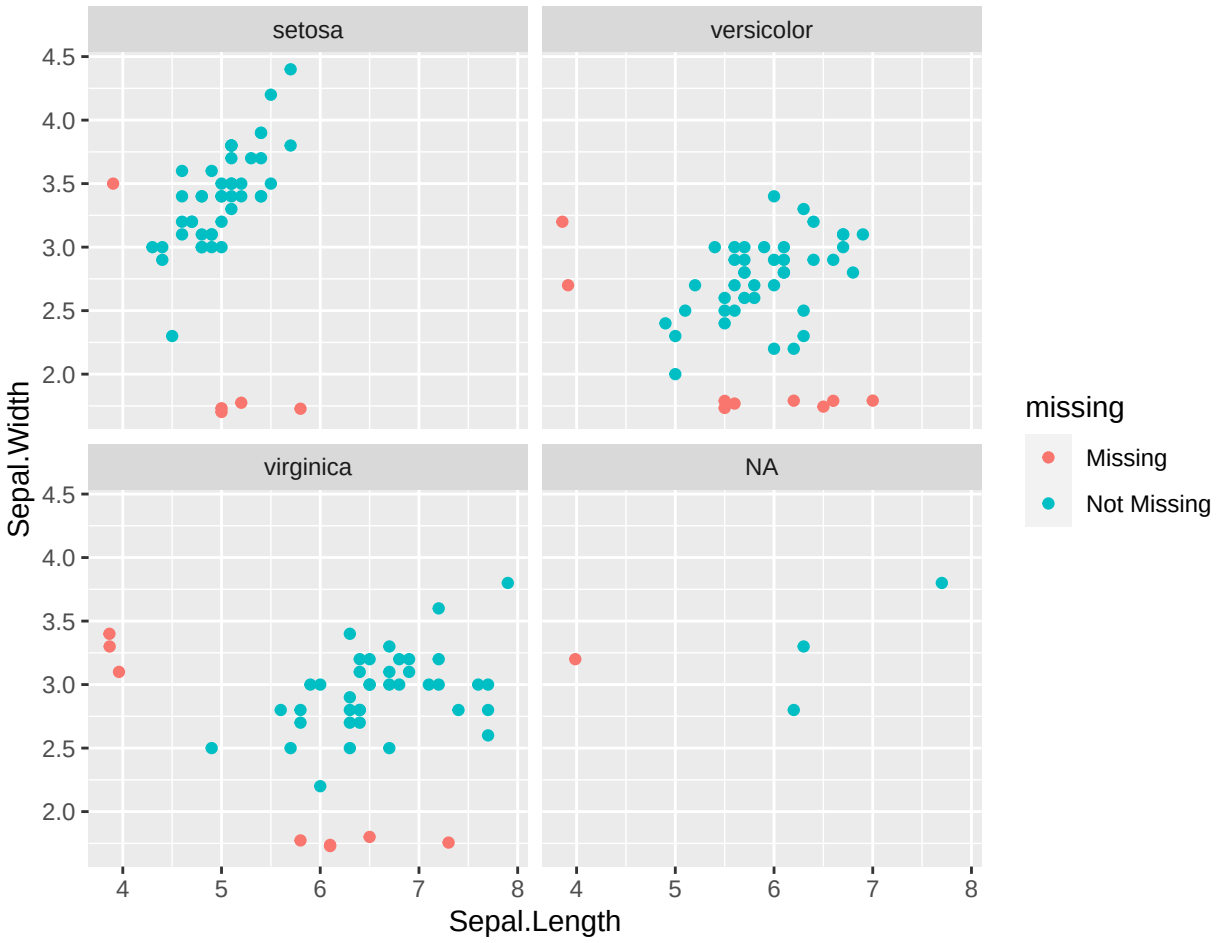
Le package **Amelia** permet de visualiser dans un graphique où sont localisées les données manquantes. Cela permet notamment de voir s'il existe un pattern ou une forme particulière (un bloc par exemple) parmi les valeurs manquantes :

```
require("Amelia")
missmap(iris.mis, main = "Missing values vs observed")
```



On notera également les packages **visdat** et **naniar** qui proposent plusieurs outils pour visualiser les données manquantes. Par exemple, pour savoir si les valeurs manquantes d'une variable sont liées à une autre variable, on peut utiliser l'outils suivant :

```
ggplot(iris.mis, aes(x = Sepal.Length, y = Sepal.Width)) +
  naniar::geom_miss_point() +
  facet_wrap(~Species)
```



Pour traiter les données manquantes, il existe plusieurs façons de procéder :

- ne rien faire. Dans ce cas, certaines fonctions comme la fonction *lm()* enlève automatiquement les observations dès lors qu'il existe au moins une valeur manquante parmi les variables à expliquer et explicatives :

```
res_lm <- lm(Sepal.Width ~ Sepal.Length + Petal.Length +
             Petal.Width + Species, data = iris.mis)
```

- supression des observations contenant au moins une valeur manquante en utilisant la fonction *subset()* ou *filter()* (du package **dplyr**)

```
iris.mis <- subset(iris.mis, !is.na(Species))
```

- imputation par la moyenne ou la médiane lorsqu'il s'agit d'une variable numérique ou alors par le mode lorsqu'il s'agit d'une variable qualitative.

```
ind_NA_Sepal.Length <- is.na(iris.mis$Sepal.Length)
mean_spec <- aggregate(Sepal.Length ~ Species,
                       data = iris.mis, FUN = mean, na.rm = T)
for (k in levels(iris.mis$Species)) {
 iris.mis[ind_NA_Sepal.Length & iris.mis$Species == k,
   "Sepal.Length"] <- mean_spec[mean_spec$Species == k, 2]
}
```

- imputation en utilisant des modèles linéaires afin de prédire les valeurs manquantes. Par exemple :

```
iris.imp_mean <- data.frame(sapply(iris.mis[, 1:4],
  function(x) ifelse(!is.na(x), x, mean(x, na.rm = T))),
  Species = iris.mis$Species)
pred <- predict.lm(res_lm, newdata = iris.imp_mean)
iris.mis[is.na(iris.mis$Sepal.Width), "Sepal.Width"] <-
  pred[is.na(iris.mis$Sepal.Width)]
```

- imputation en utilisant des modèles de prédiction sophistiqués issus du machine learning. Par exemple, les forêts aléatoires ou les K-plus proches voisins :

```
require("missForest")
iris.imp <- missForest(iris.mis)
```

```
##   missForest iteration 1 in progress...done!
##   missForest iteration 2 in progress...done!
##   missForest iteration 3 in progress...done!
##   missForest iteration 4 in progress...done!
```

```
require("VIM")
iris.knn <- kNN(iris.mis, k = 2)
```

10 Répertoires et fichiers

Il existe plusieurs fonctions de base qui permettent la gestion des répertoires et fichiers. L'utilisation de la plupart de ces fonctions est intéressante lors de l'écriture de scripts "propres" qui stockeront les résultats dans des fichiers bien rangés dans des répertoires bien nommés.

Parmi ces fonctions :

- la fonction *getwd()* retourne le chemin du répertoire de travail. Il s'agit du répertoire où seront sauvegardées par défaut les différentes opérations de sauvegarde (graphiques, codes, fichier historique, etc.) :

```
getwd()
```

```
## [1] "/media/thibault/My Passport/course/R_advanced/chapter_1"
```

- la fonction *dir()* retourne les fichiers et répertoires situés dans le chemin spécifié et éventuellement ses sous-répertoires (options **recursive = TRUE**). Par défaut, il s'agit du chemin correspondant à l'environnement de travail actuel (donné par *getwd()*) :

```
dir()
```

```
##  [1] "big_file.csv"                 "chapitre_1_avance_files"
##  [3] "chapitre_1_avance.html"       "chapitre_1_avance.log"
##  [5] "chapitre_1_avance.Rmd"        "chapitre_1_avance.tex"
##  [7] "data"                         "devoir 1"
##  [9] "devoir_1_correction.html"     "devoir_1_correction.pdf"
## [11] "devoir_1_correction.Rmd"      "devoir_1.pdf"
## [13] "devoir_1.Rmd"                 "DP15_Bvot_T1T2.txt"
## [15] "election.txt"                 "exercice1_en_correction.html"
## [17] "exercice1_en_correction.pdf"  "exercice1_en_correction.Rmd"
## [19] "exercice1_en_temp.Rmd"        "exercice1_en.html"
## [21] "exercice1_en.pdf"             "exercice1_en.Rmd"
## [23] "exercices_1.html"             "exercices_1.pdf"
## [25] "exercices_1.Rmd"              "fichier.txt"
## [27] "figure"                       "Figures"
```

```
## [29] "Graph"                    "markdown7.css"
## [31] "mattheus.R"               "mon_doc.html"
## [33] "mon_doc.Rmd"              "Num"
## [35] "slides"                   "Sorties"
## [37] "test-figure"              "test.html"
## [39] "test.md"                  "test.Rhtml"
## [41] "test.Rpres"               "twitter.R"
```

- la fonction *file.info()* prend comme argument d'entrée des chemins de fichiers ou répertoires et retourne des informations les concernant (taille, etc.) :

```
file.info(dir())
```

- la fonction *R.home()* retourne l'emplacement de **R** :

```
R.home()
```

```
## [1] "/usr/lib/R"
```

- la fonction *file.access()* donne des informations sur la permission accordée aux fichiers. Les permissions qui sont testées sont : existence (0), exécution (1), écriture (2) et lecture (4). La fonction retourne la valeur 0 pour oui et -1 pour non. Dans l'exemple qui suit, les fichiers qui sont situés dans le chemin où se situe **R** peuvent être lus et exécutés, mais sont en revanche protégés en écriture :

```
fic <- dir(file.path(R.home(), "bin"), full = T)
file.access(fic, 0)
file.access(fic, 1)
file.access(fic, 2)
file.access(fic, 4)
```

- la fonction *dir.exists()* teste l'existence d'un répertoire et la fonction *dir.create()* créé un répertoire.

```
dir.create("Sorties")
```

```
## Warning in dir.create("Sorties"): 'Sorties' existe déjà
```

```
dir.exists("Sorties")
```

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

Exercice 1.10.

écrire une fonction qui, à partir d'un nombre entier n passé en paramètre, effectue un tirage aléatoire de n valeurs selon une loi normale $\mathcal{N}(0,1)$ puis renvoie les valeurs générées dans un fichier d'un répertoire *Num* et trace une boxplot de ces données qui sera stockée dans un fichier *.jpg* du répertoire *Graph*.

11 Autour de l'espace de travail

La fonction *search()* donne la liste des packages (mais pas seulement) attachés à l'espace de travail. En gros, il s'agit des packages et environnements auxquels il est possible d'accéder dans la session en cours à l'instant t. Cette liste évolue bien entendu au fur et à mesure de la session :

```
search()
library("foreign")
search()
```

L'option **pos** de la fonction *ls()* permet de lister le contenu d'un package particulier en donnant sa position dans la liste renvoyée par *search()*. Vous pourrez en choisissant le bon indice, vous rendre compte de l'ensemble des fonctions accessibles depuis le package **base** :

```
ls(pos = which(search() == "package:base"))
```

Certains packages peuvent avoir des fonctions portant le même nom ; d'où le message d'avertissement ci-dessous au moment de charger un nouveau package, indiquant qu'un objet est masqué dans un package situé plus loin dans la liste *search()*.

```
library("pls")
```

```
##
## Attachement du package : 'pls'
```

```
## L'objet suivant est masqué depuis 'package:stats':
##
##     loadings
```

```
search()
```

```
##  [1] ".GlobalEnv"           "package:pls"          "package:VIM"
##  [4] "package:grid"         "package:colorspace"   "package:Amelia"
##  [7] "package:Rcpp"         "package:missForest"   "package:itertools"
## [10] "package:iterators"    "package:foreach"      "package:randomForest"
## [13] "package:data.table"   "package:forcats"      "package:dplyr"
## [16] "package:purrr"        "package:readr"        "package:tidyr"
## [19] "package:tibble"       "package:ggplot2"      "package:tidyverse"
## [22] "package:gplots"       "package:zoo"          "package:classInt"
## [25] "package:glue"         "package:stringr"      "package:wordcloud"
## [28] "package:RColorBrewer" "package:stats"        "package:graphics"
## [31] "package:grDevices"    "package:utils"        "package:datasets"
## [34] "package:methods"      "Autoloads"            "package:base"
```

Dans ce cas, si on souhaite obtenir l'aide de la "bonne" fonction, il suffit de précéder le nom de la fonction par le nom du package suivi de *::*.

```
find("loadings")
```

```
## [1] "package:pls"   "package:stats"
```

```
?loadings
```

```
## Help on topic 'loadings' was found in the following packages:
##
##   Package               Library
##   pls                   /home/thibault/R/x86_64-pc-linux-gnu-library/4.1
##   stats                 /usr/lib/R/library
##
##
## Utilisation de la première correspondance ...
```

```
?stats::loadings
```

Remarque : on pourra procéder de la même façon pour être sûr d'exécuter la "bonne" fonction.

On a vu précédement que pour accéder à un élément d'une **list** ou d'un **data.frame**, il est nécessaire d'appeler cet objet suivi de l'opérateur **$**. Il existe au moins deux façons de simplifier cette opération. La première consiste à utiliser la fonction *with()* :

```
with(airquality,
     summary(Ozone))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
```

```
##    1.00   18.00   31.50   42.13   63.25  168.00      37
```

La seconde consiste à utiliser la fonction *attach()* qui permet d'attacher tous les éléments d'un objet dans l'environnement de travail comme s'il s'agissait d'un package.

```
e <- list(e_vec_x = rnorm(10),
          e_fic1 = function(x) mean(x),
          e_fic2 = function(x) mean((x-e_fic1(x))^2))
attach(e)
search()
```

```
##  [1] ".GlobalEnv"           "e"                    "package:pls"
##  [4] "package:VIM"          "package:grid"         "package:colorspace"
##  [7] "package:Amelia"       "package:Rcpp"         "package:missForest"
## [10] "package:itertools"    "package:iterators"    "package:foreach"
## [13] "package:randomForest" "package:data.table"   "package:forcats"
## [16] "package:dplyr"        "package:purrr"        "package:readr"
## [19] "package:tidyr"        "package:tibble"       "package:ggplot2"
## [22] "package:tidyverse"    "package:gplots"       "package:zoo"
## [25] "package:classInt"     "package:glue"         "package:stringr"
## [28] "package:wordcloud"    "package:RColorBrewer" "package:stats"
## [31] "package:graphics"     "package:grDevices"    "package:utils"
## [34] "package:datasets"     "package:methods"      "Autoloads"
## [37] "package:base"
```

Ensuite, il est possible d'accéder directement aux éléments de la liste sans avoir à appeler l'objet **e** :

```
e_fic1(e_vec_x)
```

```
## [1] 0.6690173
```

```
e_fic2(e_vec_x)
```

```
## [1] 0.9399229
```

```
detach(e)
```

Attention : il faut être très prudent avec la fonction *attach()*, car cela peut créer des conflits avec l'environnement global.

Remarque : il est parfois utile de connaître certains détails de l'environnement de travail notamment pour comprendre *pourquoi ça ne marche pas !!!* (voir dessin ci-dessous). Les fonctions *sessionInfo()* et *R.Version()* peuvent permettre d'identifier certaines incompatibilités entre notre version de **R** et un package particulier que l'on vient d'installer.

Exercice 1.11.

Q1 Quel est l'inconvénient illustré par les manipulations de *e_fic1()*, **e_vec_x** et *e_fic2()* ?

Q2 à quoi sert l'opérateur **::** ?

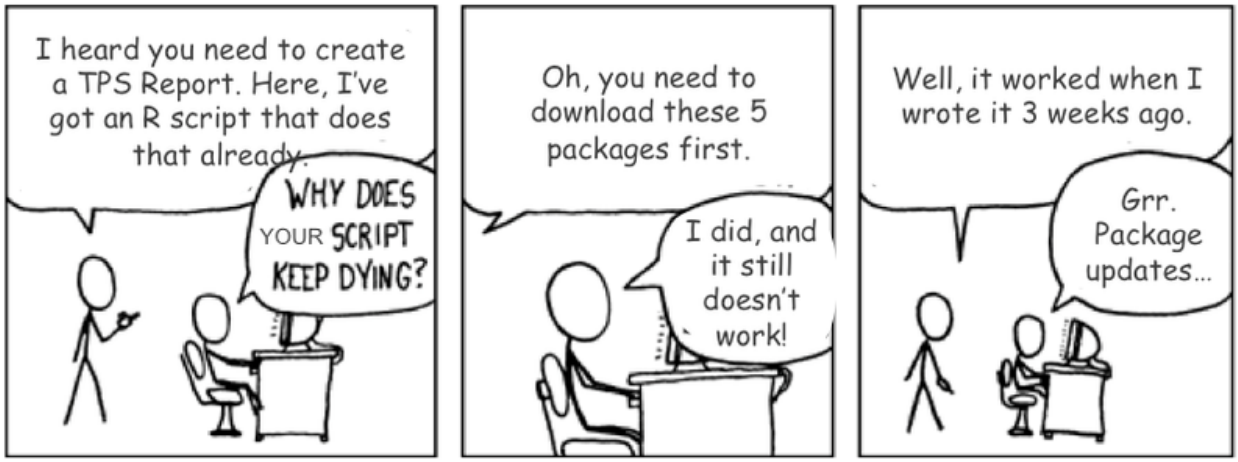


Figure 1: ça ne marche pas !!!